universidad de león

Escuela de Ingenierías
Industrial e Informática

Departamento de Ingeniería Eléctrica
y de Sistemas y Automática

TESIS DOCTORAL

Meta-algoritmos de ordenación

Tesis dirigida por el
Dr. D. Honorino A. Mielgo Álvarez

Rodrigo Raposo García
León, Noviembre de 2015

UNIVERSIDAD DE LEÓN

Resumen

Escuela de Ingenierías Industrial e Informática
Departamento de Ingeniería Eléctrica y de Sistemas y Automática

Meta-algoritmos de ordenación

Rodrigo Raposo

Una de las tareas fundamentales de los sistemas de información es la ordenación, clasificación y búsqueda de datos, a la que dedican entre el 25 y el 50 por ciento de su tiempo. Habitualmente la ordenación no es un fin en sí mismo, sino más bien una tarea fundamental que se encuentra entre dos procesos: un productor que genera elementos para un consumidor que los demanda según un orden preestablecido. Esta *Tesis* analiza minuciosamente la forma de trabajo de los algoritmos clásicos de ordenación, y define una nueva taxonomía basada en la abstracción de sus respectivas implementaciones de las estructuras de control y de datos. Se aborda el concepto ordenación en su forma más amplia y pura, describiendo la solución de forma independiente a la arquitectura hardware del sistema y al lenguaje de programación elegido.

UNIVERSIDAD DE LEÓN

Abstract

Escuela de Ingenierías Industrial e Informática
Departamento de Ingeniería Eléctrica y de Sistemas y Automática

Sorting meta-algorithms

Rodrigo Raposo

The fundamental core task in Information Systems is to sort, classify and search data, which spend between 25 and 50 percent of his time. Usually the sort task is not and end in itself, but rather a fundamental task that lies between two processes: a producer that generates elements for a consumer which demand it according to an predeterminated order. This *Thesis* thoroughly analyses how sort classic algorithms work, and defines a new taxonomy based on abstract their respective implementation of control and data structures. Sorting is addressed in its broadest and pure form and a solution is offered independently of the hardware architecture and the chosen programming language.

Agradecimientos

En primer lugar quiero mostrar el mayor agradecimiento a mí director de *Tesis* D. Honorino A. Mielgo Álvarez por todas las horas invertidas ordenando, desordenando y reordenando nuevamente elementos. Sin su ayuda, orientación y apoyo esta *Tesis* no hubiera sido posible.

A todos los profesores que he tenido a lo largo de mi vida académica por la transmisión de conocimientos y valores fundamentales para la vida. De entre todos, me gustaría recordar con un cariño especial a Mario Hernández por su fe y aliento para con todos sus alumnos. Descansa en paz, Maestro.

A todos los compañeros y amigos por los buenos ratos y experiencias vividas. Juntos hemos disfrutado momentos inolvidables, aunque sin duda alguna los mejores son los que están por llegar.

A toda mi familia, sin la cual jamás habría sido posible llegar hasta aquí. A los que tengo la suerte de tener cada día a mí lado y a los que ya no están conmigo: todos vosotros formáis mi vida y mis recuerdos. Con vuestra ayuda y comprensión los momentos difíciles se hacen mucho más llevaderos.

A todos y cada uno de vosotros sólo puedo deciros sinceramente

GRACIAS

Índice general

Resumen III

Abstract V

Agradecimientos VII

Índice general VIII

Índice de figuras XVII

Índice de algoritmos XXI

Dedicatoria XXIII

1. Introducción **1**
1.1. Fundamentos de ordenación: Breve reseña histórica 1

2. Ordenación clásica: fundamentos básicos **5**
2.1. Algoritmo: definición 5
2.1.0.1. Algoritmos de ordenación clásicos 6
2.2. Un poco de historia 7
2.2.1. Una visión clásica 7
2.2.2. Ordenación interna 8
2.2.3. Ordenación externa 9
2.3. Ordenación serie y paralelo 10
2.3.1. Introducción 10
2.4. Algoritmos de ordenación serie en paralelo 14
2.4.1. Procesamiento paralelo del algoritmo serie *Burbuja* . . . 15

2.4.2. Procesamiento paralelo del algoritmo serie *Fusión* . . . 17
2.5. Algoritmos de ordenación en bloque . . . 19
2.5.1. Una visión inicial . . . 19
2.5.2. Alternativa . . . 20
2.5.3. Reflexiones sobre la parte hardware introducida en el algoritmo . . . 22
2.5.4. Reflexiones sobre las mejoras introducidas en el software del algoritmo . . . 24

3. Meta-algoritmos de ordenación: pilares fundamentales 25
3.1. Introducción . . . 25
3.2. Abstracción . . . 27
3.2.1. El papel de la abstracción . . . 27
3.2.1.1. La abstracción como un proceso mental natural 28
3.2.1.2. La abstracción en la ingeniería del software . . 29
3.2.1.3. Tipos de datos . . . 30
3.2.1.4. Ventajas de los tipos abstractos de datos . . . 30
3.3. Construcción de meta-algoritmos . . . 31
3.3.1. La abstracción como principio fundamental . . . 31
3.3.1.1. Fundamentos de la investigación . . . 32
3.4. Análisis del problema . . . 34
3.4.1. Ángulos . . . 34
3.4.1.1. Primeros intentos del proceso de mejora 35
3.4.1.2. Meta-algoritmos de ordenación: fundamentos . 37
3.4.1.3. Para-algoritmos . . . 38
3.5. Convenciones . . . 39
3.5.1. Elementos repetidos: una restricción que no es tal . . . 41
3.6. Optimalidad frente a simplicidad y seguridad . . . 43
3.6.1. Situación de Strassen . . . 44
3.7. Obtención de resultados en orden inverso . . . 45
3.8. La pregunta más inteligente . . . 46
3.8.1. Estrategias . . . 47
3.8.2. Consideraciones . . . 47
3.9. Visión relacional del problema: representaciones . . . 49
3.9.1. Grafo dirigido acíclico . . . 51
3.9.1.1. Ejemplo de construcción de grafo dirigido acíclico 51
3.9.2. Matriz de incidencia . . . 54
3.9.2.1. Ejemplo de representación mediante la matriz de incidencia . . . 60

3.9.2.2. Propiedades de la representación mediante matriz de incidencia . . . 63
3.10. Transitividad . . . 65
3.10.1. Ejemplo sin transitividad: *Burbuja* . . . 66
3.10.2. Ejemplo de transitividad en *Binario* . . . 66

4. Meta-algoritmo descendente **71**
4.1. Introducción . . . 71
4.2. Cómo empezó todo . . . 72
4.3. Definiciones básicas . . . 73
4.4. Versión intuitiva . . . 74
4.5. Representación gráfica . . . 76
4.5.1. Primera fase: creación del árbol de procesos . . . 76
4.5.2. Segunda fase: recuperación de elementos . . . 79
4.6. Versión formal . . . 80
4.7. Corrección del meta-algoritmo descendente: versión informal . . . 85
4.8. Corrección del meta-algoritmo descendente: versión formal . . . 86
4.8.1. El proceso termina . . . 86
4.8.2. La condición invariante . . . 88
4.8.3. El estado inicial cumple el invariante . . . 88
4.8.4. Las transiciones conservan el invariante . . . 88
4.8.5. En los estados finales el objetivo se satisface . . . 88
4.8.6. Indeterminaciones . . . 88
4.9. Optimizaciones . . . 89
4.10. Derivación de *Quicksort* . . . 89
4.11. Derivación en *Binario* . . . 95
4.12. Simetría o dualidad entre *Quicksort* y *Binario* . . . 96
4.13. Optimizaciones específicas . . . 96
4.13.1. *Binario* . . . 96
4.13.2. *Quicksort* . . . 97
4.14. Consideraciones finales . . . 97

5. Meta-algoritmo ascendente **99**
5.1. Introducción . . . 99
5.2. Versión intuitiva . . . 100
5.3. Representación gráfica . . . 101
5.3.1. Primera fase: Distribución de elementos . . . 101
5.3.2. Segunda fase: fusión de elementos . . . 103
5.4. Versión formal . . . 105

5.5. Corrección del meta-algoritmo: versión informal 109
5.6. Corrección del meta-algoritmo: versión formal 110
5.7. Indeterminaciones 111
5.8. Optimizaciones genéricas 111
5.9. Derivaciones 112
5.9.1. Derivación de *Fusión* 113
5.9.2. Derivación de *Torneo* 114
5.10. Simetría o dualidad 114
5.11. Optimizaciones específicas 115
5.11.1. *Torneo* 115
5.11.2. *Fusión* 115

6. Esquemas **117**
6.1. Introducción 117
6.2. Esquemas 117
6.3. Germen de la idea 118
6.4. Esquemas generales 120
6.4.1. Resultado de la comparación $a < b$ 121
6.4.2. Resultado de la comparación $a > b$ 122
6.4.3. Algoritmo *Burbuja* 125
6.5. Esquemas de árboles 127
6.5.1. Ejemplo con elementos almacenados de forma secuencial 128
6.5.1.1. Primera posibilidad: $x < y$ 129
6.5.1.2. Segunda posibilidad: $x > y$ 131
6.5.2. Conclusiones 132
6.6. Esquemas de cadenas 133
6.7. Esquemas de cadena única 133
6.8. Una jerarquía 134
6.9. Acciones 134

7. Miscelánea **135**
7.1. Dualidad entre meta-algoritmos 135
7.1.1. Dualidad en el propio proceso 135
7.1.2. Descendente 136
7.1.3. *Quicksort* y *Binario* 137
7.1.4. Ascendente 137
7.1.5. Dualidad en la forma de abordar el problema 137
7.1.5.1. Asimetrías 138
7.1.5.2. Inserción de elementos 138

7.1.5.3. Fase de realización del trabajo 138
7.2. Un ejemplo práctico: concordancias 140
7.2.1. Análisis inicial 140
7.2.2. Situaciones 141
7.2.3. Posibilidades de solución 142
7.2.4. Conclusiones 143
7.3. Análisis exhaustivo de un algoritmo: *Burbuja* 143
7.3.1. Versión clásica 144
7.3.2. Versión mejorada 144
7.3.3. Análisis 145
7.3.4. Análisis gráfico 145
7.3.5. Versión optimizada 147
7.3.6. Vista como meta-algoritmo descendente 151
7.3.7. Vista como esquema de cadena 152
7.3.8. Vista como esquema de árbol 152
7.3.9. Posibilidades de paralelismo 152
7.3.10. Selección de prioridades 153
7.4. Otra versión: *Burbuja* inversa 153
7.5. Comparaciones e intercambios 154
7.6. Sus ventajas 154
7.7. Resumen de las variaciones posibles 155
7.8. Análisis de la eficiencia de los algoritmos clásicos 155
7.9. Análisis de una situación: paralelismo 157
7.9.1. *Quicksort* 158
7.9.2. *Binario* 159
7.9.3. Meta-algoritmo descendente 159
7.9.4. *Torneo* y *Fusión* 160
7.9.5. Meta-algoritmo ascendente 161
7.10. Observaciones 161
7.10.1. El primero lo antes posible 161
7.10.2. Los elementos llegan de tarde en tarde 162

8. Conclusiones **163**
8.1. Conclusiones 163
8.2. Desarrollos futuros 164

Apéndices **166**

A. Inserción de un elemento en una cadena **167**

B. Inserción binaria vs inserción secuencial: número medio de comparaciones **169**
B.1. Inserción binaria 169
B.2. Inserción secuencial 170
B.3. Conclusiones 170

C. Inserción por la cabeza vs inserción central: información obtenida con cada pregunta **173**
C.1. Introducción 173
C.2. Principio de entropía de Shannon 174
C.3. Inserción de un elemento por la cabeza 174
C.4. Inserción de un elemento por el centro de la cadena 176
C.5. Conclusiones 177

D. Optimalidad de la equipartición en *Fusión* **179**

E. Inserción binaria de la cabeza vs inserción binaria del elemento central **181**
E.1. Reducción de la incertidumbre 182
E.2. Dispersión de elementos 184
E.3. Conclusiones 187

F. Ordenación por *Selección* **189**

G. Ordenación por intercambio o *Burbuja* **191**

H. Ordenación por *Inserción* **195**
H.0.1. *Inserción binaria* 196
H.0.2. Número de comparaciones 197

I. Algoritmo de ordenación *Shell* **199**

J. Algoritmo de ordenación *Quicksort* **203**

K. Algoritmo de ordenación *Heap* **209**

L. Algoritmo de ordenación *Merge* **217**

M. Algoritmo de ordenación utilizando árbol *Binario* **219**
M.1. Especificación del *TAD* árbol binario 220

M.2. Implementación 221
M.2.1. Liberar memoria ocupada por árbol binario 222
M.2.2. Recorrer un árbol binario 222
M.3. Árbol binario de búsqueda 223
M.3.1. Inserción de un elemento en árbol binario de búsqueda . 223
M.3.2. Buscar un elemento en un árbol binario de búsqueda . . 224

N. Algoritmo de *Torneo* **225**

Bibliografía **227**

Indice alfabético **230**

Índice de figuras

2.1. Ordenación en bloque: visión inicial 20
2.2. Ordenación en bloque: comparando elementos adyacentes . . . 20
2.3. Primera comparación procesadores pares con adyacentes 21
2.4. Primera comparación procesadores impares con adyacentes . . 21
2.5. Segunda comparación procesadores pares con adyacentes 22
2.6. Segunda comparación procesadores impares con adyacentes . . 22

3.1. Inserción binaria de un elemento 48
3.2. Grafo dirigido acíclico: situación inicial 52
3.3. Grafo que representa una situación intermedia 53
3.4. Grafo que representa un estado final de orden total 53
3.5. Matriz de incidencia 54
3.6. (A) Matriz estado inicial (B) Matriz estado final 55
3.7. Estados inicial y final: sólo elementos necesarios 56
3.8. Ejemplo de representación mediante grafo dirigido acíclico . . . 57
3.9. Primera representación matricial de un orden parcial 58
3.10. Segunda representación matricial de un orden parcial 58
3.11. Tercera representación matricial de un orden parcial 58
3.12. Nueva representación matricial con $a < d$ 59
3.13. Matriz de incidencia en el inicio del proceso 61
3.14. Matriz de incidencia en un instante t del proceso 61
3.15. Matriz de incidencia en un instante $t' = t+1$ 62
3.16. Matriz de incidencia al finalizar el proceso de ordenación 63
3.17. El elemento x no puede ocupar una posición posterior 63
3.18. El elemento y no puede ocupar una posición anterior 64
3.19. El elemento z ocupa su posición definitiva 64
3.20. Ejecución algoritmo *Binario*: grafo situación intermedia 67
3.21. Ejecución algoritmo *Binario*: matriz situación intermedia . . . 67
3.22. Ejecución algoritmo *Binario*: añadida información $a < g$ 68
3.23. Ejecución algoritmo *Binario*: añadida información $a < c$ 68

3.24. Ejecución algoritmo *Binario*: añadida información $a < b$ 69

4.1. Procesos situación intermedia meta-algoritmo descendente . . . 74
4.2. Procesos (incluso del mismo nivel) en ejecución simultánea . . . 75
4.3. Situación inicial: único proceso 77
4.4. Proceso raíz una vez seleccionado el pivote 77
4.5. ¿Cómo es e' respecto al elemento pivote e? 77
4.6. ¿El elemento e' es menor que el elemento e? 78
4.7. ¿El elemento e' es mayor que el elemento e? 78
4.8. Árbol final del meta-algoritmo descendente 79
4.9. Transición si no existe proceso hijo izquierdo 81
4.10. Transición si inicialmente existe proceso hijo izquierdo 81
4.11. Transición si no existe proceso hijo derecho 82
4.12. Transición si inicialmente existe proceso hijo derecho 82
4.13. Transición desde el conjunto vacío 82
4.14. Transmisión señal de finalización al proceso izquierdo 83
4.15. Transmisión señal de finalización al proceso derecho 83
4.16. Transmisión de la salida proceso izquierdo hasta la raíz 83
4.17. Transmisión del elemento pivote a la salida 84
4.18. Transmisión de la salida proceso derecho hasta la raíz 84
4.19. Desaparición del proceso P 84
4.20. Matriz *Quicksort* tras la primera comparación directa 91
4.21. Matriz *Quicksort* tras añadir información por transitividad . . 92
4.22. Matriz *Quicksort* tras seleccionar segundo y tercer pivote . . . 92
4.23. Matriz *Quicksort* añadida información por transitividad 93
4.24. Matriz *Quicksort* con la que comienza la siguiente iteración . . 93
4.25. *Quicksort*: grafo tras las situaciones descritas 94
4.26. *Quicksort*: transición en el árbol de procesos 94

5.1. Situación inicial meta-algoritmo ascendente 101
5.2. Ejemplo meta-algoritmo ascendente, entrada {G,A,B,R,I,E,L} . 102
5.3. Meta-algoritmo ascendente finalizada la fase de distribución . . 103
5.4. Meta-algoritmo ascendente en la fase de fusión 104
5.5. Meta-algoritmo ascendente finalizado 105
5.6. Proceso raíz meta-algoritmo ascendente ($\mathcal{C} = \mathcal{E}$) 106
5.7. Cambio de estado de un proceso: de activo a finalizado 106
5.8. Fusión para $q < r$ 107
5.9. Fusión para $q > r$ 107
5.10. Transferencia de la salida cuando está vacía en un proceso . . . 108

5.11. Paso de un proceso a estado finalizado 108
5.12. Estado en que el proceso es eliminado liberando memoria . . . 108

6.1. Situación intermedia de trabajo con esquema general 121
6.2. Se seleccionan 2 elementos a, b con los que continuar 121
6.3. Se añade la información directa y la clausura transitiva 122
6.4. Conjuntos $\mathcal{A}$, $\mathcal{B}$ y $\mathcal{C}$ que se han de intercambiar 123
6.5. Diferenciación de conjuntos en la matriz de incidencia 123
6.6. Zonas que no se tocan (rojo) y sobre la que se trabaja (verde) . 124
6.7. Elementos tras la comparación y reorganización 124
6.8. Situación de elementos y conjuntos antes de la comparación . . 125
6.9. Situación de elementos y conjuntos tras la reorganización . . . 125
6.10. Algoritmo *Burbuja*: Situación intermedia 126
6.11. Si $a < b$, se añade toda la información conocida 126
6.12. Si $a > b$ se permutan elementos 127
6.13. Ejemplo de organización secuencial 129
6.14. Organización secuencial expresada en forma de árbol 129
6.15. Organización del árbol tras incorporar la información $x < y$ . . 130
6.16. Posibles situaciones del árbol tras incorporar $x > y$ 131
6.17. Información: situación final utilizando la primera alternativa . . 131
6.18. Información: situación final utilizando la segunda alternativa . 132

7.1. (A) *Merge* antes de la última fusión (B) *Quicksort* tras la primera división 139
7.2. *Burbuja*: información almacenada tras k iteraciones 146
7.3. *Burbuja*: grafo que representa la situación tras k pasadas . . . 146
7.4. *Burbuja*: información real obtenida tras k comparaciones 147
7.5. *Burbuja*: grafo que refleja toda la información obtenida 147
7.6. *Burbuja* optimizada: grafo de una situación intermedia 148
7.7. *Burbuja* optimizada: se compara Y con Z 149
7.8. *Burbuja* optimizada: nueva situación si $y < z$ 149
7.9. *Burbuja* optimizada: nueva situación si $y > z$ 149

A.1. ¿Con qué elemento y se compara el elemento x? 168

B.1. Numero medio de comparaciones para cada tipo de inserción . 171

C.1. Inserción de un elemento por la cabeza de una cadena 175
C.2. Inserción de un elemento por el punto medio de una cadena . . 176

E.1. Situaciones A y B: elemento cabecera vs elemento medio 183

E.2. Q: fusión elemento cabeza vs fusión elemento medio 184
E.3. Cálculos obtenidos con valores de $P = 10$ y $Q = 5$ 186
E.4. Función dispersión de elementos para situaciones A y B 186

G.1. Lista de elementos desordenada 191

H.1. Lista de elementos desordenada 195
H.2. Insertando un elemento en una lista parcialmente ordenada . . 196

I.1. Lista de elementos parcialmente desordenada 199
I.2. Comparación inserción directa vs *Shell* 200

J.1. Lista de elementos a ordenar 204
J.2. Elementos tras el primer intercambio 204
J.3. Elementos tras el segundo intercambio 204
J.4. Elementos divididos por el pivote 205
J.5. Resultado final: elementos ordenados 205

K.1. Conjunto de elementos a ordenar 212
K.2. Árbol montículo *Max* inicial 212
K.3. El elemento más profundo y a la derecha es enviado a la raíz . 213
K.4. Intercambia elemento 3 para mantener el invariante 213
K.5. Ejemplo de ordenación con algoritmo *Heap* 214
K.6. Continuación ejemplo de ordenación con algoritmo *Heap* 215

Índice de algoritmos

1. Algoritmo clásico *Burbuja* en *ADA* 144
2. Algoritmo *Burbuja* mejorado en *ADA* 145
3. *Burbuja* sin pérdida de información: código en *ADA* 150
4. Algoritmo *Burbuja* inverso: código en *ADA* 153
5. Pseudocódigo algoritmo de ordenación por *Selección* 190
6. Procedimiento para intercambiar elementos 190
7. Pseudocódigo algoritmo de intercambio o *Burbuja* 192
8. Pseudocódigo algoritmo de intercambio o *Burbuja* mejorado . . 193
9. Pseudocódigo algoritmo de ordenación por *Inserción* 196
10. Pseudocódigo algoritmo ordenación *Shell* 201
11. Pseudocódigo *Quicksort* versión inicial 207
12. Pseudocódigo algoritmo *Quicksort* detallado 208
13. Pseudocódigo algoritmo *Heap* 211
14. Pseudocódigo algoritmo *Merge* 218
15. Estructura de datos *TAD* árbol binario 221
16. Procedimiento para liberar la memoria de árbol binario 222
17. Recorrido de árbol binario en *inorden* 223
18. Insertar elemento en árbol binario de búsqueda 224
19. Buscar un elemento en un *Árbol binario de búsqueda* 224

Dedicado a mi familia, en especial
a mis dos amores: Marta y Gabriel

Capítulo 1

Introducción

1.1. Fundamentos de ordenación: Breve reseña histórica

En el mundo actual, los sistemas informáticos emplean gran parte de su tiempo en operaciones de búsqueda, clasificación y mezcla de datos. Las operaciones de cálculo numérico y sobre todo de gestión, requieren por norma general de operaciones que clasifiquen datos. Por este motivo son innumerables las ocasiones en que las operaciones de clasificación, sin ser un fin en sí mismo, son acciones necesarias para realizar otras tareas. Según un estudio los sistemas informáticos dedican aproximadamente entre el 25 % y el 50 % de su tiempo a tareas de ordenación y búsqueda.

En la visión clásica de los algoritmos de ordenación se realizan diversas clasificaciones de estos, todas ellas atendiendo a las estructuras de datos en que son construidos, a las estructuras de control propias del lenguaje de programación utilizado y a las características particulares en que se encuentra almacenada la información a ordenar. Por este motivo las clasificaciones de los algoritmos de ordenación se basan única y exclusivamente en parámetros de implementación: ordenación interna/externa, operaciones de inserción/intercambio/extracción...

Estas clasificaciones de los algoritmos (más efectivas o acertadas en unos casos que en otros) reflejan únicamente detalles de la cristalización final de los mismos[1] alejándose de la naturaleza del problema de ordenación subyacente. El hecho de que un algoritmo de ordenación requiera la utilización de almacenamiento externo, por ejemplo, lo incluye en el mismo "saco" que a otros cuya forma de abordar el problema es completamente diferente, por el mero hecho de compartir una característica ajena al proceso de ordenación como es el lugar en que se almacena la información. El trabajo realizado en esta *Tesis*

Doctoral pretende dar una visión exclusivamente centrada en el concepto de ordenación, abstrayendo las implementaciones de los algoritmos de ordenación "clásicos"[2] de forma que no exista en su definición y clasificación, ningún tipo de dependencia del conjunto de elementos a ordenar, de las estructuras de datos utilizadas para su definición y almacenamiento, y realizando una clara separación[3] de las estructuras de control utilizadas por el lenguaje de programación elegido. Se trata de conseguir llegar de la forma más pura posible, a la esencia de la ordenación de una colección de elementos, deshaciéndose de todo aquello que enmascara el fundamento y la base del concepto ordenación y que es superfluo en su definición.

El principal objetivo es el de obtener las condiciones necesarias para realizar la tarea de ordenar una colección de elementos. Dichas condiciones han de ser mínimas, pero a la vez necesarias y completas para abordar la tarea de la forma más independiente posible a factores ajenos.

El algoritmo que se diseña debe cumplir una serie de condiciones mínimas que no sean débiles, que permitan realizar el proceso de ordenación de forma correcta e independiente. Una unificación en la definición de las condiciones, así como de los diferentes pasos a seguir para realizar el proceso siendo reducidos estos a la mínima expresión posible, permite realizar una clasificación abstracta frente a las tradicionales que se recogen en la mayoría de bibliografías universales fundamentales y de referencia dentro del mundo de la informática.

Las citadas clasificaciones siguen heredando las restricciones impuestas en los primeros años de la informática. La realización de las tareas de ordenación estaba completamente determinada por las características hardware de los equipos[4], así como por la forma de obtener y tratar las diferentes colecciones de elementos almacenados en los sistemas de almacenamiento masivo de información. La eficiencia primaba sobre la abstracción y la implementación condicionaba todo análisis.

El propósito de trasladar el concepto de "abstracción"[1] un grado por encima de las propias estructuras de datos y de control utilizadas por los algoritmos de ordenación, permite construir meta-algoritmos genéricos dinámicos cuya determinación depende única y exclusivamente del problema a tratar: la búsqueda de una relación de orden total de un conjunto de elementos. El problema nada tiene que ver con el lugar en que estos se encuentren almacenados físicamente o las estructuras de datos necesarias para obtener la relación de

[1] En el sentido más amplio posible de la palabra.

orden buscada.

La construcción de un meta-algoritmo genérico permite incluir en el mismo una parte de dinamismo en el que no se determina o impone con rigidez ni la secuencialidad de procesos, ni el orden de tratamiento de los elementos que forman el conjunto de entrada. Esto permite adaptar, en la medida de lo posible, el meta-algoritmo a las características del productor/consumidor para el caso concreto de cada aplicación.

Si se permite un cierto grado de ineficiencia para mejorar la eficacia y se realiza un uso adecuado de paralelismo (concurrencia, colateralidad) los objetivos son:

- Obtener un tiempo de respuesta mínimo (*delay*) en la resolución del problema.
- Eficiencia a optimizar: se busca reducir el grado de redundancia entre procesos. Si dos procesos diferentes repiten uno la tarea del otro se están desperdiciando recursos. Reducir la dependencia entre procesos permite obtener una independencia entre las tareas que se han de realizar y, como ventaja colateral, se reducen los costes de comunicación entre ellos.

Como se desarrolla en el transcurso del trabajo, uno de los defectos más graves que presentan algunos algoritmos de ordenación es la repetición de preguntas cuya respuesta es deducible a partir de la información obtenida con anterioridad. Estas preguntas se califican sin ningún género de duda como superfluas (inútiles) para avanzar hacia la solución final. Por ello no se consideran procedimientos en que los distintos procesos obtienen la misma información o ignoran la obtenida previamente por otros procesos.

Un objetivo fundamental es el de intentar obtener un meta-algoritmo que permita aplicar el concepto de paralelismo con las mínimas restricciones posibles, permitiendo aprovechar al máximo las grandes posibilidades hardware de los últimos multiprocesadores del mercado. No se trata de imponer procesos secuenciales, pero el paralelismo introducido ha de ser en cierto sentido eficiente, porque en caso contrario, carece de toda lógica su introducción en el proceso.

Las dificultades que se presentan para obtener paralelismo desde las diferentes implementaciones tienen relación con:

- La gran variedad de arquitecturas existentes.
- Coste de reparto/coordinación (scattering/gathering) de los datos a manejar para obtener el resultado deseado.

- Coste de las comunicaciones entre procesos, para los que no hay una medida satisfactoria. Las comparaciones dejan de ser las operaciones críticas: lo realmente importante es el número de operaciones internas de un procesador junto con el coste de las comunicaciones entre ellos, lo que afecta a varios (o todos) los procesadores.

Se va un paso más allá, y una vez construidos los meta-algoritmos de ordenación (ascendente y descendente) aplicando abstracción no sólo a las estructuras de control sino también a las estructuras de datos, se obtienen los esquemas donde se encuentran contenidos ambos meta-algoritmos.

La particularización de los esquemas fijada la estructura de datos, deriva en los meta-algoritmos. Si a estos se les fija la estructura de control, se obtiene como resultado la colección de algoritmos clásicos de ordenación conocidos.

Tanto esquemas como meta-algoritmos permiten abordar el problema de ordenación desde una base más sólida y profunda, estableciendo el menor número de restricciones iniciales posible (condiciones mínimas) así como la mínima secuencialidad necesaria permitiendo, para cualquier conjunto de elementos, obtener una relación de orden total según se establezca en una función de comparación predeterminada.

Capítulo 2

Ordenación clásica: fundamentos básicos

The price of realibility is the pursuit of the utmost simplicity.
It is a price which the very rich find most hard to pay.

Sir Antony Hoare, 1980

2.1. Algoritmo: definición

Tal y como establece el diccionario de la *Real Academia de la Lengua Española*, las acepciones para la palabra algoritmo son:

- Conjunto ordenado y finito de operaciones que permite hallar la solución a un problema.
- Método y notación en las distintas formas de cálculo.

El origen etimológico de la palabra algoritmo se encuentra en el Árabe, más concretamente en el nombre del matemático Al-Khwarizmi. Una buena definición del concepto algoritmo aplicado al mundo de la informática, fue dada por Donald E. Knuth, en 1968: "Secuencia finita de instrucciones, reglas o pasos, que describen de forma precisa las operaciones en orden que debe llevar a cabo un ordenador para realizar la tarea en un tiempo finito".

De la anterior definición, se deducen las características necesarias que posee cualquier algoritmo:

1. Detallar sin ambigüedad cada paso necesario, indicando de forma clara, precisa y concisa la acción a realizar, sin dejar criterio alguno a una posible interpretación.

2. Completar su tarea en un número finito de pasos, lo que implica que su ejecución se realiza en un tiempo finito.

3. Disponer de un conjunto de datos a tratar en su entrada, estando permitido que dicho conjunto sea vacío.

4. Disponer de una salida que refleje el resultado al finalizar la ejecución, permitiendo que la salida se corresponda con el conjunto vacío.

5. Dos ejecuciones diferentes del algoritmo que dispongan del mismo conjunto de datos en su entrada, deben obtener idéntico resultado en su salida. Aquí se establecen dos visiones diferentes: una visión determinista, que acepta varios estados finales que sean igualmente válidos. Una visión no determinista, que aceptada salidas distintas que son igualmente válidas, en tanto en cuanto satisfacen la postcondición final.

La construcción de un algoritmo requiere una especificación de las estructuras de datos que utiliza y las operaciones permitidas sobre estas, junto a una estructura de control abstracta (no ligada al lenguaje de programación) que detalle únicamente el orden de precedencia de las acciones al realizar una ejecución.

2.1.0.1. Algoritmos de ordenación clásicos

En el mundo informático y matemático, un algoritmo de ordenación es definido como *aquel algoritmo cuya tarea asignada consiste en realizar la ordenación de un conjunto de elementos que le son facilitados en su entrada, devolviendo tras la ejecución, el mismo conjunto de elementos en su salida, satisfaciendo una relación de orden total predefinida.* Por tanto la salida de todo algoritmo de ordenación al finalizar su ejecución, se corresponde con el conjunto de elementos de su entrada permutados (o reordenados) satisfaciendo la relación de orden establecida.

Los algoritmos de ordenación clásicos tienen determinado su tiempo de ejecución no sólo por las circunstancias externas en las que se ejecutan, sino en gran medida por el volumen de elementos que procesan, el tiempo que emplean en acceder a las estructuras de datos elegidas y por las estructuras de control utilizadas según el lenguaje de programación seleccionado.

El resultado de las características precedentes descritas produce algoritmos incapaces de adaptarse al medio en que se ejecutan. Su implementación hace que sean inflexibles a cualquier tipo de cambio o adaptación a un posible paralelismo de procesos. En determinadas ocasiones unas circunstancias externas concretas podrían ser aprovechadas para obtener mejores tiempos de respuesta en su tarea. Sin embargo una definición y estructuras de datos y control muy rígidas, impiden aprovechar esas ventajas circunstanciales que ocasionalmente pudieran presentarse.

La implementación final de cualquier algoritmo de ordenación clásico es una forma cristalizada y cualquier eventual modificación tiene repercusiones difíciles de determinar y sobre todo de validar. La corrección de una modificación sobre la definición original del algoritmo no es deducible sobre la base de la corrección de la forma original lo que se traduce en *inflexibilidad*.

La clasificación de los algoritmos de ordenación en inserción, selección, intercambio, etc. se refiere únicamente a la implementación (representación final) que condiciona y está condicionada por la estructura de datos en que se implementa. Y la implementación final responde a condiciones y requisitos externos, mientras que la solución general básica del problema sólo debería atender a la naturaleza del propio problema.[1]

2.2. Un poco de historia

2.2.1. Una visión clásica

A lo largo de la historia de la informática y de forma tradicional, los algoritmos de ordenación se han clasificado atendiendo al lugar en que se encuentra almacenado el conjunto de elementos a ordenar.

Al realizar la clasificación siguiendo este criterio, se diferencian 2 grupos en los que incluir los algoritmos de ordenación clásicos:

- *Algoritmos de ordenación interna:* El proceso de ordenación se produce sobre un conjunto de elementos almacenado en la memoria interna de la máquina.
- *Algoritmos de ordenación externa:* El proceso de ordenación se realiza sobre un conjunto de elementos almacenado en dispositivos externos de almacenamiento masivo de información, cuyas características técnicas imponen restricciones a las operaciones del algoritmo.

[1] Principio fundamental de los meta-algoritmos definidos en los capítulos 4 y 5.

Ambas clasificaciones, lamentablemente arrastran vestigios del pasado, trasladándonos a los inicios del mundo de la informática donde la escasez de recursos disponibles y el elevado precio que se debía pagar por los mismos, imponían restricciones en el proceso de ordenación del conjunto de elementos: dichas restricciones, nada tienen que ver con la naturaleza del problema a resolver.

La escasez de memoria central unida a sistemas de almacenamiento externos secuenciales, propiciaba soluciones alejadas de la naturaleza del problema que se trataba de resolver.

La preocupación en primer lugar por conseguir algoritmos que no sobrepasasen las características técnicas de las máquinas de las primeras épocas de la historia de la informática, unido a principios como eficiencia y eficacia, encauzaron la forma de pensamiento y desarrollo hacia soluciones preocupadas por la gestión de recursos del sistema informático más que por el análisis minucioso del propio proceso de ordenación.

La ordenación, búsqueda y en menor medida la intercalación de información, son operaciones básicas en el campo de la documentación y en las que, según señalan las estadísticas, los sistemas de información emplean la mitad de su tiempo, consumiendo en mayor o menor medida la cantidad de recursos disponibles en el sistema.

Además de realizar búsquedas dentro de conjuntos de elementos, se necesita realizar una clasificación u ordenación de los mismos siguiendo algún orden particular: ejemplo de esto son las clasificaciones que se producen por un determinado número asociado a cada elemento, o las que devuelven en orden lexicográfico los elementos de un conjunto.

La clasificación es una operación tan frecuente en los sistemas de información actuales, que una gran cantidad de algoritmos se diseñan para clasificar conjuntos de elementos con la mayor eficacia, eficiencia y rapidez posibles.

La elección de un determinado tipo de algoritmo depende en gran medida del tamaño del conjunto de elementos que se deba clasificar, del lugar en que dicho conjunto de elementos se encuentra almacenado, de la forma de acceso a los mismos y de la necesidad de disponibilidad de resultados. Encontrar un equilibrio que satisfaga al productor de elementos y al consumidor de los mismos, debe ser una premisa básica y fundamental del algoritmo elegido.

2.2.2. Ordenación interna

Como se ha comentado, la forma tradicional de clasificar los diferentes tipos de algoritmos de ordenación se basa en el entorno físico en que se produce

la ordenación de elementos.

Aquellos algoritmos de ordenación que realizan su tarea sobre la memoria central del ordenador se denominan algoritmos de ordenación interna.

Por las características de almacenamiento y acceso de dicha memoria, el acceso al conjunto de elementos se realiza de forma aleatoria y directa, por lo que su gestión y tiempo de procesamiento es muy rápido.

Cualquier algoritmo de ordenación a lo largo del proceso de ejecución, necesita en mayor o menor medida disponer del conjunto de elementos en memoria. La diferencia fundamental para realizar la clasificación entre ordenación interna y externa, radica en la posibilidad de mantener el conjunto completo de elementos en dicha memoria.

Tanto en la resolución de problemas simples como a la hora de transmitir los conceptos básicos de ordenación, se utilizan ejemplos con conjuntos de elementos con una cardinalidad reducida, por lo que en estos casos se puede establecer la condición de mantener el conjunto completo de elementos en memoria central. Lógicamente esto no es lo habitual, ni se encuentra entre los casos más comunes en que estos algoritmos desarrollan su tarea.

2.2.3. Ordenación externa

Una ordenación se clasificada como externa cuando el conjunto de elementos a tratar se encuentra almacenado en dispositivos de almacenamiento externos a la máquina en la que se va a realizar el proceso de ordenación.

Con el paso de los años estos dispositivos han evolucionado, mejorando en gran medida el tiempo de acceso a los elementos que se encuentran almacenados en ellos.

No obstante, el proceso de ordenación incrementa sus tiempos de ejecución con todas aquellas tareas relacionadas con los siguientes conceptos:

1.- Búsqueda del elemento a tratar. En cada ejecución del algoritmo, este debe seleccionar del conjunto de elemento de entrada aquel al que le corresponde ser el siguiente elemento a tratar.

2.- Transmisión del elemento seleccionado hasta la memoria central del ordenador.

3.- Realizar las comparaciones y permutaciones de elementos necesarias hasta colocar el elemento seleccionado en el lugar que le corresponde.

4.- En caso de ser necesario, transmitir la nueva información obtenida hasta el dispositivo de almacenamiento externo.

Como parece lógico, todas las fases anteriormente definidas introducen una serie de retardos en la ejecución del algoritmo de ordenación que, lejos de ser considerados poco significativos, producen como resultado (en la mayoría de ocasiones) que algoritmos realmente rápidos en "tiempo de ordenación" se conviertan en verdaderos lastres para el sistema informático que los utiliza.

Si se analizan aquellos sistemas a los que no les queda otra alternativa que llevar a cabo una ordenación externa sobre dispositivos de almacenamiento masivo muy lentos,[2] se puede afirmar que la mejor gestión posible consiste en asignarles la prioridad más alta dentro del sistema.

Trasladando esa afirmación a la carga de trabajo que soporta el procesador, para obtener el mejor rendimiento por parte de este en aquellas circunstancias en que sea necesaria la utilización de sistemas periféricos externos (proporcionalmente mucho más lentos que el procesador), la mejor solución es asignar a esos procesos máxima prioridad sobre el resto. Mediante esta estrategia el procesador atiende de forma inmediata al sistema externo (de velocidad muy lenta) en el momento que este lo solicite, enviando/recibiendo la información necesaria. De esta forma, el sistema externo se pone a trabajar dejando al procesador libre por tiempo máximo para atender a otras tareas.

2.3. Ordenación serie y paralelo

2.3.1. Introducción

El término ordenación en el mundo informático, se define como el proceso de colocar un conjunto de elementos según algún tipo de orden preestablecido. El proceso de ordenación sigue un criterio ascendente o descendente en la búsqueda del resultado deseado.

Los programas de ordenador tales como compiladores o editores, con frecuencia necesitan ordenar tablas y listas de símbolos almacenados en memoria, con el fin de incrementar la velocidad y simplificar los algoritmos utilizados para acceder a los datos (para realizar búsquedas o añadir nuevos elementos al conjunto a manipular).

Debido a la gran importancia de ambas prácticas (búsqueda e inserción) y a su enorme interés teórico, los algoritmos para organizar valores almacenados en memoria de acceso aleatorio (ordenación interna según se establece en las clasificaciones clásicas) son la fuente de una colección innumerable de algoritmos.

[2] Proporcionalmente a la velocidad de trabajo del resto del sistema informático.

En los inicios de la resolución de problemas utilizando sistemas informáticos, los campos de investigación y desarrollo se centraron en algoritmos de trabajo en serie, es decir, aquellos que sólo realizan una única secuencia de tareas en un orden predefinido y determinado y que, para que se inicie la ejecución de la tarea $n+1$ es necesario que haya finalizado la tarea precedente n. Los primeros algoritmos de ordenación, por tanto, no contemplaban la posibilidad de dividir la tarea a ejecutar fusionando los resultados independientes obtenidos.

Años más tarde se introduce la posibilidad (al principio sólo de forma teórica), de realizar los procesos de ordenación en paralelo. A día de hoy se dispone de hardware multinúcleo que posibilita la ejecución real de varias tareas de forma simultánea, lo que ha permitido desarrollar una gran área activa de investigación en torno a los multiprocesos para realizar tareas, basándose en la división de una tarea compleja en varias simples, ensamblando posteriormente los resultados obtenidos.

Los algoritmos más eficientes que realizan su trabajo en serie, son conocidos por la cualidad de poder ordenar n valores en una media del orden $O(n \log n)$ comparaciones. Este es el límite lógico para la resolución del problema que se aborda.

De forma particular, los algoritmos de ordenación son examinados minuciosamente evaluando su comportamiento respecto a la ocupación de memoria (cantidad de memoria adicional requerida para ser ejecutados, añadida esta a la memoria que utiliza la secuencia original para la ordenación), estabilidad (requisito por el cual los elementos iguales van a mantener su orden relativo original) y un especial cuidado con la distribución de valores iniciales (en particular la complejidad del mejor y el peor de los casos posibles para cada algoritmo).

El conjunto de elementos a ordenar dispone de $n!$ disposiciones iniciales diferentes para los n elementos a tratar. Para realizar cualquier ordenación de dicho conjunto de elementos, cada pregunta (comparación) responde a la cuestión de, *los elementos comparados, ¿se encuentran en el orden apropiado?*.

Las "mejores" preguntas posibles para ordenar un conjunto de elementos del que no se dispone, en principio, de información alguna, son aquellas que consiguen que cada respuesta *divida por 2* el número de posibles configuraciones.

En la actualidad, la posibilidad de ejecutar procesos en paralelo añade una nueva dimensión a la resolución de problemas de todo tipo, algo que se particulariza en la resolución de problemas de ordenación. El inconveniente

que se encuentra al tratar de utilizar paralelismo para la resolución de tareas de ordenación, es que la mayoría de modelos en paralelo han sido concebidos con una idea propia que proporciona soluciones muy particulares a:

- Situaciones de contingencia.
- Localizaciones en memoria de los conjuntos de elementos a ordenar.
- Los caminos para el acceso a la memoria de los múltiples procesadores.

Para enfrentarse de forma clara y concisa a la resolución de un problema de ordenación utilizando técnicas de ordenación en paralelo, en primera instancia se necesita una completa definición del problema concreto, lo que se entiende por una secuencia ordenada de procesos en paralelo. Cuando los procesadores comparten una memoria común, la idea de las posibles contingencias a la hora de realizar los accesos a memoria por parte de un procesador en paralelo, es idéntica a la que se tiene para un procesador serie. De la misma forma que cuando se dispone de un único procesador, la complejidad de un algoritmo de ordenación en paralelo puede ser expresada en términos relativos al número de comparaciones y de movimientos internos de memoria.

Por otro lado, cuando los procesadores no comparten memoria y estos se comunican a través de redes de ordenadores (o de algún tipo de circuitería hardware específica para un problema concreto), la definición de los problemas de ordenación requiere de una convención para ordenar los resultados obtenidos de forma aislada en cada procesador, fusionando los resultados hasta obtener la solución completa al problema planteado.

Cuando se utilizan varios procesadores trabajando en paralelo sobre el mismo conjunto de elementos, la complejidad del algoritmo de ordenación se expresa mediante:

- Comparaciones en paralelo.
- Intercambios de información entre procesadores contiguos.
- Tiempos de retardo en la trasferencia de información entre procesadores.

La resolución a un problema de ordenación de elementos en paralelo, se suele expresar como *"la ordenación de un conjunto de n elementos utilizando m procesadores"*. Para resolver el problema los m procesadores han de compartir memoria. Por tanto, el acceso a esta debe realizarse teniendo en cuenta varios grados de contención (por ejemplo realizando lecturas y escrituras en paralelo de forma controlada).

Si nos situamos dentro del contexto de *procesamiento de la información*, se debe recalcar que los procesos de ordenación de un conjunto de elementos son una tarea básica y fundamental. El acceso a los miles de registros de información (datos almacenados para tareas de gestión, por ejemplo), sólo puede ser abordado mediante un procesamiento previo que mantiene bajo algún orden determinado la información guardada.

Los registros se almacenan ordenados respecto al valor de una clave, la cual puede ser un campo simple o estar formada por varios campos de datos. Los ficheros habitualmente se encuentran organizados en función de la salida que se espera van a producir con más asiduidad a los usuarios o a otros procesos externos.

Aceptando como un problema de base la limitación de memoria de los sistemas informáticos, y teniendo en cuenta la ingente cantidad de datos que se ha de almacenar en el día a día, abordar un problema de ordenación de elementos se ha convertido en una tarea imposible de realizar si se pretende mantener en memoria el conjunto completo de elementos a tratar.

La limitación de memoria impide la ordenación de ficheros directamente sobre esta. En los sistemas informáticos actuales se produce una obvia y real necesidad de mantener ordenada, de alguna forma, la información almacenada en ficheros. La tarea cada vez más compleja por el notable crecimiento de la cantidad de información almacenada, junto a la necesidad cada vez más creciente de inmediatez en los tiempos de respuesta, han demandado el estudio de nuevos campos de investigación donde se desarrollen técnicas que reduzcan los tiempos de ejecución.

La respuesta al almacenamiento masivo de información son las bases de datos. Dentro de estas, la forma de mantener ordenada la información manejando la mínima cantidad de datos necesarios, tiene su respuesta en la utilización de índices.

Un índice de una base de datos es una estructura que permite mejorar la velocidad de las operaciones mediante un identificador único para cada fila de cada tabla, obteniendo un rápido acceso a los registros que conforman la base de datos. Los tiempos de respuesta en el acceso a los registros mejoran cuando los índices se definen sobre los campos más utilizados en las búsquedas más frecuentes.[3]

[3]Para lo cual es necesario un conocimiento del dominio externo de la aplicación.

2.4. Algoritmos de ordenación serie en paralelo

En la actualidad, cuando se pretende afrontar la resolución de grandes problemas, el concepto de "procesamiento en paralelo" suele venir aparejado. La idea de "paralelismo" real se basa en la ejecución de tareas diferentes en la misma unidad de tiempo. El beneficio obtenido con la correcta aplicación de este concepto se ve reflejado en una disminución del tiempo de ejecución de la tarea global, posibilitando abordar problemas de otra forma irresolubles.

Algunas medidas de programación en paralelo (más concretamente las que se utilizan en las *redes de ordenación* [Sorting networks]), se basan en realizar la comparación de diferentes claves en la misma unidad de tiempo (en la misma unidad de tiempo, en diferentes procesadores, se comparan diferentes pares de claves de elementos).

Expresando el concepto de una manera más formal, se describe como la comparación de n claves con un número idéntico y diferente de estas en una unidad de tiempo t. Tras dicha comparación, se obtienen las n claves que han "ganado" sus respectivas comparaciones en la citada unidad de tiempo t.

Otra posible alternativa para introducir el concepto de paralelismo en un problema de ordenación (en el fondo idéntica en cuanto a tiempos de ejecución), se produce al realizar la comparación de una única clave con n claves diferentes en idéntica unidad de tiempo t.

El rendimiento de un algoritmo de ordenación en paralelo, se define como *la proporción entre el número de comparaciones/movimientos que se requieren por parte de un algoritmo de ordenación serie óptimo, y el número de comparaciones/intercambios que necesita el algoritmo homónimo que presenta ejecuciones paralelas.*

Tal y como se comentó en la introducción (apartado 2.3.1), el valor más lógico obtenido por un algoritmo de ordenación serie que trabaje con n elementos requiere del orden de $O(n \log n)$ comparaciones para disponer del conjunto de los n elementos ordenados. Si se dispusiera de n procesadores en los que realizar las comparaciones de forma simultánea, se podría reducir el tiempo de ejecución hasta un valor teórico de orden $O(\log n)$, de forma que se realizan n comparaciones en cada unidad de tiempo.

Este mínimo orden teórico, lamentablemente ha de ser incrementado en su tiempo de ejecución con los tiempos necesarios para la transmisión e intercambio de los elementos entre los diferentes procesadores. En muchos casos, el coste es tan elevado que no compensa la construcción e inversión en toda la arquitectura, salvo determinadas circunstancias críticas muy precisas (por ejemplo tareas prioritarias en sistemas de tiempo real[5]).

Lamentablemente una propiedad que caracteriza a las redes de ordenación es la inadaptabilidad a los cambios. Un algoritmo de ordenación en red puede ser implementado de forma conveniente en una máquina *SIMD* (single instruction stream, multiple data stream ⇒ una única instrucción, múltiples datos en paralelo).

En computación, *SIMD* es una técnica empleada para conseguir paralelismo a nivel de datos. Una máquina *SIMD* es un sistema que consiste en una unidad de control y un conjunto de procesadores con memoria local interconectados por una red de cableado. Los procesadores tienen un alto grado de sincronización. La unidad de control comparte las instrucciones que todos los procesadores activos ejecutan de forma simultánea (una máscara especifica un subconjunto de procesadores que están en modo "espera" durante un ciclo de instrucción). Los repertorios *SIMD* consisten en instrucciones que aplican una misma operación sobre un conjunto más o menos grande de datos. Nos encontramos ante una organización en la cual una única unidad de control común distribuye las instrucciones a diferentes unidades de procesamiento[6]. Todas ellas reciben la misma instrucción pero operan sobre diferentes conjuntos de datos, por lo que de forma síncrona, la misma instrucción se ejecuta sobre conjuntos de elementos diferentes.

2.4.1. Procesamiento paralelo del algoritmo serie *Burbuja*

El problema que se pretende abordar es, mediante la utilización del algoritmo *Burbuja* en su variante de procesamiento serie, ordenar un conjunto de n elementos de los que inicialmente no se dispone de información alguna. Para realizar la tarea se dispone de un sistema informático que consta de m procesadores, los cuales pueden trabajar de forma simultánea.

El principal problema que plantea la construcción de un hardware específico para ejecutar algoritmos de ordenación en paralelo[7], radica en que dicho hardware no es adaptable a lo largo del proceso de ejecución, lo cual impide maximizar el rendimiento de la inversión realizada.

Una ejecución con paralelismo para el algoritmo *Burbuja*, es la que compara cada elemento con su adyacente de forma simultánea. Para n elementos del conjunto, se requieren $\frac{n}{2}$ procesadores comparando primero las posiciones impares con sus adyacentes y, en el siguiente ciclo de reloj las posiciones pares.

El principal inconveniente se encuentra en el aprovechamiento (o mejor dicho, desaprovechamiento) de los recursos hardware a lo largo del proceso completo de ordenación. En las primeras fases de la ejecución los recursos hardware son aprovechados al máximo, trabajando de forma simultánea los m procesadores. Sin embargo, a medida que el proceso de ordenación avanza, los

elementos van ocupando sus posiciones definitivas en la relación de orden, por lo que cada vez el número de elementos a tratar es menor.

Como consecuencia de esto y en oposición al aprovechamiento máximo de recursos de las fases iniciales, se presenta una situación en la que muchos procesadores se encuentran ociosos en las fases finales. La elevada inversión realizada en hardware específico, lamentablemente sólo es aprovechada durante las primeras fases de la ejecución. Cuanto más se acerque el proceso a un estado final más hardware se desperdicia: esta redundancia de procesadores se considera inútil, ya que ejecutar un proceso que es inútil y se sabe de antemano que lo es, sólo produce un desperdicio de recursos tal y como sucede en esta ocasión.

Analizando la situación con más detalle, cada vez que finalicen 2 pasadas completas del algoritmo *Burbuja*, se necesita un procesador menos: en uno de los extremos se encuentran los 2 valores (máximos o mínimos) del conjunto de elementos tratado.

Supongamos un tamaño del conjunto de elementos a ordenar igual a n. Los m procesadores disponibles son aprovechados siempre que $n \geq 2 \times m$. A medida que se avanza hacia un estado final, en cada "pasada" del algoritmo se dispone de 2 elementos ocupando sus posiciones definitivas, lo que se traduce en que en la siguiente "pasada" el conjunto de elementos a tratar reduce su número en 2 unidades. Siempre que la cantidad de elementos que reste por tratar sea igual o superior a $2 \times m$, se están aprovechando los m procesadores. Por contra, todas aquellas fases de la ejecución en las que el número de elementos por tratar sea menor de $2 \times m$, está desperdiciando tiempo de procesador o, lo que es lo mismo, la inversión económica realizada en hardware está siendo desaprovechada.

Realmente en la última "pasada" del algoritmo sólo se requiere el uso de un procesador. En este instante de la ejecución se están desaprovechando $\frac{n}{2} - 1$ procesadores. Este es el principal inconveniente en el diseño de arquitecturas hardware específicas para abordar un problema de ordenación concreto: es muy complicado, y en muchas ocasiones como la descrita, resulta imposible mantener un nivel de ocupación elevado para todos los procesadores, estando incluso desocupados en varias fases del proceso.

En resumen, la paralelización mediante hardware para el algoritmo *Burbuja*, sólo resulta interesante en las primeras fases de la ejecución que son las únicas que maximizan la utilización de recursos y con ello la inversión realizada.

Además, a la presumible mejora de los tiempos de ejecución conseguida

por la aplicación de paralelismo, ha de añadirse el incremento de tiempo producido por la fusión de los resultados obtenidos en cada procesador, así como el tiempo que se dedica a evitar posibles interbloqueos producidos por accesos simultáneos a memoria.

2.4.2. Procesamiento paralelo del algoritmo serie *Fusión*

Otro de los algoritmos clásicos de ordenación que se adapta de forma sencilla a la paralelización por hardware es el algoritmo *Fusión*. El planteamiento es el siguiente: antes de realizar la fusión de los subconjuntos ordenados el proceso de ordenación de cada subconjunto de elementos se realiza en paralelo.

Desgraciadamente, de la misma forma que ocurre con el algoritmo adaptado *Burbuja*, a medida que se avanza en el proceso de ordenación de elementos cada vez son necesarios menos procesadores, desaprovechando la inversión realizada.

Este caso tiene una explicación igual de sencilla a la de *Burbuja*: en las primeras fusiones de elementos los m procesadores son aprovechados, pero en la última fusión sólo trabajan 2 procesadores en paralelo y sus resultados son fusionados por un tercer procesador.

El resultado obtenido, al igual que en *Burbuja*, es que la inversión hardware sólo se rentabiliza al máximo en las primeras fases del proceso. En cuanto a la mejora de los tiempos de ejecución no se consideran un éxito al estar condicionados por el coste de la inversión realizada. Las últimas fases en las que se aproxima a la solución, el proceso de ejecución paralelo se asemeja cada vez más al que realiza su homólogo en serie. A esto han de añadirse los siguientes tiempos extra requeridos por la arquitectura hardware que posibilita el paralelismo:

- Tiempos de comunicación entre procesadores.
- Tiempos de espera del procesador al no encontrarse disponibles en la entrada los elementos a tratar.
- Colisiones por accesos simultáneos a memoria de los diferentes procesadores.

Los tiempos de transmisión de elementos entre procesadores[4] junto con las colisiones producidas por accesos simultáneos a memoria por parte de estos, son inevitables. Sin embargo, a modo de aperitivo sobre el núcleo central

[4]Primero con los elementos a tratar y posteriormente con los resultados obtenidos por cada procesador.

de la presente *Tesis*, es posible afirmar que los tiempos muertos de procesador que se derivan de no disponer en la entrada de elementos a tratar pueden ser eliminados[5] permitiendo entremezclar fusiones a varios niveles.

Si se centra el estudio en la paralelización del algoritmo *Fusión*, se puede proponer una solución alternativa que no desaprovecha tanto los recursos hardware: para ello se realiza la ordenación de elementos en una matriz de $filas \times columnas$.

En cada ordenación y para que no accedan de forma simultánea los procesos de fila y columna (se procesan simultáneamente $n\,filas + m\,columnas$), sólo se permite actuar sobre un grupo de elementos. Siguiendo esta premisa se ejecutan por ejemplo, todos los procesos sobre las filas de la matriz y, a continuación, aquellos procesos que realizan su tarea sobre las columnas. Al finalizar la ejecución se obtiene como resultado una matriz cuyos elementos tienen una disposición de orden total.

Si se parte del supuesto de disponer de una matriz de tamaño 4×4, se requieren al menos $4(n-1)$ movimientos para ordenar cada array particular.

Una posible mejora es la que construye un diseño en que los procesadores comparten una memoria común. Se ha demostrado que es preferible que los m procesadores accedan a una memoria común (controlando las posibles colisiones por accesos a memoria), frente al diseño que propone una memoria propia para cada uno de ellos.

Otra posible mejora consiste en que en cada unidad de tiempo cada elemento sea comparado con los $(n-1)$ restantes, mediante el diseño de una máquina con n líneas y n procesadores cuya salida sólo es *0 o 1* en función del resultado obtenido al realizar la comparación con el elemento introducido.

Con el diseño descrito, en una unidad de tiempo se conoce a partir de un elemento x introducido la posición que ocupa este frente a los $(n-1)$ elementos restantes. Además se clasifica a sus "competidores" en 2 subconjuntos: los elementos mayores y los menores que x. Por otro lado se permite pasar al siguiente grupo de comparaciones obviando esa información, y comparar el segundo elemento con los $(n-1)$ valores de forma similar. Al finalizar el proceso según los resultados obtenidos de las n comparaciones en n pasos, se conoce la posición definitiva que ocupa cada elemento (dentro del subconjunto formado por los n elementos).

Realizado el proceso según la descripción anterior no se desaprovechan procesadores, pero se desaprovecha información (se repiten comparaciones con

[5] Como se demuestra en la construcción del meta-algoritmo ascendente.

idénticos elementos). Al finalizar la primera pasada se obtiene la posición del primer elemento seleccionado respecto a los $(n-1)$ restantes. Siguiendo el proceso, el segundo elemento seleccionado sólo es necesario que sea comparado con los $(n-2)$ elementos restantes, el tercero con $(n-3)$ valores, y así sucesivamente. De esta forma en cada paso se obtiene la posición de un elemento concreto, por lo que se reduce el tiempo de ejecución final pero se desperdicia hardware porque, cuanto más se acerca el proceso a un estado final, menos procesadores son necesarios.

2.5. Algoritmos de ordenación en bloque

2.5.1. Una visión inicial

Para ordenar un conjunto formado por n elementos, se considera la posibilidad (que es a la vez imposición) de disponer físicamente de n procesadores. Esto es algo que desde el punto de vista económico se antoja como imposible de realizar, por el excesivo coste que conlleva. Por ello una mejora sustancial y más económica que se propone es la siguiente: si se desean ordenar n elementos, y para ello se dispone de p procesadores, lo que se hace es dividir el conjunto de elementos a ordenar en M bloques de forma que:

$$M = n/p$$

Así definido, cada bloque se encarga de realizar de forma independiente la ordenación de n/p elementos. Al finalizar todos y cada uno de los procesos, se obtienen sus correspondientes salidas $s_1, s_2, ..., s_p$, las cuales disponen en el orden adecuado los elementos que contienen. En última instancia deben ser fusionadas las salidas previas ya ordenadas obteniendo con ello el resultado definitivo.

Detallando un poco más el proceso anterior, cada procesador se encarga de ordenar M elementos con el algoritmo que se prefiera (cualquiera de los algoritmos serie es válido). Supongamos la utilización del algoritmo serie *Fusión*. En una segunda fase los M elementos ordenados por cada procesador P, son fusionados con los M' elementos de otro procesador diferente P'. Obtener la fusión de las salidas parciales s_1 y s_2 es una tarea relativamente sencilla. Los elementos de las salidas s_1 y s_2 están ordenados, por lo que la acción a realizar es tan simple como ir seleccionando el menor elemento de cada salida parcial, obteniendo una nueva salida s_3 que contiene en orden todos los anteriores elementos.

FIGURA 2.1: Ordenación en bloque: visión inicial

En este proceso se fusionan las salidas $s_1, s_2, ..., s_p$ combinándolas de 2 en 2 hasta obtener como único resultado una salida con todos los elementos ordenados, de tamaño n (la mezcla de 2 en 2 elementos se puede realizar con el algoritmo par-impar). Dado que pueden mezclarse los resultados de cualquier procesador, en un ciclo de tiempo se fusionan los procesadores pares y en el siguiente ciclo los impares hasta que se obtenga el resultado definitivo.

Otra posible alternativa para ordenar los M elementos es la siguiente: el procesador p_1 dispone su salida en orden ascendente y, el procesador p_2 dispone la suya en orden descendente. Basta ir comparando los pares de elementos que ocupen idénticas posiciones en los diferentes procesadores, para obtener como resultado en un procesador los elementos mayores y en el otro los menores.

FIGURA 2.2: Ordenación en bloque: comparando elementos adyacentes

Cuidado: Las salidas no están ordenadas, pero en la salida superior se encuentran todos los elementos menores y en su homóloga inferior, todos los elementos mayores.

2.5.2. Alternativa

En este punto se describe una posible alternativa para el algoritmo de bloque en paralelo. Se toma como punto de partida la premisa de disponer de 4 procesadores (p_0, p_1, p_2, y p_3), actuando todos ellos sobre la misma memoria compartida. El tamaño de la citada memoria debe ser al menos $4 \times M$, siendo M el valor del bloque tal y como se describe en el anterior apartado ($M = n/p$).

La situación propuesta es la siguiente: en el ciclo t son comparados los elementos que se encuentran accesibles por los procesadores pares con sus

procesadores adyacentes y, en el ciclo $t+1$, los elementos accesibles por los procesadores impares con los de sus adyacentes (cada uno con el suyo). Realizando las tareas en este orden, en el ciclo t se comparan los elementos de los procesadores p_0 y p_1 simultáneamente a los elementos situados en los procesadores p_2 y p_3. En el siguiente ciclo $t+1$ son comparados los valores de los procesadores p_1 y p_2.

Tras cada comparación de elementos se ha de realizar un reordenamiento "local" de estos en cada procesador. Se debe conservar la premisa de que si en el procesador z los elementos son ordenados mediante un criterio "descendente", su procesador adyacente $z+1$ debe disponer de los elementos en su salida utilizando un criterio de ordenación "ascendente".

En el ejemplo que se muestra a continuación los procesadores p_0 y p_2 utilizan una organización de sus elementos siguiendo un criterio descendente, y los procesadores p_1 y p_3 siguen un criterio ascendente.

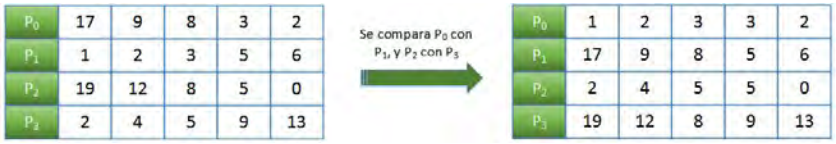


FIGURA 2.3: Primera comparación procesadores pares con adyacentes

Cada procesador debe reordenar los elementos de que dispone siguiendo el criterio inicial que le haya sido asignado. Finalizada la ordenación se vuelven a comparar elementos, en este caso entre los procesadores impares y sus adyacentes (procesador p_1 con p_2).

FIGURA 2.4: Primera comparación procesadores impares con adyacentes

La siguiente acción a realizar es nuevamente la de reordenar los elementos de los procesadores p_1 y p_2, para que en la fase siguiente se comparen los elementos de los procesadores pares (p_0 y p_1 por un lado y los elementos de los

procesadores p_2 y p_3 por otro lado). El proceso es iterado tantas veces como sea necesario hasta que no se produzca ningún intercambio de elementos.

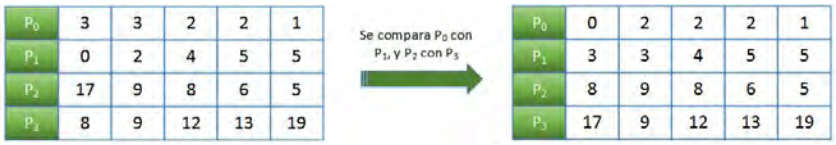


FIGURA 2.5: Segunda comparación procesadores pares con adyacentes

Se reordenan valores de los procesadores y se itera el proceso:

FIGURA 2.6: Segunda comparación procesadores impares con adyacentes

En la última comparación no se produce ningún intercambio de elementos entre procesadores. Esto indica que el proceso ha finalizado: los elementos de los que se dispone en cada procesador se encuentran ordenados.

Lógicamente el orden en que se encuentran los elementos de cada procesador responde a la premisa inicial de ordenamiento que le fuese asignada. Por este motivo, para obtener el conjunto completo y ordenado que se busca, los elementos situados en cada procesador han de ser recorridos según el criterio que le fuese asignado inicialmente.

2.5.3. Reflexiones sobre la parte hardware introducida en el algoritmo

Los 4 procesadores disponibles son aprovechados al máximo durante la ejecución completa del algoritmo. Se debe disponer de una unidad de control central, que es la encargada de indicar a todos ellos de forma simultánea que han de realizar la función que tienen encomendada. El trabajo a desempeñar por cada uno se realiza sobre una memoria compartida, cuyo tamaño mínimo disponible para una correcta ejecución debe ser al menos de $4 \times M$.

Principalmente este es uno de los mayores inconvenientes que plantea el desarrollo de esta arquitectura de procesadores. Disponer de un elevado número de procesadores implica que el valor de memoria M demandado por

la arquitectura es muy elevado, por lo que se plantea un problema de inexistencia de hardware al no disponer el mercado de memorias con una capacidad tan elevada.

Una primera aproximación para subsanar este problema, consiste en tener conectados los procesadores adyacentes mediante una línea de datos bidireccional. De esta forma un procesador p envía el elemento a tratar a otro procesador *p'*, que se encarga de establecer la relación de orden entre ambos elementos. Este devuelve su salida con la relación de orden establecida al procesador p inicial.

El principal problema de esta solución radica en la obligatoriedad de intercambiar información entre los diferentes procesadores, lo que conlleva necesariamente un incremento en el tiempo de ejecución.

La ejecución del algoritmo ve incrementados sus tiempos de respuesta por el tiempo destinado al intercambio de información entre procesadores. En cada unidad de tiempo uno de los procesadores es el encargado de comparar 2 elementos del conjunto y, el otro procesador en lugar de encontrarse ocioso a la espera de la respuesta, se encarga de comparar los siguientes pares de elementos.

La ventaja obtenida ejecutando la solución de esta manera está directamente relacionada con el tamaño de la memoria: cada procesador sólo necesita disponer de un tamaño de memoria mínimo de *M+1.* Lógicamente la introducción de esta mejora en lo que a memoria se refiere, acarrea el inconveniente de incrementar de forma notable el tiempo de ejecución, que es desperdiciado en los intercambios de información entre procesadores.

Sin perder de vista todo lo anterior, es posible añadir una última mejora que evita una penalización excesiva en el tiempo de ejecución: se trata de añadir una línea más de comunicación hardware entre procesadores aleatorios. De esta forma, cada procesador se encuentra conectado con un procesador "vecino" y además algunos de ellos se encuentran conectados entre sí de forma aleatoria. Esto permite que la información fluya no sólo entre procesadores vecinos (adyacentes), sino que también se realice un intercambio de información entre procesadores elegidos al azar, lo que acelera el proceso de ordenación.

En determinadas situaciones esta alternativa aporta una mejora al tiempo de ejecución, pero no es siempre así. Es muy probable que en determinadas ocasiones bajo unas circunstancias de ordenación concretas, esta alternativa incremente el tiempo de ejecución. Por otro lado, está demostrado de forma estadística que el intercambio de información aleatorio entre procesadores permite mejorar los tiempos medios de la ejecución completa, obteniendo mejores resultados globales, eso sí, a base de un ligero incremento en el coste hardware

que según el proyecto a desarrollar, puede ser o no asumible.

2.5.4. Reflexiones sobre las mejoras introducidas en el software del algoritmo

El hecho de disponer de los elementos ordenados "ascendentemente" en un procesador y "descendentemente" en otro procesador adyacente, permite realizar comparaciones de forma más rápida. Sin embargo, no se debe olvidar que para introducir esta pequeña mejora en la velocidad de procesamiento se ha de cumplir obligatoriamente una premisa que no siempre es fácil: el número de elementos a ordenar en cada uno de los procesadores debe ser *exactamente el mismo.*

Los meta-algoritmos de ordenación que serán descritos en los próximos capítulos, tratan siempre de evitar en la medida de lo posible, cualquier premisa o precondición ajena al propio proceso de ordenación impuesta con el fin (que no siempre es posible conseguir) de mejorar la velocidad de ejecución (sobre todo las que vienen impuestas por los lenguajes de programación junto a estructuras de datos determinadas).

No son pocas las ocasiones en las que cumplir con esos "pequeños" detalles impuestos como premisa, totalmente ajenos a la naturaleza del problema, resulta prácticamente imposible de realizar. Y no son menos aquellas ocasiones en las que adaptar una solución con el único fin de satisfacer unas condiciones iniciales, lleva aparejado unos costes de mantenimiento de estructuras tan elevados que no compensa su utilización.

Capítulo 3

Meta-algoritmos de ordenación: pilares fundamentales

Maestro, no tengo nada en mi mente, ¿qué debo hacer?.
Tíralo fuera.
Pero si ya no tengo nada.
Entonces, quédatelo.

Filosofía Zen

3.1. Introducción

¿Qué se entiende por meta-algoritmo de ordenación?.

La primera parte de la expresión, *meta-algoritmo*, parece dejar claro en la propia definición que se trata de un concepto que engloba a otros algoritmos, situándose un paso por encima de estos. Desde el punto de vista conceptual los algoritmos contenidos son particularizaciones del meta-algoritmo. Este posee unas condiciones mínimas pero completas que establecen los mínimos requisitos para resolver un problema. Esta nomenclatura (basada en la aplicación del concepto *abstracción*) trata de sugerir que un meta-algoritmo[1] pretende realizar idéntica tarea que un conjunto de algoritmos, deshaciéndose de todo aquello que sea considerado superfluo centrándose exclusivamente en la raíz del problema a resolver.

[1] **meta-:** Prefijo utilizado para indicar que un concepto es una abstracción tomando otro concepto como base.

La segunda parte de la expresión, *ordenación*, indica el objetivo para el cual se desarrolla. Se trata de resolver un problema cuyo fundamento es la *ordenación de elementos,* o lo que es lo mismo, se trata de obtener una relación de orden total para un conjunto de elementos dado de cualquiera naturaleza. Aunque parezca una obviedad el problema de ordenación de elementos es un problema con concepto global: hasta que no han sido tratados todos los elementos del conjunto no es posible establecer cuál de ellos es el menor o el mayor.

Fundiendo ambos conceptos, un meta-algoritmo de ordenación se define con el objetivo de resolver el problema de obtener ordenado un conjunto de elementos de cualquier naturaleza, de una manera abstracta, estableciendo el mínimo número de restricciones posible y alejándose por completo en su definición de los detalles de implementación. Con esta definición es posible afirmar que el meta-algoritmo engloba a varios algoritmos de ordenación clásicos.

Para obtener el detalle de los meta-algoritmos definidos el estudio se basa por un lado, en los algoritmos clásicos *Quicksort* y *Binario*, y por otro lado en *Fusión* y *Torneo*. Al analizar en profundidad el comportamiento de estos algoritmos (y de otros muchos que se engloban en su misma naturaleza por presentar similar comportamiento), se detalla una generalización que incluye a todos aplicando dos estrategias fundamentales de resolución de problemas: los primeros utilizan una estrategia de resolución descendente por división de tareas mediante refinamientos sucesivos, llevando a cabo la búsqueda de la solución mediante la división del problema en subproblemas, lo que se conoce como estrategia top-down. El segundo grupo de algoritmos, por contra (o más bien de forma dual), resuelven idénticos problemas con una estrategia de fusión de soluciones parciales desde abajo hacia arriba, lo que tradicionalmente se conoce como estrategia bottom-up.

En síntesis se puede afirmar que cualquier algoritmo de ordenación clásico es eficaz, puesto que planteado el problema lo resuelve en un tiempo finito: otra cosa muy distinta es la eficiencia en el proceso. Tomando como base para el análisis y comparación de cada algoritmo el meta-algoritmo, se ven claramente aquellos procesos a los que cada algoritmo asigna mayor prioridad, dejando a las claras las carencias y virtudes en el proceso de ordenación de cada uno de ellos. Esto permite aportar soluciones parciales a las carencias detectadas e intentar trasladar las virtudes de cada algoritmo, en la medida de lo posible, a otros cuyas particularizaciones específicas así lo permitan.

Cuando se utiliza una versión final de cualquiera de los algoritmos clásicos conocidos para resolver un problema de ordenación, se arrastran todas

las decisiones que se tomaron en su implementación (particularizaciones), debiendo adoptar tanto las estructuras de control como las estructuras de datos estrictamente definidas. En muchas ocasiones esas particularizaciones no se adaptan al problema que se pretende resolver, y resultaría deseable disponer de una flexibilidad del algoritmo que permita a este adaptarse a la naturaleza del problema.

La solución local sería retroceder en la concreción del algoritmo hasta un punto en que recubra el problema y sus especificaciones y llevar a cabo un refinamiento desde ese punto.

Un algoritmo abstracto y sus primeros refinamientos, recubren todas las posibles situaciones: sólo se ha de escoger el refinamiento más apropiado y terminar su particularización para el caso concreto en que se tienen en cuenta las especificaciones del problema.

3.2. Abstracción

3.2.1. El papel de la abstracción

A lo largo de la historia de la informática, analistas, diseñadores y constructores de aplicaciones software, se han enfrentado en no pocas ocasiones a la tarea de resolver problemas cuya complejidad podía ser considerada como "muy elevada". En todos los casos se debía encontrar soluciones con las herramientas disponibles en cada época. En multitud de ocasiones la solución propuesta al problema se debía adaptar, obligatoriamente, a las restricciones hardware de la máquina en que iba a ser ejecutada, a los lenguajes de programación disponibles en cada época (habitualmente a los que están más "de moda") y a sus estructuras de datos predeterminadas e inflexibles.

Afortunadamente el paso del tiempo y sobre todo la experiencia, han ocasionado un cambio en la mentalidad de los desarrolladores en el momento de representar las soluciones de la forma más independiente posible de aquellas restricciones que no tienen nada que ver con el problema que se aborda. Esto ha posibilitado la introducción del concepto *abstracción* en la ingeniería del software, con todas las ventajas que ello conlleva al detallar las soluciones.

Como bien describe Wulft: «*Los humanos hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad: abstraernos de ella. Incapaces de dominar en su totalidad los objetos complejos, se ignoran los detalles no esenciales, tratando en su lugar con el modelo ideal del objeto y centrándonos, exclusivamente, en el estudio de sus aspectos esenciales*».

Se puede afirmar que en la resolución de problemas complejos por parte de la ingeniería del software, *la introducción del concepto abstracción[8] permite añadir la capacidad de encapsular y aislar la información del diseño y*

ejecución. En este sentido al realizar el desarrollo de nuevas aplicaciones, las técnicas orientadas a objetos[9] son vistas como un hito más en la historia de la ingeniería del software, culminando su definición y aplicación en los tipos abstractos de datos. Hoy día la utilización de *TAD* se ha convertido en un pilar fundamental para desarrollar las mejores soluciones software bajo cualquier circunstancia. Los *TAD* permiten la generación de código fuente basado en conceptos abstractos que, afortunadamente, nada tienen que ver con las restricciones impuestas por el lenguaje de programación utilizado o los equipos hardware sobre los que debe correr, preocupándose únicamente por la propia naturaleza del problema y su forma de resolución.

3.2.1.1. La abstracción como un proceso mental natural

Habitualmente el ser humano comprende el mundo mediante la construcción de modelos mentales formados por partes del mismo, tratando de aprender de aquellas cosas con las que interactúa; un modelo mental no es más que una vista simplificada que muestra cómo funciona el objeto, la manera en que se interactúa con él. Básicamente realizar un proceso que permite la construcción de modelos es lo mismo que llevar a cabo el diseño de software: a pesar de que el desarrollo de software es único, su diseño produce un modelo que permite ser manipulado por parte de un equipo informático.

Para que sean útiles y aportar sentido (cada uno dentro del contexto que proceda) los modelos mentales han de resultar más sencillos que el sistema que tratan de imitar o recrear. Si no se sigue dicho principio, el modelo generado con toda probabilidad, resulte inútil. A continuación se muestra un ejemplo sencillo: consideremos un mapa como el modelo de un territorio. Con el fin de que este sea útil debe representar de forma sencilla el territorio que modela. Un mapa bien abstraído y modelado sirve de gran ayuda. Transmite la información precisa y esencial de las características del territorio que modela: características que se antojan como básicas y fundamentales para realizar desplazamientos de forma eficiente por el terreno modelado.

Del mismo modo que un mapa ha de ser más pequeño (significativamente) que el territorio al que representa y sólo ha de incluir información que ha sido seleccionada minuciosamente, los modelos mentales abstraen las características de un sistema para que sea comprendido, ignorando aquellos detalles considerados como irrelevantes. Este proceso de *abstracción* es psicológicamente necesario y natural, siendo crucial para comprender el complejo mundo en que vivimos.

3.2.1.2. La abstracción en la ingeniería del software

La abstracción es por tanto un concepto fundamental que ayuda a resolver problemas complejos de forma sencilla, proporcionando soluciones claras y comprensibles. En la ingeniería del software dichas soluciones han de ser independientes y encontrarse alejadas de los entornos de construcción elegidos para realizar la codificación.

El concepto de abstracción es "adoptado" dentro de la ingeniería del software en el sentido más amplio de la palabra. Lamentablemente, a pesar de las grandes ventajas que aporta a la resolución de problemas informáticos no sólo en las fases de análisis y diseño, sino también en la fase de codificación del producto, no son pocos los profesionales que, con criterios inflexibles, anquilosados y anclados en una nostalgia por tiempos pretéritos, no se resignan a abandonar la exclusiva utilización de estructuras de datos y control proporcionadas por el lenguaje de programación de "moda" elegido como única vía posible para la resolución del problema.

La *abstracción* es un elemento fundamental e indispensable para enfrentarnos a la complejidad inherente del software. Permite representar las características esenciales de un objeto sin preocuparnos de los detalles superfluos del mismo (no esenciales).

Una buena abstracción se centra en la vista externa de un objeto, de manera que permite separar el comportamiento esencial de este de su implementación en un lenguaje de programación. Definir una abstracción, implica describir el funcionamiento esencial de una entidad del mundo real sin importar lo compleja que pueda llegar a ser.

Si se centra el objetivo en los tipos de datos proporcionados por los lenguajes de programación, el proceso de abstracción da como resultado nuevos tipos no predefinidos. Son representaciones a más alto nivel conceptual, siendo englobados dentro de en un nivel lógico alejado del hardware del sistema y denominados *tipos abstractos de datos (TAD).*

Los *TAD* son tipos de datos definidos por el diseñador, analista o programador, que se manipulan de un modo similar a los tipos de datos definidos por el sistema. De forma análoga a estos los *TAD* corresponden a un conjunto de valores (que puede ser de tamaño indefinido). Los usuarios crean variables con valores que se encuentren dentro del rango de valores legales operando con ellos mediante las operaciones establecidas.

Sirva como muestra el siguiente ejemplo: en el caso de que se deba programar una estructura de datos *pila,* esta se define mediante un *TAD* y las

operaciones que se realizan sobre ella, definen las únicas operaciones permitidas sobre las instancias de dicha estructura de datos. De esta forma se tienen controlados en todo momento los elementos que componen la pila, al permitirse únicamente sobre ella las operaciones predefinidas. Con estas condiciones se garantiza la integridad de la estructura de datos elegida.

3.2.1.3. Tipos de datos

Todos los lenguajes de programación soportan algún tipo de dato. En general los lenguajes de programación convencionales soportan, en mayor o menor medida y con mayor o menor acierto en su gestión, tipos de datos base como enteros, reales y caracteres, así como otros tipos compuestos por estos tales como *arrays* (vectores y matrices) y registros.

Un tipo de dato es un conjunto de valores y un conjunto de operaciones definidas por esos valores.

Un valor depende de su representación y de la interpretación de dicha representación, por lo que una definición informal de un tipo de dato podría ser: *Representación + Operaciones.* Un *tipo de dato* se describe como un conjunto de objetos que son representados de igual manera junto con las operaciones permitidas sobre dicho conjunto.

La mayoría de los lenguajes tratan las variables y constantes de un programa como *instancias* de un *tipo de dato.* Los tipos de datos proporcionan al compilador la información necesaria sobre la memoria que debe asignarse a cada instancia, la forma de interpretar los datos o las operaciones que están permitidas sobre ellos. A continuación se muestra un pequeño ejemplo: cuando se escribe una declaración tal como $float\ z$ en C o C++, se está declarando una instancia denominada z del tipo de dato $float$ indicándole al compilador que se reserven, por ejemplo 32 bits de memoria, así como las operaciones que están permitidas.

Sin embargo no es necesario escribir la declaración del tipo $float$. El autor del compilador ya lo ha hecho por nosotros. Los tipos de datos construidos en un compilador de este modo son conocidos como *tipos de datos fundamentales o predefinidos.*

3.2.1.4. Ventajas de los tipos abstractos de datos

Un *TAD* es un modelo (estructura) junto a las operaciones que afectan a dicho modelo. Es similar a la definición de objeto y de hecho estás íntimamente ligadas. Los *TAD* proporcionan numerosos beneficios al programador que se resumen en los siguientes:

1.- Permite una mejor conceptualización y modelado del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.

2.- Mejora la robustez del sistema. Si hay características subyacentes en los lenguajes que permiten la especificación del tipo de cada variable, los tipos abstractos de datos permiten la comprobación de tipos con el fin de evitar posibles errores en tiempo de ejecución.

3.- Mejoran el rendimiento (prestaciones). Para sistemas *tipeados*, el conocimiento de los objetos permite la optimización de tiempo de compilación.

4.- Separa la implementación de las especificaciones. Permite la modificación y mejora de la implementación sin afectar al interfaz público del *TAD*.

5.- Permite la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.

6.- Recoge mejor la semántica del tipo. Los *TAD* agrupan o localizan las operaciones y la representación de atributos.

3.3. Construcción de meta-algoritmos

3.3.1. La abstracción como principio fundamental

El uso sistemático de la abstracción permite reducir la complejidad de un problema, eliminando los detalles irrelevantes propios de una implementación concreta; detalles tanto de estructuras de datos como de estructuras de control que son necesarios en una versión final, pero que en un primer análisis imponen restricciones no esenciales aumentando la complejidad del mismo.

En una abstracción sólo se conservan, por un lado, aquellas condiciones que son necesarias para la corrección del proceso y que cualquier implementación ha de respetar. Y, por otro lado, aquellas operaciones que son necesarias en algún momento del proceso sin tener en cuenta que una implementación ofrece más operaciones de las estrictamente necesarias.

La idea de plasmar las soluciones a los problemas planteados tomando como base fundamental la abstracción no es nueva. Sin embargo su correcta descripción y aplicación no han sido siempre realizadas de la forma más acertada posible para el problema que se pretendía resolver.

Resulta curioso el interés desmesurado que desde no hace demasiado tiempo ha despertado el intento de que los nuevos aprendices del mundo de la

informática utilicen exclusivamente soluciones basadas en programación orientada a objetos, con independencia del problema que se aborde.

Es cierto que la utilización de programación orientada a objetos exige la puesta en práctica de, entre otros conceptos, el de abstracción. Sin embargo no es menos cierto que la utilización de abstracción independientemente del paradigma de programación a utilizar es, sin ningún género de duda, el concepto de mayor peso en la búsqueda de una solución fiable, clara y sencilla.

La utilización de metodologías basadas en la *orientación a objetos* introduce el concepto de abstracción en las estructuras de datos de la solución. Sin embargo se deja de lado a las estructuras de control, que pueden ser abstraídas de igual forma, obteniendo una solución aún más limpia y eficiente.

El uso de la abstracción mantiene la libertad en la toma de decisiones durante el mayor tiempo posible. Los desarrollos basados en este principio consiguen evitar un compromiso prematuro con aquellos detalles que en lugar de acercarnos a la solución, limitan y condicionan esta. La mayoría de los lenguajes de programación actuales proporcionan abstracción de datos, lo que permite el uso de otra serie de capacidades (muy deseables por otro lado) como son la encapsulación, herencia y polimorfismo[10] que añaden un potencial adicional.

La utilización correcta de la abstracción durante el proceso de análisis de un problema conlleva el manejo único de conceptos básicos y fundamentales para el dominio de la aplicación, dejando para fases posteriores (una vez analizado y comprendido el problema) decisiones de diseño e implementación.

Un uso adecuado de la abstracción permite utilizar el mismo modelo para el análisis, diseño de alto nivel, estructura del algoritmo, estructura de una base de datos con el conjunto de elementos a tratar o documentación del problema: un estilo de diseño independiente que pospone los detalles de programación hasta la fase final donde se encuentran los procesos más mecanizados de todo el sistema.

El desarrollo del núcleo fundamental de los meta-algoritmos y los esquemas que se describen en los próximos capítulos, basan su peso específico en el principio fundamental de abstracción como pilar básico y necesario para el análisis, diseño y resolución de las nuevas taxonomías definidas para la resolución de problemas de ordenación de elementos.

3.3.1.1. Fundamentos de la investigación

A partir del minucioso análisis del proceso que utiliza cada uno de los algoritmos clásicos de ordenación para realizar su tarea, aparecen una serie de similitudes que lejos de mostrar esas grandes diferencias que transmiten

sus respectivas implementaciones finales, muestran enormes similitudes en el proceso de ejecución.

La abstracción de los algoritmos de ordenación clásicos se realiza bajo dos aspectos diferentes. En primer lugar si se realiza una abstracción de la implementación de las estructuras de control, se obtienen los meta-algoritmos. Si además la implementación de las estructuras de datos es abstraída se obtienen los esquemas. Unos y otros se preocupan exclusivamente del problema real que se trata: la búsqueda de la relación de un orden total para un conjunto de elementos sin ningún tipo de restricción.

El análisis profundo intenta que satisfaciendo las mínimas restricciones posibles se describa el procedimiento que realiza la ordenación de elementos. La aplicación del concepto abstracción permite centrar la atención en los aspectos esenciales inherentes al propio proceso de ordenación, ignorando aquellas propiedades accidentales que, por una u otra circunstancia, derivan de una implementación concreta o de unas estructuras de datos y control determinadas. Sólo desde una solución abierta y abstracta es posible encontrar los puntos comunes de los algoritmos de ordenación que, sorprendentemente, conducen a una serie de conclusiones inesperadas antes de comenzar el estudio.

El germen de la idea tiene su origen en la necesidad (y al mismo tiempo inconveniente) de adaptar cuando se necesita conocer la relación de orden total de un conjunto de elementos, el algoritmo de ordenación clásico que se seleccione a las estructuras de datos y control disponibles en el lenguaje de programación elegido. Como añadido, el sistema hardware en que se deba realizar la ejecución impone más restricciones aún si cabe a dicha adaptación. Bajo estas premisas, la ordenación se presenta como la realización de una tarea muy rígida que impide cualquier tipo de adaptación.

En el desarrollo de sistemas abstraer los conceptos fundamentales permite centrar toda la atención sobre el problema a resolver: en este caso se debe priorizar la atención sobre el cometido y funcionamiento del propio proceso de ordenación. El resto de información añadida que se encuentra alrededor del mismo no hace sino enmarañar el estudio del problema, distrayendo la atención con detalles de implementación que lejos de acercarnos a una solución eficiente, únicamente desvían la atención del núcleo del problema.

Ese es uno de los pilares fundamentales sobre los que se apoya el desarrollo de esta *Tesis:* tratar de eliminar cualquier imposición o restricción en la búsqueda de una solución. Restricciones que por norma general resultan ajenas a la naturaleza del problema, considerándose superfluas e innecesarias para su resolución. Sin embargo marcan y condicionan el camino a seguir hacia una

cristalización de la solución dependiente por completo de factores externos.

Conceptualmente el proceso de ordenación de elementos es simple de explicar, sencillo de comprender y fácil de transmitir. Sin embargo las múltiples soluciones utilizadas para resolver el problema a lo largo de la historia de la informática, se han visto siempre enmarañadas con detalles de implementación que lejos de acercarnos a una óptima solución, restringen y condicionan los mejores desarrollos basados únicamente en la naturaleza intrínseca del problema.

3.4. Análisis del problema

El problema en cuestión radica en obtener una relación de orden total para un conjunto $\mathcal{E}$ de elementos, cuya naturaleza es irrelevante salvo por la operación de comparación entre dos de ellos. El orden de aparición de dichos elementos y la temporalidad con que se presentan los mismos no está determinada: al comienzo de la ejecución no necesariamente se ha de disponer de todos los elementos a tratar.

Realizando una definición más formal del problema, la solución buscada pretende satisfacer la necesidad de encontrar, a partir de un conjunto de elementos de entrada, una correspondencia biyectiva entre dicho conjunto de elementos y un intervalo de los números naturales.

Sea un conjunto de elementos $\mathcal{E}$ de cualquier naturaleza cuya cardinalidad asciende a un valor n (siendo $n \geq 0$). Se trata de encontrar una correspondencia entre cada uno de esos n elementos y el conjunto $[1\,..\,n]$ (para $n = 0$ se devuelve directamente $\emptyset$) de los números naturales, que cumpla la condición de poseer una relación de orden total en función de la operación de comparación establecida. Lógicamente dada la irrelevancia de la naturaleza de cada conjunto de elementos,[2] se hace necesario definir para cada ocasión en qué consiste la operación de comparación entre dos elementos cualesquiera del conjunto, así como los diferentes resultados que se obtienen tras realizar la misma.

3.4.1. Ángulos

La anteriormente citada correspondencia biyectiva es vista en ambos sentidos: desde el punto de vista de un elemento encontrar la posición final que ha de ocupar. O bien, desde el punto de vista de todas las posiciones disponibles encontrar para cada una el elemento que le corresponde.

[2] En cuanto al establecimiento de la relación de orden.

Esta descripción se corresponde con las dos alternativas posibles de pregunta:

- Fijado un elemento, ¿qué posición ha de ocupar?.
- Fijada una posición, ¿a qué elemento le corresponde ocuparla?.

Puesto que la tarea a realizar se puede expresar en cómo hallar la asociación entre elementos y posiciones, los procedimientos locales son los encargados de responder a uno de esos dos planteamientos.

El procedimiento que se establezca para resolver el problema de forma completa debe estar necesariamente contenido en una de estas alternativas:

- Para cada elemento de $\mathcal{E}$ calcular su posición[3] (se fija el elemento y se ha de encontrar la posición).
- Para cada número natural en $[1\,..\,n]$ encontrar el elemento $x \in \mathcal{E}$ que le corresponde (se fija la posición y se ha de encontrar el elemento que ha de ocuparla).

Cada una de estas alternativas produce como resultado el desarrollo de un meta-algoritmo, satisfaciendo una de las dos cuestiones anteriores.

La dualidad que se manifiesta entre ambas preguntas se propaga a los meta-algoritmos descritos en los capítulos 4 y 5 a los que dan lugar. La cuestión que fija un elemento y busca la posición que ha de ocupar da lugar al meta-algoritmo descendente. De forma dual la cuestión que fija una posición y busca el elemento que ha de ocuparla genera el meta-algoritmo ascendente.

3.4.1.1. Primeros intentos del proceso de mejora

Tal y como se describe en el capítulo "Ordenación serie y paralelo" (apartado 2.3.1) la adaptación de un algoritmo serie a su versión paralela requiere en la mayoría de los casos y siempre que esta sea posible, la construcción de un hardware específico. Esta "inflexibilidad" a la hora de hacer realidad una versión paralela de cualquier algoritmo de ordenación serie la convierte prácticamente en una utopía, al ser exagerados los costes de construcción de las arquitecturas descritas. Y por si fuera poco, esta solución hardware sólo sirve para resolver un problema muy concreto y determinado, imposibilitando su desarrollo como solución generalizada.

[3] El número natural que indica su posición.

Plasmar como realidad en el desarrollo de software el concepto de "paralelismo" requiere como condición necesaria el uso de varios procesadores. Hoy en día cualquier equipo medio del mercado dispone de varios núcleos en los que ejecutar tareas de forma simultánea, por lo que si se diseñan los diferentes procesos del algoritmo de forma que aprovechen ese paralelismo hardware soportado se pueden obtener mejores tiempos de ejecución.

No obstante la aplicación de concurrencia no es la panacea, ya que en muchas ocasiones la mejora de tiempos teórica que se debiera obtener gracias a su utilización, se ve reducida a "cenizas" (empeorando incluso en ocasiones la situación) debido a los tiempos de transmisión de resultados entre procesadores y entre estos y las memorias.

Disponer de una posibilidad real de paralelismo sin desembolsar ingentes cantidades de dinero en hardware es uno de los pilares fundamentales que sustentan los inicios y el germen del desarrollo del meta-algoritmo. Habitualmente cuando se trabaja sobre una base de tiempos críticos, se está dispuesto a desembolsar las cantidades económicas que sean necesarias para conseguir la ansiada reducción de tiempos. Lógicamente las soluciones cuyo fundamento se basa en realizar inversiones hardware no pueden ser consideradas como soluciones generalizadas, ya que lo habitual es trabajar con una disposición de recursos limitada. Aprovechar al máximo dichos recursos sólo es posible desde un diseño flexible, práctico y adaptable al entorno en que se quiere ejecutar, y no al revés.

La pregunta principal que se plantea es: *¿por qué no pensar en obtener un meta-algoritmo de ordenación que, de una manera totalmente abstracta, encarne a varios (o todos) algoritmos clásicos de ordenación?*. Para responder satisfactoriamente a dicha cuestión es necesario llevar a cabo la construcción, mediante alguna forma de parametrización sencilla, de un meta-algoritmo que pueda especializarse en varios algoritmos (tanto software como hardware) que permita alternar en la misma ejecución entre varios métodos de ordenación diferentes, dotando al propio meta-algoritmo de una posibilidad de paralelismo real sin grandes costes de infraestructuras y, lo que es más importante, alejándolo al máximo de la inflexibilidad en su definición.

Los capítulos 4 y 5 detallan las estructuras necesarias para la construcción de un meta-algoritmo que satisfaga la cuestión planteada. Para ello debe cumplir con los requisitos iniciales permitiendo la mayor adaptabilidad posible en su proceso de ejecución. Ambos capítulos desarrollan la definición y descripción de los principales componentes del meta-algoritmo así como el detalle de ejecución de procesos para cada situación.

3.4.1.2. Meta-algoritmos de ordenación: fundamentos

Consisten en la especificación de las acciones necesarias para resolver un problema con el más alto nivel de abstracción posible eliminando todos los detalles irrelevantes de implementación, tanto de las estructuras de control como de las estructuras de datos. Su éxito radica en la especificación de las acciones a realizar para resolver el problema mediante una especificación abstracta, sin detallar ningún tipo de estructura de datos ni de control.

Un algoritmo abstracto hace el papel de un teorema y sólo tiene en cuenta la abstracción esencial mínima necesaria para la resolución del problema, que es tanto más fuerte cuanto más débil sea la hipótesis (es una situación similar a la "precondición" más débil de *Hoare*). Una implementación no es más que una mera particularización que tiene en cuenta otros factores externos. Un meta-algoritmo se define como la especificación de las acciones a realizar para resolver un problema, centrándose únicamente en la esencia y fundamento del mismo.

Para resolver cualquiera problema lo fundamental es preocuparse por la naturaleza del mismo, apartando de la mente todos los aquellos detalles de programación que no aportan nada a la solución, salvo restringirla. Por tanto se ha de dejar para las últimas fases aquellas particularizaciones de cada caso que cristalizan en una solución bajo una determinada implementación.

De esta forma, la verificación de la corrección del proceso se hace sin necesidad de atender a la forma de representación final, la cual es detallada en el último paso. Si una abstracción cumple las condiciones y es verificable (y demostrable), las particularizaciones que de ella se deriven también lo son.

Las estructuras de control del meta-algoritmo son las mínimas que cualquier implementación derivada de este ha de cumplir; de igual modo, las estructuras de datos especifican sólo las operaciones que intervienen en el proceso.

La corrección de una implementación concreta se reconduce entonces a la verificación de las condiciones mínimas del meta-algoritmo por parte de la implementación, que queda así en libertad de atender a las condiciones externas y las decisiones de optimización que se estimen convenientes.

Por otra parte, un meta-algoritmo da lugar a distintas implementaciones que aprovechen los grados de libertad de una especificación no detallada, permitiendo por ejemplo realizaciones secuenciales o que aprovechen el posible paralelismo entre (o entremezclado de) acciones cuyo orden de realización es irrelevante.

La elección de una estructura de datos particular restringe las posibilidades del resto de la implementación de la misma forma que una elección sobre una estructura de control a utilizar condiciona, en ocasiones de forma muy restrictiva, la implementación. Estas elecciones pueden venir impuestas por el entorno externo al núcleo del problema.

Un ejemplo de la misma idea aplicada al análisis sintáctico se encuentra en [11] [12].

La utilización del meta-algoritmo de ordenación cuya capacidad de abstracción se encuentra muy por encima de la que presentan los algoritmos "clásicos", permite sin necesidad de conocer detalles propios de cada implementación, describir los procesos que resuelven el problema planteado abstrayendo todos aquellos detalles que son ajenos al propio proceso de ordenación.

No obstante y como parece lógico, esta abstracción necesita una serie de pautas, indicando qué acciones preceden a otras en orden de tiempos pero sin imponer restricciones innecesarias. Esto posibilita una adaptabilidad máxima a las circunstancias cambiantes del medio y, lo que es más importante, permite una ejecución simultánea de procesos independientes, lo que dota al meta-algoritmo de una posibilidad real de paralelismo.

El meta-algoritmo (o meta-algoritmos, para ser más precisos) desarrollado desde sus fases iniciales de análisis y diseño[13], ha sido concebido de forma que cada proceso creado se comunica e intercambia información única y exclusivamente con su *proceso padre* y su/s *proceso/s hijo*. De esta forma se obtiene una independencia en el método de trabajo que posibilita la ejecución simultánea de tantos procesos como disponga cada ejecución.

3.4.1.3. Para-algoritmos

Con la descripción anterior que define un meta-algoritmo se imponen una serie de restricciones que, lógicamente han de seguir respetando las premisas iniciales sobre las cuales fue diseñado.

Entre esas restricciones se encuentra el establecer una imposición en el orden de actuación de los procesos. Dicha restricción lo único que hace es concretar más el meta-algoritmo, pero no lo invalida. La imposición de dichas restricciones hacen que se pase de un *meta-algoritmo* a un *para-algoritmo*. En este caso, las estructuras de control ya quedan fijadas al ser impuesto el orden de actuación de los procesos.

El siguiente paso que se debe llevar a cabo es el de fijar las estructuras de datos. Estas han de respetar y cumplir las condiciones iniciales establecidas en la definición para que la corrección del meta-algoritmo no se vea afectada. Además se debe disponer de las operaciones necesarias que se encuentran en

la especificación, al más alto nivel posible, como cualquier tipo abstracto de dato con el que se quiera trabajar.

El nivel más bajo al que se debe llegar es el de la genericidad de los elementos a comparar. Dado que la naturaleza del conjunto de elementos de entrada del meta-algoritmo es irrelevante, se ha de definir la forma de establecer la relación de orden entre dos elementos cualesquiera del conjunto, detallando las acciones a realizar ante la obtención de las diferentes respuestas que se presenten.

Una particularización lo que pretende es simplificar la estructura de datos o la estructura de control (dinámica de ejecución). En general puede decirse que ambos aspectos se influyen mutuamente, de manera que una restricción en la estructura de datos implica una limitación de la estructura dinámica y, de forma recíproca, las restricciones en la estructura de control exigen de una estructura de datos en particular.

En cuanto a la forma definitiva de cristalizar una implementación del meta-algoritmo, se puede sugerir como la solución más sencilla y eficiente de todas la utilización de los *generic* de *Ada*. No se recomienda utilizar los tipos más genéricos de Java o C++ al no disponer de subprogramas que sean pasados como parámetros, lo que limita a una sola operación la comparación.

El siguiente nivel representable en *Ada* a base de *packages* genéricos permite especificar las operaciones y condiciones a cumplir por la estructura, dejando para una *subunit* la implementación. De esta forma se transfiere al lenguaje la responsabilidad de verificar la consistencia de la implementación determinada.

3.5. Convenciones

Para desarrollar el meta-algoritmo es necesario definir una serie de premisas básicas sobre las que apoyarse relacionadas con los posibles resultados obtenidos al comparar los elementos a procesar.

Según la interpretación elegida al realizar la comparación se obtienen diferentes estados finales en el orden de los elementos, siendo en cualquier caso todos ellos equivalentes. Las convenciones establecidas en este apartado y siguientes pueden ser alteradas sin que, por ello, cambie en absoluto el resultado final proporcionado por el meta-algoritmo: basta con alterar la interpretación que se realiza de la solución proporcionada por este sin perdida alguna de generalidad.

Al tratar el meta-algoritmo con elementos de cualquier naturaleza se hace necesario definir la operación (u operaciones) a realizar en el proceso de comparación, estableciendo la interpretación del resultado obtenido.

La forma más sencilla de definir la operación que establezca el orden final es tomar los elementos del conjunto a tratar de dos en dos, y realizar su comparación. Han de ser definidas las posibles respuestas obtenidas así como la forma de proceder en cada caso.

Supongamos que se dispone de un conjunto $\mathcal{E}$ de elementos para ordenar. Se define la operación de ordenación Φ entre pares de elementos de la siguiente forma: se toman 2 elementos α y β cualesquiera pertenecientes al conjunto $\mathcal{E}$. La operación de comparación $\alpha\ \Phi\ \beta$ devuelve uno de los siguientes resultados:

- $\alpha\ \Phi\ \beta \Rightarrow \alpha$, si $\alpha\ >\ \beta$
- $\alpha\ \Phi\ \beta \Rightarrow \beta$, si $\alpha\ <\ \beta$
- $\alpha\ \Phi\ \beta \Rightarrow Iguales$, si $\alpha\ =\ \beta$

Así definida la operación Φ, para cada pregunta que se realiza se dispone de 3 posibles alternativas de respuesta. Sin embargo es fácil simplificar el problema reduciendo el número de posibles respuestas de tres a dos. Cuando la respuesta de la función Φ es la igualdad, realmente puede ser incluida en cualquiera de las otras dos posibilidades sin pérdida alguna de la generalidad.

No importa saber la precedencia de dos elementos cuyas claves de ordenación son idénticas, y dado que una comparación con resultado de igualdad resulta irrelevante en cuanto a la corrección del proceso a tratar se refiere, puede incluso posponerse la elección de la alternativa hasta el último momento de la implementación, permitiendo de esta forma aquella que en cada caso concreto tenga un menor coste de ejecución. En caso de que existan m *elementos* repetidos en el conjunto de entrada se obtienen $m!$ soluciones diferentes, pero todas ellas equivalentes, ya que sólo se encuentra alterada la posición de los elementos cuyas claves de ordenación son idénticas.

Con esta propuesta se pretende reducir las 3 posibles alternativas de respuesta (que implican 2 preguntas para obtener el orden de dos elementos) a sólo 2 (caso que se resuelve con una única pregunta). Al quedarnos sólo con 2 posibles alternativas para cada comparación, el meta-algoritmo realiza su trabajo como si fuese la "bandera portuguesa" descrita por Dijsktra en contraposición a la bandera Francesa (Holadesk). Mantener las 3 posibilidades de respuesta incrementa la complejidad del algoritmo y en la mayoría de los casos produce beneficios puramente marginales. Por tanto no compensa el esfuerzo adicional que implica la distinción de las 3 alternativas.

Preguntar si la relación entre 2 elementos del conjunto de entrada es $<$ o $\leqslant$ (o de forma análoga preguntar por $>$ o $\geqslant$) es indiferente, y se dirige la alternativa de igualdad hacia la posibilidad que se desee. El resultado es un meta-algoritmo más claro, menos complejo y más sencillo de verificar formalmente.

La simplificación del problema con la reducción de alternativas se toma como premisa fundamental en el desarrollo de los meta-algoritmos descritos.

3.5.1. Elementos repetidos: una restricción que no es tal

En lo que sigue, se supone que los elementos pertenecientes al conjunto $\mathcal{E}$ disponen siempre de claves de ordenación diferentes, lo que se traduce en que: para todo elemento α perteneciente al conjunto $\mathcal{E}$ la clave de ordenación es siempre única. Esto simplifica la discusión de cualquier aspecto de los procedimientos de ordenación al no tener que estar constantemente haciendo salvedades para los casos de elementos que posean la misma clave.

En la práctica se producen dos alternativas. La diferencia radica en que se admita que los elementos con la misma clave se puedan encontrar en cualquier posición relativa o, por otro lado, se añada alguna condición que imponga un orden entre ellos.

Puede ser visto como si en un principio lo que se define es un orden parcial en el conjunto $\mathcal{E}$ de los elementos a tratar. Dicho orden parcial se define por una partición C_1, C_2, ..., C_n, en la que los elementos C_i son anteriores a los C_j cuando se cumple que $i < j$. Por otro lado, entre los elementos de un conjunto de la partición o bien cualquier orden es válido (primer caso), o bien se especifica un orden secundario entre ellos (segundo caso), entendiendo que los elementos de un conjunto C_i comparten la misma clave.

En el primer caso la solución obtenida no es única, puesto que cualquier orden total compatible con el orden parcial definido constituye una solución válida.

En el segundo caso lo que se produce en realidad es una especificación de un orden total en dos partes: la clave del elemento por un lado y el orden secundario entre los elementos del mismo conjunto por otro (aquellos cuya clave de ordenación sea idéntica). En estos casos se interpreta como clave efectiva del procedimiento una clave compuesta formada por: la clave inicial y una clave secundaria, que es la encargada de diferenciar aquellos elementos con idéntica clave inicial.

Para establecer el orden secundario pueden utilizarse diversas alternativas, las cuales añaden cierto grado de complejidad al proceso de comparación.

Por este motivo la elección de una clave secundaria lo más sencilla y rápida de comparar posible, se considera un aspecto fundamental en el proceso inicial de definición. Algunas de las claves secundarias más sencillas que se proponen son las siguientes:

- Utilizar el orden de llegada de los elementos al conjunto de entrada, estableciendo en caso de igualdad el elemento que llegue antes al conjunto de entrada como ganador.
- Utilizar la posición relativa que ocupa cada elemento dentro del conjunto a tratar. De esta forma, en caso de igualdad es considerado como elemento ganador aquel que ocupe una posición precedente dentro de dicho conjunto.
- Añadir un campo de tipo *timestamp* a cada elemento, de manera que es considerado como elemento ganador de la relación de orden aquel que disponga de la fecha más antigua.
- Cualquier otro tipo de dato o información que permita diferenciar unívocamente a todos los elementos del conjunto a tratar, permitiendo la obtención de una relación de orden total.

Con cualquiera de las anteriores interpretaciones, esa aparente restricción que supone la necesidad de utilizar una clave secundaria no representa limitación alguna. Solamente existe un punto en la primera interpretación, en que se pueden presentar problemas que degeneren en una circularidad obviamente indeseable.

La circularidad se produce cuando la comparación entre dos elementos con idéntica clave se responde de diferente manera en dos momentos distintos t y t' de la ejecución del meta-algoritmo. Si en un primer momento se responde afirmativamente a $a \leqslant b$ y más adelante se responde afirmativamente a lo "contrario", $b \leqslant a$, se genera una circularidad que ocasiona un comportamiento no deseado impidiendo la finalización del meta-algoritmo. Este comportamiento indeseado, sólo ocurre si ambos elementos tienen la misma clave.[4]

En cierta medida esta circularidad aparece porque se está haciendo una pregunta inútil, puesto que ya se conoce la respuesta. El problema no se produce por realizar comparaciones directas entre dos elementos. En situaciones

[4]Podría pensarse que una disciplina que evite la respuesta afirmativa a ambas preguntas elimina el problema, pero preguntar por si $a < b$ y $b < a$ dando respuestas negativas, lleva a que el procedimiento considere que no hay relación entre ambos elementos, es decir, que no se trata de un orden total lo que se intenta construir.

más elaboradas podría ser que $a_1 \leq a_2 \leq a_k \leq a_1$, con comparaciones en un ciclo.

En este caso la inutilidad de la pregunta que cierra el ciclo, consiste en que la respuesta podría ser deducida de las comparaciones precedentes puesto que la relación a construir es transitiva. El resultado es desastroso. Se puede dar el caso de que un procedimiento que intercambie las posiciones provisionales de dos elementos entre en un bucle de intercambio sin fin impidiendo alcanzar un estado final.

Dado que la situación sólo se produce cuando se comparan dos elementos en más de una ocasión y se obtienen respuestas inconsistentes, cualquier procedimiento que no realice comparaciones cuyo resultado ya es conocido (que no formule preguntas inútiles) o que sea deducible por las respuestas de comparaciones precedentes, está a salvo.

En cualquier caso, una pregunta inútil no posibilita que el procedimiento avance hacia la solución final al no aumentar la información de que se dispone.[5]

3.6. Optimalidad frente a simplicidad y seguridad

Cuando se plantea un problema del día a día cotidiano, se suele disponer de múltiples alternativas que lo resuelven (eficacia) por caminos diferentes, cuya complejidad y coste de realización suele ser dispar. Entre las alternativas posibles unas son más sencillas de realizar que otras, unas necesitan más tiempo para su ejecución y, lógicamente, no todas necesitan utilizar los mismos recursos. Por todo ello la solución seleccionada no ha de ser necesariamente la más sencilla, o la más rápida, o la que menos recursos consuma: se intenta elegir aquella que presente el mejor equilibrio entre todos estos factores, siendo propuesta como la solución más apropiada al problema planteado.

Si se traslada la misma filosofía al mundo tecnológico sucede algo similar. No es infrecuente que para resolver un problema se utilice un algoritmo del que se conoce de antemano que no es óptimo frente a otros. Sin embargo es preferido por la simplicidad de su representación, que redunda en facilidad de verificación y por lo tanto en seguridad.

La visión propuesta al diseñar los meta-algoritmos persigue conseguir ya desde las fases iniciales de análisis y diseño, y durante todo el proceso de desarrollo, un equilibrio entre la necesaria eficacia en la resolución del problema y los muy deseables conceptos de sencillez y eficiencia. De esta forma la

[5] Desde el punto de vista de las máquinas abstractas, es como si se realizara una transición entre estados igual de alejados de el (o los) estado final, pudiendo entrar en un bucle sin fin.

solución desarrollada puede ser aplicada a la resolución de problemas de naturaleza compleja sin que ello implique la necesidad de trabajar con un algoritmo complejo.

3.6.1. Situación de Strassen

Cuando se busca una solución para un problema complejo, es muy frecuente que el proceso óptimo que elabora la solución presente una complejidad combinatoria molesta. Esto no es deseable ya que la resolución de problemas mediante soluciones complejas, suele traer como consecuencia el rechazo a la utilización de dicha solución por parte de potenciales usuarios interesados.

Por este motivo se opta por un proceso que permita una expresión o solución igual de válida pero con menor complejidad y que, aunque no sea óptima, sea expresable (y verificable/demostrable) con mayor simplicidad. Es posible que no se realice un aprovechamiento máximo de los recursos, e incluso que su tiempo de ejecución sea peor que el de otras soluciones con mayor complejidad, pero la sencillez en la representación de la expresión resultado la hacen atractiva para su puesta en práctica.

Un ejemplo claro es el utilizado para resolver sistemas de ecuaciones lineales. El algoritmo habitual que se emplea es el de eliminación de Gauss (o el de Jordan) que presentan una complejidad cúbica en su resolución. Sin embargo se sabe que no es óptimo[14], ya que se conoce una alternativa mucho más óptima para resolver el problema: *Strassen.*[6]

Si la solución de Strassen es más óptima que la propuesta por Gauss, ¿por qúe no es el método utilizado habitualmente para resolver sistemas de ecuaciones?. La respuesta tiene una sencilla explicación: la expresión que emplea Strassen para representar la solución es mucho más compleja que el método de Gauss, prefiriéndose la solución más simple de ambas. El coste en tiempo de ejecución al utilizar Gauss es realmente marginal, y más en los casos de aplicación práctica en los que lo más habitual es que los valores se encuentren en un rango con $n \leq 30$ o $n \leq 50$.[7] La pérdida que se produce en tiempo de ejecución al utilizar Gauss se ve compensada por una mayor simplicidad de la expresión solución, lo que provoca que esta sea más fácil de verificar (o de ser aceptada como correcta), siendo elegida como la solución preferida.

[6] El método de Strassen presenta un orden de complejidad de $n^{\log_2 7} \approx n^{2,807}$. Desarrollos posteriores han ido rebajando esa cota a $n^{2,3736}$ y $n^{2,3736}$.

[7] Las cotas dadas solamente se aproximan para valores muy grandes de n, en cuyo caso no se utilizan métodos directos de resolución de sistemas de ecuaciones lineales por no ser numéricamente estables.

3.7. Obtención de resultados en orden inverso

Para realizar el proceso de ordenar cualquier conjunto de elementos sea cual sea su naturaleza, lo primero que se debe conocer es el orden, ascendente o descendente, en que se desea obtener el resultado. En el desarrollo de todos los ejemplos de los meta-algoritmos utilizados, se toma como premisa que la salida buscada se proporciona en orden creciente ($s_1 \leqslant s_2 \leqslant ... \leqslant s_n$).

No obstante realizar la ordenación de elementos en *orden inverso* (con las premisas establecidas consiste en proporcionar una salida en orden decreciente), no añade complejidad alguna al meta-algoritmo: es más un problema de disponibilidad inicial sobre la información en el orden de salida que un problema añadido al proceso de ordenación.

Al comenzar la ejecución del meta-algoritmo se pregunta al usuario si puede facilitar la información sobre el orden deseado en que han de encontrarse los elementos al finalizar la ejecución: conocer al inicio del proceso si la salida se ha de proporcionar en orden creciente o decreciente permite encauzar desde el inicio el desarrollo del meta-algoritmo.

De cualquier forma, es muy probable que existan ejecuciones en las que el usuario no facilite esa información por desconocimiento de la misma al comienzo del proceso y esta solo se obtenga en un punto más avanzado de la ejecución. En estos casos la premisa adoptada es la descrita anteriormente, estableciéndose el orden de salida en forma creciente.

Si una vez obtenido el resultado del meta-algoritmo este es proporcionado en el orden inverso al deseado, invertir la salida se convierte en una tarea rutinaria: con el resultado final disponible ($s_1 \leqslant s_2 \leqslant ... \leqslant s_n$) compuesto por los n elementos ordenados se puede realizar una sencilla operación mediante una estructura de datos abstracta tipo "pila". La solución deseada, que es inversa a la ya proporcionada, es tan sencilla como apilar los n elementos de la solución en el orden producido para en una segunda fase "desapilarlos", enviando cada *elemento desapilado* a la salida del meta-algoritmo hasta que la pila se encuentre vacía.

El único inconveniente que presenta invertir el resultado una vez concluida la ejecución del meta-algoritmo es el incremento en el tiempo de ejecución total, que se ve afectado (según el número de elementos a tratar) por los procesos *Apilar* y *Desapilar*. Lógicamente el tiempo de ejecución es menor si el usuario facilita la información del orden de salida al comienzo de la ejecución.

Si en contra a la premisa establecida el usuario facilita al inicio del proceso su preferencia por una salida en orden decreciente, todas las referencias realizadas en los meta-algoritmos a los "procesos *hijo derecho*" deben ser

entendidas como "procesos *hijo izquierdo*". Y de forma análoga, aquellas realizadas a "procesos hijo izquierdo" deben ser interpretadas como "procesos hijo derecho". Con estos pequeños ajustes (que no alteran de modo alguno ni la ejecución ni la verificación formal del meta-algoritmo) la solución obtenida se encuentra ordenada en forma decreciente, evitando así el uso de un *TAD* pila.

3.8. La pregunta más inteligente

En la búsqueda de la mejor solución, la pregunta más inteligente a formular en cada instante es analizada desde dos puntos de vista diferentes que son ambos compatibles entre sí.

Cuando se realiza una pregunta, la nueva relación de orden parcial (o el nuevo estado del procedimiento) es la mínima relación de orden que contiene a la anterior y que añade la nueva información obtenida tras la comparación realizada.

Cada pregunta que se realiza (comparación) entre dos elementos genera dos posibles respuestas. Bajo un primer punto de vista, la pregunta que proporciona más información es aquella cuya respuesta divida las posibilidades lo más igualmente posible. Entre todos los órdenes totales compatibles con el orden parcial obtenido, la mejor pregunta es aquella que permite dividir el conjunto (universo desconocido) en dos subconjuntos cuyos tamaños sean lo más iguales posible ("equiparta el conjunto de entrada").[8]

Un segundo punto de vista alternativo se basa en tratar de construir una relación de orden total. En un momento intermedio se dispone de una relación de orden parcial que no es más que un subconjunto de la relación buscada. Por este motivo la mejor pregunta es aquella que más acerque la relación de orden parcial a la relación de orden total buscada.

Cuando se responde a una pregunta que compara dos elementos no se resuelve únicamente la relación de orden parcial entre ambos elementos. Al disponer de una relación parcial esa información puede ser aprovechada y propagada a la relación con otros elementos por transitividad. Por ello un procedimiento es tanto mejor en cuanto no ignore información de la que dispone hasta ese momento.

En resumen se puede aseverar que un procedimiento pierde eficiencia por:

- Repetir preguntas.

[8] Puesto que la pregunta sólo tiene dos respuestas posibles, es la que da más información según establece la teoría de información de Shannon.

- Hacer preguntas cuya respuesta podría haberse deducido de la información disponible (transitividad).
- Hacer preguntas que no proporcionen el máximo de información posible.

Cuando se realiza una pregunta cuya respuesta es deducible de la información ya obtenida, claramente la pregunta es redundante y por lo tanto inútil. Esta consideración representa uno de los principales problemas del algoritmo de ordenación *Burbuja*: en su ejecución realiza innumerables preguntas redundantes al desperdiciar gran cantidad de información.

En el capítulo 7 apartado 7.3.5, se desarrolla una posible optimización de dicho algoritmo con el objetivo principal de evitar el desperdicio de información obtenida, algo que lamentablemente no consigue el algoritmo en su versión más pura ni tampoco en su versión mejorada.

3.8.1. Estrategias

Para realizar la pregunta más inteligente se establecen dos estrategias:

1.- Hacer en cada momento la pregunta que proporciones el máximo de información dependiendo de toda la historia anterior, considerando el conjunto de permutaciones compatibles con las respuestas obtenidas hasta ese momento. Esto da lugar a procedimientos óptimos a cambio de una representación más complicada (casuística). Al tener en cuenta toda la historia anterior se les denomina procedimientos globales.

2.- Proponer un esquema más lineal que aunque desperdicie parte de la información obtenida sea de estructura más simple. Se adopta un punto de vista y se tiene en cuenta únicamente la información referida al mismo. Son procedimientos locales que pueden ser razonablemente optimales (*Quicksort*) o alejados de este (*Burbuja*).

3.8.2. Consideraciones

El trabajo está delimitado por la restricción (consideración) de que todas las acciones reducen la incertidumbre. Hay dos consideraciones que justifican esta decisión:

- Las acciones que no reducen la incertidumbre pueden realizarse en bucle, por lo que el procedimiento no termina. Esto exige la necesidad de demostrar que no hay bucles, complicando la verificación de la corrección.

- Las acciones de este tipo pueden ser consideradas como acciones de mantenimiento[9] y colisionan con las acciones útiles, desapareciendo del proceso como tales.

El trabajo contempla la colateralidad eficiente y orienta en la consideración de ineficiencias que reducen el tiempo dentro de las restricciones al número de procesadores. Considerar el disponer de un número arbitrario de procesadores no es intelectualmente productivo. El "intuicionismo" no es una función computable.

Si la colateralidad no presenta redundancias, nos encontramos ante nuestro caso.

Si se admiten redundancias (acciones cuyo resultado es descartado) se supone que la pérdida de eficacia se ve compensada con una reducción del tiempo global.

Se prueba esta afirmación mediante la utilización de un ejemplo gráfico:

Supongamos que se dispone de una cadena con 3 elementos a, b, c ordenados, y se pretende realizar la inserción del elemento x, del cual se desconoce relación alguna con los elementos que componen la cadena.

Para realizar una inserción binaria se compara el elemento x con el elemento b (como se muestra en la figura 3.1) y, en función del resultado, se procede a comparar el elemento x con a o c. Por tanto el *delay* del proceso es igual a 2 (son necesarios 2 ciclos de procesador). Lógicamente cuanto mayor es el número de elementos de la cadena en que insertar x mayor es (en término medio) el valor del *delay.*

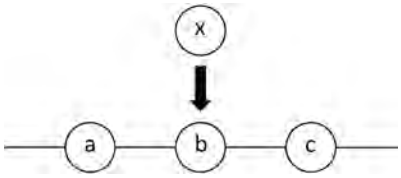


FIGURA 3.1: Inserción binaria de un elemento

Si se renuncia a la eficacia del paralelismo permitiendo comparaciones entre elementos que bajo una premisa de eficiencia se muestran como superfluas, se realiza de forma simultánea la comparación del elemento x con todos los elementos que conforman la cadena (en este caso los elementos a, b y c).

[9] Mantenimiento que corresponde a un cambio de punto de vista con el fin de facilitar la determinación de las próximas acciones a realizar.

Al realizar la comparación se obtiene una cadena de bits formada por los siguientes valores:

- Si $x < y$, se obtiene un 0.
- Si $x > y$, se obtiene un 1.

De esta forma al comparar en paralelo y en un único ciclo de reloj (*delay uno*) el elemento x con los 3 elementos, se obtiene una de las siguientes combinaciones de bits:

- 000, si x es mayor que a, b, c.
- 001, si x es mayor que a y b.
- 011, si x sólo es mayor que a.
- 111, si x es menor que los 3 elementos.

En un único ciclo de reloj se obtiene la posición que ocupa el elemento en la cadena. Para ello, al resultado obtenido basta con añadirle un 0 al inicio y un 1 al final y buscar en la nueva secuencia de bits resultante la secuencia 01. Esa es la posición en que se debe insertar el elemento x.

Ejemplo: Imaginemos que tras la comparación la cadena devuelta es 001. Al añadir los dos bits mencionados se obtiene 00011. Basta buscar el patrón 01 para conocer que el elemento x ha de ser insertado entre los elementos b y c.

Con este método, a pesar de realizar comparaciones innecesarias, el tiempo empleado para encontrar la posición del elemento es siempre el mismo (independientemente de la longitud de la cadena en que ha de ser insertado): se emplea un único ciclo de reloj para realizar la comparación del elemento a insertar con todos los elementos de la cadena, y otro ciclo de reloj para encontrar el patrón 01 en el resultado. Aunque de forma local se muestra el proceso como menos eficiente (por realizar comparaciones innecesarias), en términos globales el tiempo de ejecución se reduce considerablemente.

3.9. Visión relacional del problema: representaciones

A lo largo del proceso de ejecución del meta-algoritmo en la búsqueda del orden total del conjunto de elementos de entrada, se hace necesario analizar una situación intermedia que exprese un orden parcial representando el camino

recorrido en busca del estado final. Se ha de permitir mostrar para un instante concreto de la ejecución el universo de elementos que restan por analizar.

Como lo que se pretende es hallar una relación de orden total para un conjunto de elementos cuya construcción paulatina a base de comparaciones entre dos elementos proporciona únicamente la relación existente entre ambos, se puede decir que se parte de una relación mínima de orden en la que cada elemento sólo está relacionado consigo mismo, y del resto de pares de elementos se ignora toda relación.

En pasos sucesivos se construyen relaciones de orden que contienen a la anterior (estrictamente si se hace la comparación adecuada) hasta obtener el orden total buscado. Para completar el proceso, se parte de un estado inicial en que se desconoce todo acerca de las posiciones relativas de los elementos, hasta llegar a un estado final en que se obtiene una relación de orden total entre ellos. Los pasos sucesivos se realizan gracias a las respuestas a comparaciones adecuadamente recogidas.

En el proceso de ejecución se aprovecha el hecho de estar buscando una relación de orden, con las propiedades reflexiva y transitiva que ello conlleva. Al inicio del proceso la relación de orden es la mínima, mientras que al final del mismo la relación de orden es total.

En cualquier otro momento intermedio del proceso, la relación parcialmente construida es la mínima compatible con las respuestas a las comparaciones realizadas hasta ese momento.

Como resulta lógico, la relación parcial intermedia es utilizada para optimizar el proceso (aunque este no sea el objetivo principal del mismo). Por ejemplo la citada propiedad de transitividad, se utiliza de forma sencilla para evitar comparaciones cuya respuesta pudiera ser deducida de respuestas a otras comparaciones precedentes. Si se conoce por un lado la relación de elementos $a < b$ y por otro se deduce que $b < c$, por transitividad, se concluye que $a < c$. Realizar la pregunta que compare los elementos a y c además de ser innecesaria, implica una pérdida de tiempo en la ejecución.

Como complemento a las descripciones algebraicas formales y dada la naturaleza del problema, las formas más habituales para representar una relación de orden son:

- Mediante un grafo dirigido acíclico, se muestra el conocimiento parcial de las relaciones entre los elementos del conjunto a tratar.
- Mediante la matriz de la relación, que representa para un instante concreto la relación existente entre los elementos del conjunto (lo que se conoce y lo que se desconoce hasta ese instante).

3.9.1. Grafo dirigido acíclico

La representación mediante un grafo muestra el conocimiento absoluto de la relación en un determinado instante del proceso de ejecución y cuya completa construcción hasta obtener un grafo lineal es la tarea a realizar. El diagrama de *Hasse*, es un grafo con la mínima relación cuya clausura transitiva representa el orden parcial conocido.

La primera imagen válida que se propone utiliza un grafo dirigido y acíclico. La representación es más gráfica si cabe que la que se realiza mediante la matriz de incidencia, aunque más compleja de manejar si se pretende realizar cálculos con ella.

Dada la naturaleza intrínseca del problema, el grafo ha de ser dirigido para que se refleje la relación establecida entre cada par de elementos, y por otro lado debe ser acíclico. De no ser así se podría entrar en un bucle infinito irresoluble. Tal y como se comentó en el apartado 3.5.1, carece de sentido dentro de la relación de orden que dos elementos en un instante t presenten una relación y posteriormente, en un instante t', la contraria. Esta situación se refleja en el grafo mediante un bucle el cual impide obtener una solución al problema.

El grafo dirigido acíclico está formado por un conjunto de vértices V y un conjunto de aristas A. Cada uno de los vértices $v \in V$, se corresponde con cada elemento del conjunto de entrada $\mathcal{E}$. Por tanto el tamaño del conjunto V es igual al número de elementos a tratar por el meta-algoritmo, e igual al número de elementos presentes en $\mathcal{E}$.

Cada arista $a \in A$ representa la relación de orden entre dos elementos del conjunto V. Supongamos que se eligen dos elementos i, $j \in \mathcal{E}$ que se corresponden con los vértices $v[i]$, $v[j] \in V$. La arista $a \in A$ se representa de la siguiente forma: [10]

- Si $elem_i < elem_j$, la arista a tiene inicio en $v[i]$ y final en $v[j]$.
- Si por el contrario $elem_i > elem_j$, la arista a tiene su inicio en $v[j]$ y finaliza en $v[i]$.

3.9.1.1. Ejemplo de construcción de grafo dirigido acíclico

Con este planteamiento gráfico, al inicio del problema lo que se tiene es un conjunto de vértices no vacío,[11] formado por un número igual que la cardinalidad del conjunto $\mathcal{E}$ compuesto por los elementos a tratar ($\#V = \#\mathcal{E}$).

[10] Por convención, la igualdad de elementos se incluye dentro del caso "menor que".

[11] Siempre que el conjunto de entrada sea $\neq \emptyset$. Si el conjunto de entrada es $\emptyset$, el meta-algoritmo finaliza devolviendo idéntico resultado, $\emptyset$.

El conjunto de aristas A es igual al conjunto vacío ($\emptyset$), ya que al inicio no se conoce ninguna relación entre pares de elementos ($\#A = 0$). Por tanto en la situación inicial el grafo (se muestra en la figura 3.2) está compuesto por un conjunto de "islas" sin relación alguna, siendo el número de estas igual al de elementos que se pretende ordenar.

Supongamos que se desea ordenar el conjunto de elementos $\{a, b, c, d, e, f, g, h\}$. La situación inicial se encuentra representada en la figura 3.2.

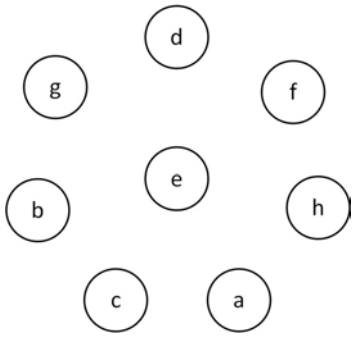


FIGURA 3.2: Grafo dirigido acíclico: situación inicial

A medida que se avanza hacia un estado final los vértices del conjunto V se relacionan mediante las aristas del conjunto A. En el proceso de ejecución el número de vértices V permanece invariable, mientras que el número de aristas del conjunto A ha de ir creciendo hasta encontrar la solución.

En un cualquier punto intermedio del proceso (como por ejemplo el representado en la figura 3.3) la cardinalidad del conjunto V permanece invariable con respecto a su valor inicial ($\#V{=}\#\mathcal{E}$), y el conjunto de aristas tiene una cardinalidad *superior* a la inicial (salvo que al inicio $\#V{=}0$ o $\#V{=}1$, con lo que $\#A{=}0$ para toda la ejecución). El número de aristas, o lo que es lo mismo $\#A$, debe estar comprendido durante toda la ejecución entre los valores $[0, \#V-1]$.

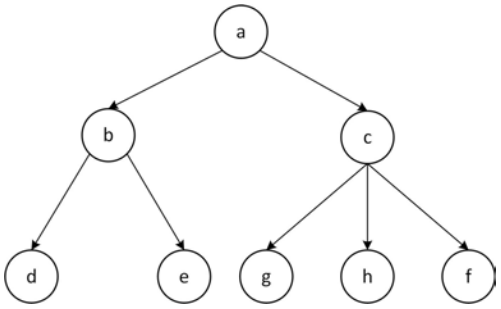


FIGURA 3.3: Grafo que representa una situación intermedia

Con esta representación, gráficamente se concluye que el proceso alcanza un estado final y por tanto encuentra una relación de orden total cuando se cumplan las siguientes premisas (siempre que $\#\mathcal{E} > 1$):[12]

- El número de aristas del grafo sea igual a $\#V - 1$.
- No exista vértice $v \in V$ que sea "isla": $\forall\ v \in V,\ G(e) + G(s) \geq 1$.
- Los grados del nodo raíz son $G(e) = 0$ y $G(s) = 1$, y los del único nodo hoja $G(e) = 1$ y $G(s) = 0$.
- Para el resto de nodos, $G(e) = G(s) = 1$.

La imagen de la finalización del proceso (representada en la figura 3.4) es similar a un grafo cuya representación es una "cadena de eslabones".

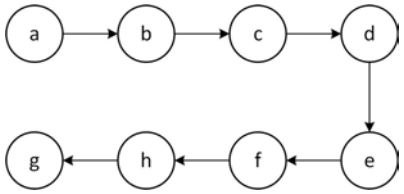


FIGURA 3.4: Grafo que representa un estado final de orden total

Se tratan como particularidad aquellos casos en que $\#\mathcal{E} \leq 1$, es decir, cuando el conjunto de elementos a tratar esté vacío o se disponga de un sólo elemento. En estos casos:

- Si $\#\mathcal{E}=0$ se alcanza la solución sin necesidad de realizar tarea alguna, devolviendo directamente como resultado $V=A=\emptyset$.

[12] $G(e)$=Grado de entrada del vértice y $G(s)$=grado de salida del vértice.

- Si $\#\mathcal{E}$=1, de forma similar se devuelve como solución el propio conjunto V, siendo A=$\emptyset$.

3.9.2. Matriz de incidencia

Con el fin de simplificar la presentación y dado que se trata de una representación complementaria que no se usa más que como ilustración, la matriz de incidencia se muestra con una serie de convenciones. Se puede utilizar como modelo mental y eventualmente como apoyatura para el razonamiento.

La representación matricial trata de reconstruir la posición que ocupa cada elemento de la matriz a partir de la acción elemental de comparar dos elementos. El resultado de la comparación permite además deducir la posición de otros elementos al aplicar la propiedad transitiva de la relación. El objetivo consiste en completar todos los valores representativos de dicha matriz.

La visión detallada se describe como una matriz cuadrada de orden n, siendo n el número de elementos a ordenar del conjunto $\mathcal{E}$ (se muestra en la figura 3.5). Los valores de los elementos de la matriz representan el conocimiento que se tiene en un momento determinado sobre el orden parcial entre los elementos del conjunto $\mathcal{E}$. A continuación se definen los posibles valores de los elementos de la matriz, así como algunas características especiales en la búsqueda de la solución que permiten reducir el escenario a la mínima expresión.

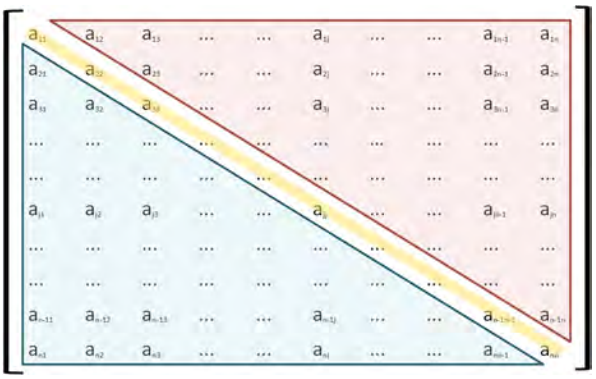


FIGURA 3.5: Matriz de incidencia

Dada una matriz A de orden n y dos posiciones cualquiera de la matriz i, j cumpliéndose que $0 < i, j \leq n$, el valor del elemento $A[i, j]$ en un instante determinado representa si se conoce o no la relación entre los elementos que ocupan las posiciones i, j. De conocerse, indica también qué elemento es el

precedente. Concretando los posibles valores que se presentan, la matriz está rellena de la siguiente forma:

- Si el valor en $A[i,j]$=1, entonces $e_i < e_j$.
- Si el valor en $A[i,j]$=0, se desconoce la relación entre los elementos i, j.
- Si el valor en $A[i,j]$=-1 entonces $e_i > e_j$.

Por una cuestión de comodidad de lectura cuya utilidad se apreciará más adelante, se opta por una forma normal: como se trata de un orden parcial, cualquier reindexado de los elementos compatible con ese orden parcial (obtenido por ordenación topológica) resulta en que:

- Si $e_i < e_j \Rightarrow i < j$, con lo que la matriz de incidencia tiene los *unos* (1) en el triángulo superior y los *menos unos* (-1) en el inferior

Al disponer de una matriz antisimétrica, es suficiente trabajar con los valores situados en una de las zonas de la diagonal: en este caso se utilizan únicamente los valores situados por encima de la diagonal (valores igual a *cero* o *uno*). Con esta reducción los valores de los elementos de la matriz se encuentran en "binario", puesto que sólo se dispone de 2 posibles alternativas:

- Valor 0: Indica que no se conoce todavía relación alguna entre los elementos i, j.
- Valor 1: Si se ha respondido alguna pregunta de comparación y se conoce la relación de orden entre los elementos i, j.

Así definida la matriz, las situaciones inicial y final se corresponden con las imágenes de la figura 3.6.

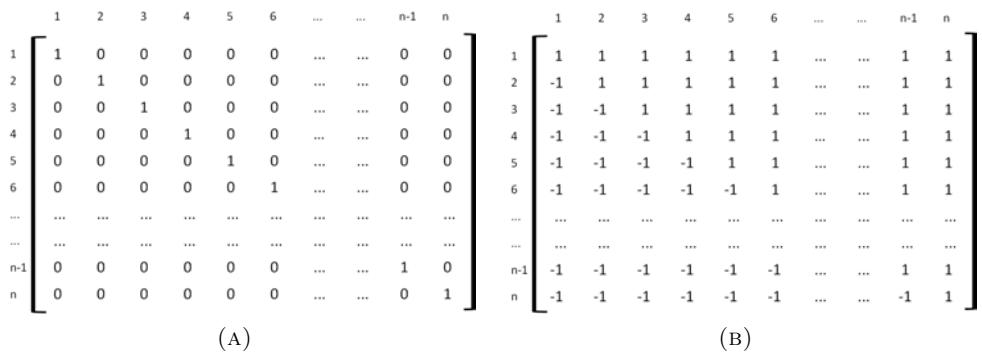


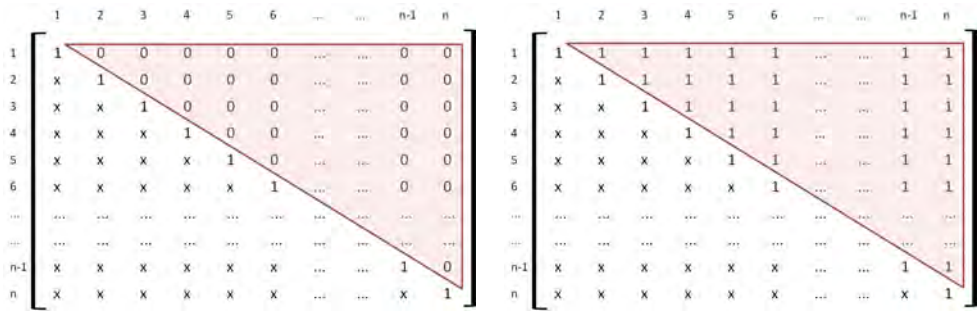
FIGURA 3.6: (A) Matriz estado inicial (B) Matriz estado final

Para tratar de reducir la carga de trabajo que supone utilizar los $n \times n$ elementos de la matriz, se diferencian aquellos elementos realmente interesantes de aquellos que carecen de valor. Es suficiente trabajar con los elementos que se encuentran por encima de la diagonal de la matriz, ya que el resto de relaciones o bien carecen de sentido en la búsqueda de la solución (elementos situados en la diagonal), o bien son deducidos de la información contenida en la parte superior de esta.

Con la anterior definición de valores matriciales y aprovechando la propiedad de reflexividad, se garantiza que toda la información sobre las relaciones entre elementos del conjunto de entrada está almacenada en el triángulo superior de la matriz por los siguientes motivos:

- Todos los elementos situados en la diagonal de la matriz donde $i = j$, son irrelevantes, puesto que al tratarse de una relación de orden la reflexividad impone que cada elemento esté relacionado consigo mismo.
- Si el valor que se encuentra en la posición de la matriz $A[i, j]$ es un 1 y no se está analizando un elemento de la diagonal de la matriz (por tanto $i \neq j$), por la propiedad reflexiva del problema que se trata el valor situado en la posición de la matriz $A[j, i]$ debe ser un -1, ya que si se conoce la relación de orden entre el elemento de la fila i y la columna j, lógicamente se conoce que la relación existente entre el elemento j y el elemento i es inversa.

En la imagen de la figura 3.7 se muestran los estados inicial y final de la matriz únicamente con aquellos elementos que son relevantes.



(A) (B)

FIGURA 3.7: Estados inicial y final: sólo elementos necesarios

Todos los elementos marcados con "x" no son necesarios. El resto de elementos (sombreados de color rojizo) son suficientes para conocer en cada

momento de la ejecución la situación en que se encuentra el proceso, facilitando la elección de la siguiente comparación.

De cualquier forma se hace necesario realizar una tarea más. El orden que ocupa cada elemento en la matriz no influye cuando se representa un instante determinado del proceso de ordenación, por tanto filas y columnas son intercambiables (manteniendo, eso sí, idéntica posición relativa de cada elemento en fila y columna: si el elemento z se representa en la fila j, debe estar representado igualmente en la columna j). La información contenida entre matrices que sólo se diferencian en el orden que ocupan los elementos en filas y columnas, es exactamente la misma.

En cada momento se dispone de un subconjunto de las permutaciones compatibles con el orden parcial conocido hasta entonces. Cada comparación posterior es inútil si no se reduce ese conjunto y es más adecuada (al menos localmente) en cuanto divida igualmente el conjunto de posibilidades.

Inicialmente las posibilidades son todas las permutaciones de n elementos. Al final del proceso se reduce a una sola permutación.

Supongamos que en un instante determinado se conoce la relación de orden parcial representada por el grafo de la figura 3.8.

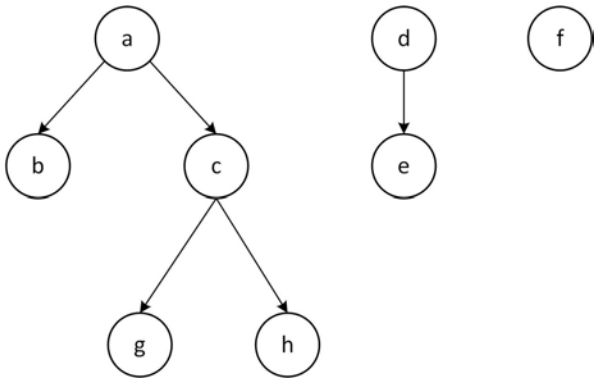
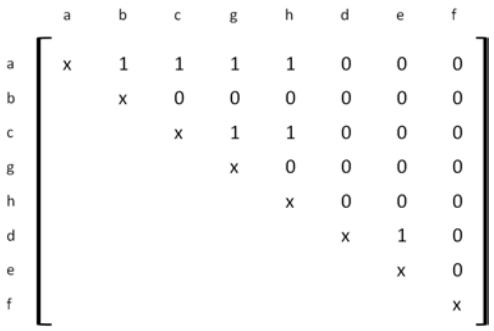


FIGURA 3.8: Ejemplo de representación mediante grafo dirigido acíclico

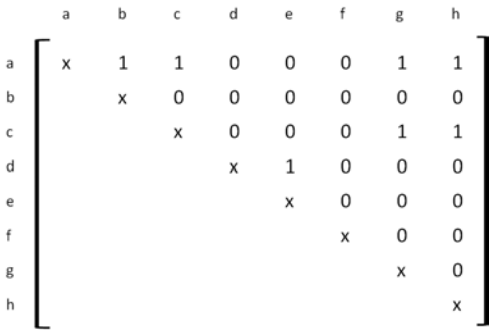
Las matrices de incidencia de las figuras 3.9, 3.10 y 3.11 representan idéntico momento en el proceso de ordenación, siendo la información que contienen todas ellas equivalente.



	a	b	c	g	h	d	e	f
a	x	1	1	1	1	0	0	0
b		x	0	0	0	0	0	0
c			x	1	1	0	0	0
g				x	0	0	0	0
h					x	0	0	0
d						x	1	0
e							x	0
f								x

FIGURA 3.9: Primera representación matricial de un orden parcial

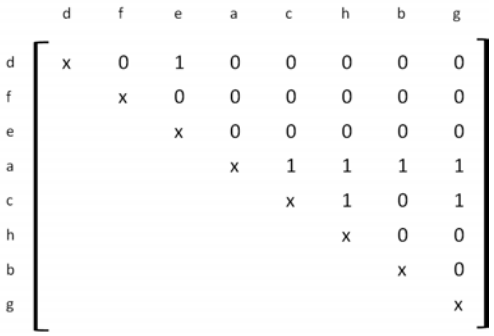
Otro orden es:



	a	b	c	d	e	f	g	h
a	x	1	1	0	0	0	1	1
b		x	0	0	0	0	0	0
c			x	0	0	0	1	1
d				x	1	0	0	0
e					x	0	0	0
f						x	0	0
g							x	0
h								x

FIGURA 3.10: Segunda representación matricial de un orden parcial

Y aún otro orden distinto pero equivalente:



	d	f	e	a	c	h	b	g
d	x	0	1	0	0	0	0	0
f		x	0	0	0	0	0	0
e			x	0	0	0	0	0
a				x	1	1	1	1
c					x	1	0	1
h						x	0	0
b							x	0
g								x

FIGURA 3.11: Tercera representación matricial de un orden parcial

Dado que en esta representación el orden en que se encuentran los elementos en la matriz es irrelevante, para poder trabajar únicamente con los que están situados por encima de la diagonal de la matriz se requiere mantener el orden de filas (y por tanto columnas) según la relación de orden parcial conocida.

Realizando la tarea bajo esta premisa, al avanzar hacia un estado final los elementos menores deben ocupar las primeras posiciones de filas y columnas de la matriz. Si no se hace esta salvedad, es muy posible que al comparar dos elementos el valor 1 que marca su relación deba colocarse en la zona que se encuentra por debajo de la diagonal de la matriz: es algo que se soluciona fácilmente llevando a cabo una reordenación de los elementos.

Supongamos nuevamente el orden parcial representado por el grafo de la imagen 3.8, junto con la tercera representación matricial equivalente que se encuentra en la figura 3.11. En dicha representación matricial el primer elemento que aparece es el d. Si se lleva a cabo la comparación entre los elementos d y a con el resultado de $a < d$, se hace necesario "recolocar" la posición matricial del elemento a, que ha de ser anterior a la del elemento d. En esta ocasión el elemento a pasa a ocupar la primera fila y columna de la matriz, siendo desplazados una posición el resto de elementos hasta la posición que ocupaba dicho elemento a. De no hacerse así, el elemento de la matriz en que se debe colocar el valor 1 tras realizar la comparación, se encuentra situado en la zona inferior de la diagonal de la matriz. Con esta alteración irrelevante en el orden de fila y columna, al no influir dicho cambio en la representación como se ha explicado con anterioridad, se incluye toda la nueva información tras la comparación en la matriz de incidencia sin pérdida alguna de la generalidad. La nueva matriz se corresponde con la mostrada en la figura 3.12.

	a	d	f	e	c	h	b	g
a	x	1	0	0	1	1	1	1
d		x	0	1	0	0	0	0
f			x	0	0	0	0	0
e				x	0	0	0	0
c					x	1	0	1
h						x	0	0
b							x	0
g								x

FIGURA 3.12: Nueva representación matricial con $a < d$

En definitiva, con la definición de los valores matriciales anteriormente descritos, aprovechando la reflexividad proporcionada por la operación de comparación de elementos y junto a esa irrelevante pero necesaria reorganización de las filas y columnas (elementos representados), se garantiza la comodidad de trabajar únicamente con los elementos situados sobre la zona superior de la diagonal de la matriz de incidencia.

3.9.2.1. Ejemplo de representación mediante la matriz de incidencia

A continuación se muestra mediante pequeños ejemplos la utilización de la matriz de incidencia en diferentes momentos del proceso de ordenación.

Como apunte inicial, señalar que en el momento de llevar a la práctica la representación matricial descrita no sólo como herramienta que permite representar un instante determinado en el proceso de ordenación, sino también como herramienta matemática que posibilita aprovechar la propiedad de *transitividad* de la operación de comparación, el lenguaje de programación elegido ha de disponer entre sus tipos de datos de matrices de tamaño variable o, de alguna manera su simulación mediante un *TAD*.

La explicación a esta afirmación es sencilla: al inicio del proceso es posible que no se disponga en la entrada de todos los elementos a tratar. Imponer en cualquier momento de la ejecución la restricción de tener que esperar la llegada de todos los elementos (como por ejemplo impone *Quicksort*) no hace sino condicionar, con detalles ajenos a la naturaleza del problema abordado la solución al mismo.

Supongamos que el conjunto de elementos $\mathcal{E}$ que se desea ordenar es de tamaño $n = 8$ y está formado por los elementos $\{f, g, c, a, e, b, c, d\}$.

En el instante inicial (figura 3.13) la matriz A se ve reducida a la matriz diagonal, puesto que la única relación que se conoce se corresponde con la propiedad de reflexividad entre todos los elementos del conjunto, que se relacionan consigo mismos y, lógicamente, el orden en filas y columnas de los elementos en la matriz es irrelevante.

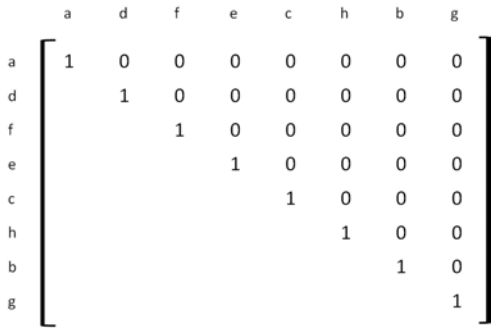


FIGURA 3.13: Matriz de incidencia en el inicio del proceso

Con la definición que se ha realizado la diagonal de la matriz se encuentra puesta a 1 durante toda la ejecución, por lo que es ignorada junto al resto de elementos situados bajo la propia diagonal.

En la figura 3.14 se muestra un instante determinado en el proceso de ordenación, en el que se conoce la relación entre un determinado conjunto de elementos y la del resto es desconocida.

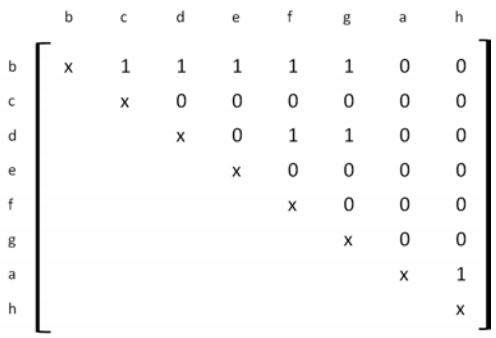
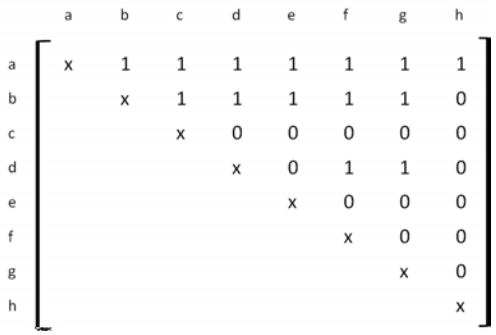


FIGURA 3.14: Matriz de incidencia en un instante t del proceso

En la matriz de incidencia, aquellos elementos cuyo valor en la relación está puesto a 0, se traduce en un desconocimiento de la relación entre ambos elementos, lo cual indica que son posibles candidatos a realizar una comparación. Supongamos que se realiza la comparación entre los elementos a y b obteniéndose el resultado $a < b$. A continuación se procede a trasladar dicha información (y la derivada por transitividad) a la matriz de incidencia, la cual pasa a tener el aspecto que se muestra en la figura 3.15.



	a	b	c	d	e	f	g	h
a	x	1	1	1	1	1	1	1
b		x	1	1	1	1	1	0
c			x	0	0	0	0	0
d				x	0	1	1	0
e					x	0	0	0
f						x	0	0
g							x	0
h								x

FIGURA 3.15: Matriz de incidencia en un instante $t' = t+1$

Al estar situado el elemento b en la primera fila/columna de la matriz y resultar ganador de la comparación el elemento a, este pasa a ocupar una posición anterior a la de b, por lo que es el elemento a quien ocupa ahora la primera fila/columna de la matriz, y el resto de elementos desde b hasta g se desplazan un lugar de la posición que ocupaban.

La información incluida en la matriz de incidencia no se limita únicamente a la obtenida de la comparación de los elementos a y b. Además se conocía que el elemento b era menor que todos los elementos del conjunto $\{c, d, e, f, g\}$. Para añadir esta información conocida por transitividad a la matriz, se ha de incorporar no sólo la información de que $a < b$, sino también que es menor que todos esos elementos.

Por la forma en que se define la matriz de incidencia resulta muy útil para seleccionar los candidatos con los que continuar el proceso de comparación, evitando la repetición de preguntas cuya respuesta se conoce de antemano. Aquellos elementos de la matriz cuyo valor sea igual a *cero* se encuentran entre los posibles candidatos para la siguiente comparación.

El proceso de búsqueda de la relación de orden total finaliza cuando la mitad superior de la matriz se encuentra *completa,* lo que se traduce en que todos los valores por encima de la diagonal son iguales a 1[13] (situación que se muestra en la figura 3.16).

El orden final de los elementos coincide con la posición relativa de fila/columna que ocupa cada elemento en la matriz de incidencia.

[13]Realmente al finalizar disponen de valor absoluto *uno* todos los elementos de la matriz. Los situados en la diagonal desde el inicio. Los situados por debajo de la diagonal, por reflexividad, desde el momento en que disponga de dicho valor su elemento simétrico situado encima de la diagonal.

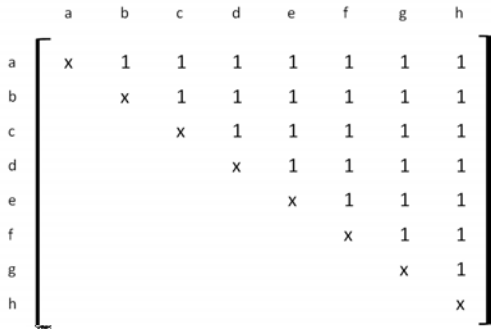


FIGURA 3.16: Matriz de incidencia al finalizar el proceso de ordenación

3.9.2.2. Propiedades de la representación mediante matriz de incidencia

La representación mediante la matriz de incidencia presenta una serie de propiedades que permiten conocer visualmente, una determinada posición de los elementos tratados.

Si se elige un elemento $x \in \mathcal{E}$ representado en la matriz de incidencia cuya fila (valores que se encuentran a la derecha de la diagonal de la matriz) sea completa (como se muestra en la figura 3.17), se deduce que dicho elemento x es menor que todos los elementos que aparecen posteriormente a él en la representación matricial. Por este motivo la posición definitiva de dicho elemento no es, bajo ninguna circunstancia, posterior a la que ocupa en ese preciso instante. El elemento x puede finalizar en una posición anterior a la que ocupa (porque es posible que todavía no haya sido comparado con los elementos que actualmente le preceden) pero nunca posterior.

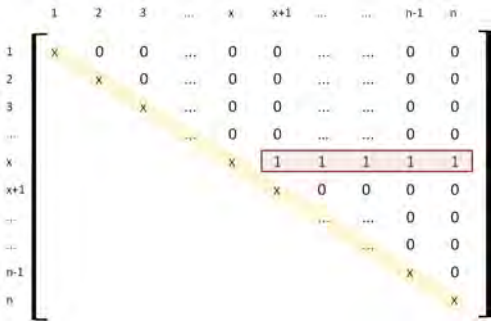


FIGURA 3.17: El elemento x no puede ocupar una posición posterior

Análogamente, si un elemento $y \in \mathcal{E}$ posee en su columna (por encima de la diagonal) todos los valores de la matriz puestos a *uno* (se muestra la imagen en la figura 3.18), dicho elemento es mayor que todos los que le preceden en la representación matricial, por lo que no ocupará de manera alguna una posición "anterior" a la que ocupa en este momento. De forma análoga a la situación anterior, el elemento y puede ocupar posiciones posteriores a la actual pero nunca anteriores.

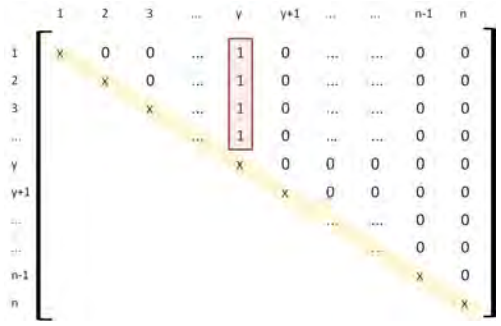


FIGURA 3.18: El elemento y no puede ocupar una posición anterior

Si se unen ambas propiedades de la representación en un sólo elemento y se selecciona de la matriz un elemento $z \in \mathcal{E}$, cuyos valores en la fila a la derecha de la diagonal y en su columna por encima de la diagonal se encuentran todos puestos a *uno*, se deduce que dicho elemento ocupa la posición definitiva que le corresponde en el orden final: no puede ni ascender ni descender en el orden respecto al resto de elementos a tratar, por lo que se garantiza que su posición relativa en el orden final es la definitiva.

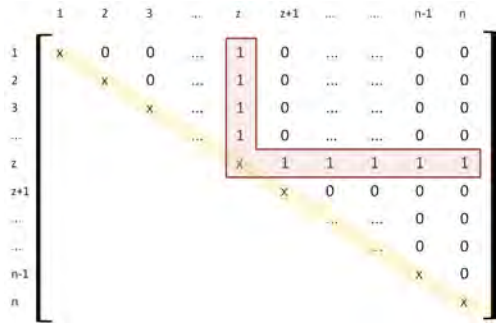


FIGURA 3.19: El elemento z ocupa su posición definitiva

Un claro ejemplo de esta propiedad se encuentra reflejado en la ejecución del algoritmo *Quicksort* (Apéndice J). Cada vez que dicho algoritmo selecciona un elemento como "pivote", divide el conjunto de elementos a tratar en dos subconjuntos. Realmente lo que se produce es la fijación de la posición del elemento "pivote". En el proceso, el pivote es comparado con los n–1 elementos restantes, lo que se traduce en que su posición queda fijada, reflejándose en la matriz de incidencia como una "L" formada por *unos* tal y como se muestra en la figura 3.19.

3.10. Transitividad

La propiedad de *transitividad* permite sacar el máximo provecho de las comparaciones que se realizan en la búsqueda del orden total. En ocasiones bajo determinadas circunstancias, cuando se comparan dos elementos del conjunto $\mathcal{E}$, la información obtenida no sólo arroja información sobre el orden de ambos, sino que aplicando la propiedad de transitividad es posible obtener la relación con otros elementos del conjunto. Esta información es desperdiciada en muchas ocasiones, haciendo que en un futuro no lejano se tengan que realizar preguntas cuya respuesta podría haber sido deducida.

Si entre los elementos del conjunto $\{a, b, c\} \in \mathcal{E}$ es conocida la información de que $a < b$ y $b < c$, es innecesario preguntar en un futuro si $a < c$, puesto que aun no habiéndose realizado la comparación, la respuesta es deducible por el resto de información que se dispone.

Tal y como se describe en el apartado 3.9.2, la obtención del orden total se refleja en la matriz de incidencia cuando todos los elementos por encima de su diagonal dispongan de un valor igual a 1. Por tanto avanzar hacia la solución, consiste en obtener en cada paso el mayor número de *unos* posible en la matriz.

Cuando se realiza la comparación entre dos elementos es deseable no sólo reflejar la información obtenida de forma directa tras la comparación, sino también aprovechar aquella que se obtiene de forma indirecta, con el fin de evitar en futuras comparaciones preguntas cuya respuesta era deducible. Representada la información directa obtenida por comparación de dos elementos en la matriz de incidencia y, una vez realizada la clausura transitiva a esta, los *ceros* en la matriz de incidencia representan posibles comparaciones útiles, mientras que los *unos* representan las preguntas superfluas.

En los siguientes ejemplos se muestran dos algoritmos de ordenación en sus vertientes opuestas respecto al concepto de transitividad: en primer lugar un algoritmo que se olvida del concepto y lo desaprovecha: *Burbuja*. En

segundo lugar uno de los múltiples algoritmos que sí lo aprovecha: *Binario*.[14]

3.10.1. Ejemplo sin transitividad: *Burbuja*

Hay muchas ocasiones en las que la mejor forma de describir un concepto radica en utilizar un ejemplo que carezca de él. Esto es lo que sucede con la transitividad y el algoritmo *Burbuja*.

La eficacia de dicho algoritmo es indiscutible, pero su eficiencia es, cuanto menos, cuestionable. Los motivos de dicha ineficiencia parecen estar claros:

- Inserción secuencial frente a inserción binaria.
- Realización de comparaciones evitables por dos motivos: o bien ya habían sido realizadas (repeticiones), o bien el resultado se podía haber deducido por transitividad.
- En parte es consecuencia de la limitación a comparaciones entre elementos adyacentes (lo que representa, a su vez, su posible ventaja marginal).

En cada pasada del algoritmo, cada elemento es comparado con los $n-1$ elementos restantes hasta ocupar la posición que le corresponde, por lo que cada pasada del algoritmo coloca de forma definitiva un elemento: ocupa la menor posición el elemento vencedor de la última comparación de cada pasada. Sin embargo en la misma pasada es muy probable que existan elementos que hayan ganado comparaciones parciales, cuyos resultados tras la comparación son despreciados por el algoritmo debiendo repetir comparaciones en futuras pasadas, con el consiguiente desperdicio de tiempo de ejecución.

Supongamos un elemento a_i que va ganando varias comparaciones en el proceso de ejecución hasta tropezar con un b_i que resulta ser menor que él, y menor que todos los elementos. De forma efectiva en esa pasada el elemento b_i pasa a ocupar la posición definitiva en el orden final, pero de toda la información de las "victorias" obtenidas por a_i no queda rastro alguno. En la siguiente pasada a_i vuelve a ser comparado con elementos cuyo resultado de la operación ya es conocido. Este hecho se produce porque el algoritmo desaprovecha la información que se obtiene por transitividad.

3.10.2. Ejemplo de transitividad en *Binario*

El algoritmo clásico de ordenación mediante la utilización de un árbol binario (se encuentra en el apéndice M) se muestra como ejemplo del aprovechamiento de la información que se obtiene por transitividad.

[14] Los algoritmos *Quicksort* y *Fusión* también aprovechan la información obtenida por transitividad.

La mejor forma de mostrarlo es realizar un análisis sobre la forma de trabajar de dicho algoritmo, utilizando las formas de representación antes descritas mediante un ejemplo práctico.

Supongamos que el conjunto de entrada $\mathcal{E}$ está formado por los elementos $\{g, c, p, d, j, r, b, a, s, e\}$ correspondiéndose su posición con el orden de aparición, y el orden buscado se corresponde con el lexicográfico. En un instante intermedio del proceso de ejecución, cuando sólo falten por analizar los elementos $\{a, s, e\}$, el grafo de la situación se corresponde con el mostrado en la figura 3.20.

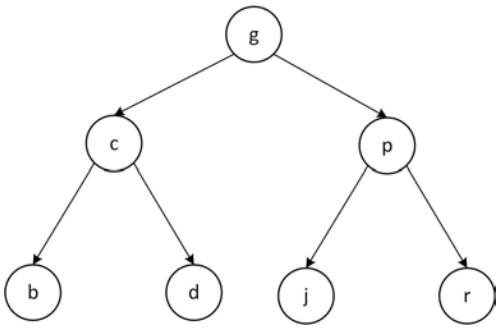


FIGURA 3.20: Ejecución algoritmo *Binario*: grafo situación intermedia

Y la matriz de incidencia correspondiente con dicha situación se encuentra reflejada en la figura 3.21.

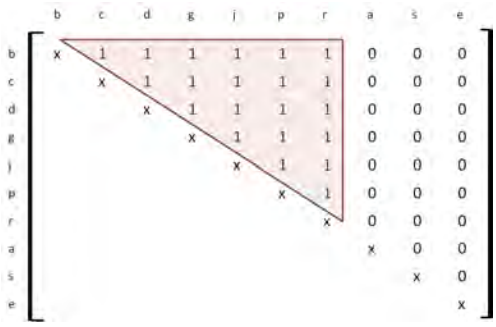


FIGURA 3.21: Ejecución algoritmo *Binario*: matriz situación intermedia

Los elementos sombreados de color rojizo representan el universo conocido hasta este momento, y del resto de elementos de la matriz marcados con 0 no se dispone de ningún tipo de información. A continuación se presenta

en la entrada el elemento a, que tal y como establece la ejecución del algoritmo *Binario* debe ser comparado con la raíz del árbol (elemento g). Tras la comparación se obtiene nueva información: $a < g$. Se traslada la información obtenida a la matriz, que ha de reflejar un valor igual a 1 en la posición que relaciona a ambos elementos. Pero además, por transitividad, el elemento a es menor que todos aquellos elementos de los que a su vez g también es menor. La matriz de incidencia tras la comparación se encuentra representada en la figura 3.22.

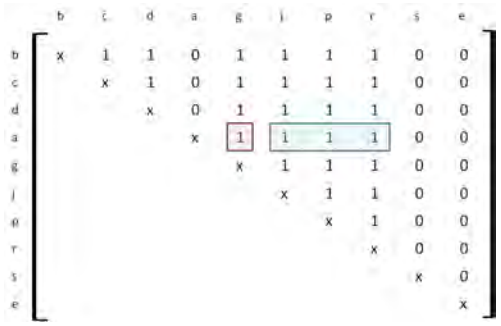


FIGURA 3.22: Ejecución algoritmo *Binario*: añadida información $a < g$

La posición de la matriz sombreada de color rojizo, representa la información obtenida tras la comparación directa de ambos elementos. Las posiciones de la matriz sombreadas de color azul, representan la información obtenida por transitividad.

El proceso continúa con la comparación de los elementos a y c. Tras obtener el resultado de la comparación ($a < c$) se traslada toda la información obtenida a la matriz, cambiando esta a la situación mostrada en la figura 3.23.

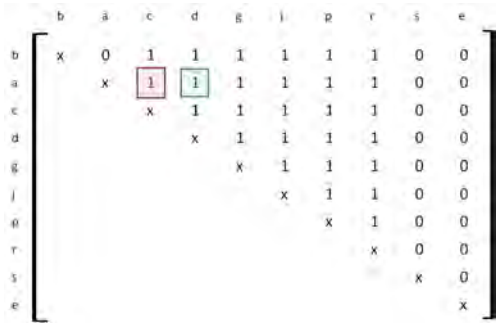


FIGURA 3.23: Ejecución algoritmo *Binario*: añadida información $a < c$

De nuevo los colores rojizo y azul representan la información obtenida por comparación directa (rojo) y por transitividad (azul).

Se llega a la última comparación: sólo resta comparar los elementos a y b. La información directa $a < b$ se traslada a la matriz, y no se traslada ningún tipo de información por transitividad ya que el elemento b no dispone de *hijos por la derecha.* Por tanto la única información obtenida es por comparación directa: el elemento a ocupa su posición final en este orden parcial y los elementos de la matriz presentan los valores dispuestos en la figura 3.24.

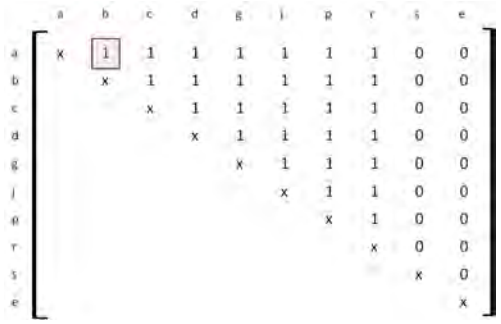


FIGURA 3.24: Ejecución algoritmo *Binario*: añadida información $a < b$

Las situaciones mostradas son un claro ejemplo de como el conocimiento que nos acerca a la solución del problema no proviene exclusivamente de la información obtenida por comparación directa entre elementos, sino que además se puede obtener más información por transitividad, lo cual evita realizar futuras comparaciones cuya respuesta es deducible de comparaciones precedentes.

Capítulo 4

Meta-algoritmo descendente

No hay rama de la matemática, por abstracta que sea, que no pueda aplicarse algún día a los fenómenos del mundo real.

Nikolay Lobachevsky

4.1. Introducción

El concepto de meta-algoritmo de ordenación descrito en el capítulo 3, junto con los componentes necesarios para su ejecución, permiten construir dos meta-algoritmos que se diferencian únicamente en las mínimas estructuras de control necesarias.

Este capítulo se centra en el estudio de lo que se ha denominado *meta-algoritmo descendente.* A la expresión meta-algoritmo ya detallada, se añade ahora el calificativo de *descendente.* Utilizando la máxima de "divide y vencerás" para la resolución de problemas y haciendo una analogía con las metodologías incluidas en dicha máxima, el problema es abordado mediante una metodología de división de problemas, de refinamientos sucesivos, aplicando una filosofía *top-down.*

Analizando minuciosamente el funcionamiento de los algoritmos clásicos *Quicksort* (Apéndice J) y *Binario* (Apéndice M), se observa que la metodología utilizada para llegar a la relación de orden total se basa en divisiones del conjunto de elementos a tratar, realizando por tanto la tarea encomendada por división de esta en subtareas de menor tamaño.

Si se aplica el concepto de *abstracción* y obviamos la estructura de control en ambos algoritmos, se encuentran una serie de similitudes que sugiere

una estructura común más profunda subyacente. Los puntos en común, salvo por matices de implementación final, permiten afirmar que ambos algoritmos trabajan de idéntica forma: en cada iteración uno de los elementos del conjunto a tratar pasa a ocupar su posición definitiva respecto al resto de elementos, que en las siguientes iteraciones ocupan el lugar que les corresponde en la relación de orden final.

En el caso del algoritmo *Quicksort*, dicho elemento es el denominado "pivote", y es el encargado de marcar el valor por el cual se ha de realizar la división de cada conjunto de elementos. En el caso del algoritmo *Binario* es la propia construcción del árbol de búsqueda quien se encarga de proporcionar la citada división.

Este proceso inductivo de unificación (o reducción) da lugar a un meta-algoritmo, que prescinde en la mayor medida posible de las características individuales de carácter particular de cada implementación, conservando únicamente las cualidades comunes.

A continuación se presenta un formalismo unificador, del que se comprueba su corrección en primera instancia de manera informal y posteriormente utilizando el formalismo necesario para su demostración.

Por un procedimiento deductivo se derivan ambos algoritmos a base de particularizar la estructura de datos y la estructura de control, al imponer algunas restricciones que se espera que hagan al resultado susceptible de optimizaciones de implementación.

Además, se muestra como la eliminación de la estructura de control permite que más algoritmos de los considerados clásicos tengan cabida dentro del meta-algoritmo descendente, siendo considerados todos ellos como pertenecientes a la misma familia de algoritmos de ordenación al realizar su trabajo de forma similar.

4.2. Cómo empezó todo

Observando la dinámica de los algoritmos *Quicksort* y *Binario*, prestando atención a las acciones esenciales y sin considerar los aspectos dependientes de la implementación se detecta lo siguiente:

Se toma un elemento y todos los demás son comparados con él, dividiéndose el conjunto inicial en dos subconjuntos, el de los elementos menores y el de los elementos mayores que el seleccionado.

En el algoritmo *Quicksort* esto se hace inicialmente como primera acción, mientras que en *Binario* estas comparaciones vienen repartidas en el

tiempo, pero todos los elementos son comparados con el primero escogido, derivándose hacia el subárbol derecho o el izquierdo según sea el resultado de la comparación.

Este proceso se repite con cada uno de los conjuntos obtenidos, y se sigue realizando de forma recursiva hasta encontrar la relación de orden final para el conjunto de elementos inicial.

En resumen, ambos algoritmos realizan las mismas acciones esenciales aunque las distribuyen en el tiempo de forma diferente.

El meta-algoritmo es un formalismo que representa esas acciones, definiendo las mínimas restricciones de secuencia en la ejecución dinámica de su tarea y sin prestar atención especial a las estructuras de datos más allá de unas condiciones mínimas.

Tanto *Quicksort* como *Binario* responden a la idea de insertar un elemento en una cadena, pero lo hacen según dinámicas distintas.

En *Binario* se observa esta acción más claramente: el árbol implementa la cadena de elementos, y su estructura incluye el proceso de inserción que, si el árbol es razonablemente equilibrado, se aproxima o coincide con una inserción binaria (por otro lado la más eficiente, ver apéndice B).

4.3. Definiciones básicas

El meta-algoritmo descendente, a partir del conjunto de entrada de elementos, trata de responder a la siguiente pregunta: *dado un elemento x del conjunto $\mathcal{E}$, ¿cuál es la posición que este ocupa?.*

Aunque la pregunta pudiera parecer un poco extraña, es la que se ajusta a la forma de resolver el problema por parte de este meta-algoritmo. Una vez que el meta-algoritmo descendente selecciona un elemento su posición queda fijada para todo el proceso: por tanto se ha encontrado su posición final. El siguiente paso consiste en repetir la tarea para el resto del conjunto de elementos hasta encontrar la posición definitiva para todos ellos.

La descripción del procedimiento completo consta de dos fases:

1. La primera fase se corresponde con la fase de ordenación propiamente dicha, alcanzando un estado final que permite con facilidad la realización de la siguiente fase.

2. La segunda fase de recuperación de los elementos en el orden deseado.

La estructura de los estados finales es un árbol binario que contiene en cada nodo un elemento del conjunto de entrada, y el resto de componentes

del estado (salvo la información estructural del árbol) no intervienen en la segunda fase.

La realización de la fase de recuperación consiste en recorrer el árbol creado en *inorden*[1] tomando el elemento que se encuentra en cada nodo.

Dado que la fase de recuperación no presenta un interés especial, el resto del capítulo se dedicada íntegramente a tratar la fase de ordenación.

4.4. Versión intuitiva

La imagen intuitiva en un instante concreto de la ejecución del meta-algoritmo descendente es la de un árbol binario de procesos: por tanto, cada nodo de dicho árbol se corresponde con un proceso en ejecución. A su vez cada proceso en ejecución dispone de un conjunto de entrada con los elementos a procesar y que, al ser creado cada proceso, adquiere un elemento distinguido (al que por analogía con el algoritmo clásico *Quicksort* denominaremos *pivote*) con el que compara los elementos que a continuación procesa. Según el resultado que se obtenga en la comparación transfiere el elemento comparado al proceso hijo derecho o izquierdo, creando previamente el *proceso hijo* en caso de ser necesario. (La figura 4.1 se muestra tras haber tratado la cadena "GABRIEL" y disponer todavía en el conjunto $\mathcal{E}$ elementos por analizar).

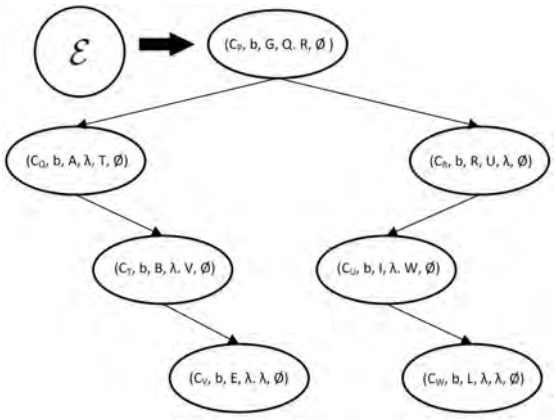


FIGURA 4.1: Procesos situación intermedia meta-algoritmo descendente

[1] Recorrer un (sub)árbol en *inorden* es:

- Recorrer en *inorden* el subárbol izquierdo.
- Visitar el nodo raíz del subárbol.
- Recorrer en *inorden* el subárbol derecho.

Las transiciones de las que dispone cada proceso consisten en la selección del pivote y la posterior transferencia (después de la comparación con dicho pivote) de los demás elementos de la entrada al proceso hijo correspondiente.

Las indeterminaciones que producen un repertorio de estados finales posibles son generadas por:

1. El orden de realización de las transiciones.
2. El elemento que se toma como pivote al activar un proceso (de entre los elementos presentes en ese momento en el componente de entrada).

En un refinamiento (como paso previo a una implementación) de este meta-algoritmo se fijan uno o ambos de estos factores, obteniéndose como resultado una reducción en el número de estados finales, pudiendo ser dicho valor incluso la unidad (único estado final).

Las actividades (transiciones) de cada uno de los procesos, son completamente independientes entre sí y se efectúan en cualquier orden, e incluso en paralelo, con una única restricción: *las actuaciones sobre los componentes compartidos han de ser excluyentes.* La consecuencia directa de esta restricción es la posibilidad de obtener distintos estados (árboles de procesos diferentes) según el orden en que se produzcan las transiciones. Esto a su vez conduce a estados finales distintos, no hay un estado final único, pero todos los estados finales representan la solución al problema.[2]

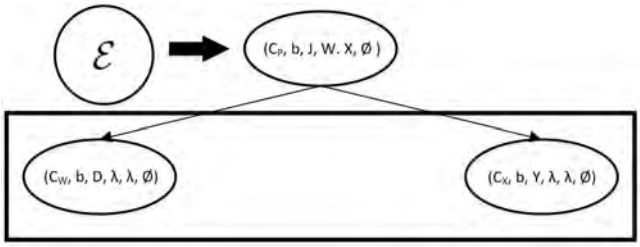


FIGURA 4.2: Procesos (incluso del mismo nivel) en ejecución simultánea

La única situación que ha de ser vigilada se produce cuando dos procesos acceden al mismo componente del estado. En este caso los accesos deben ser excluyentes e indivisibles (atómicos). Esto sólo ocurre cuando un proceso transfiere un elemento desde su entrada a la entrada de uno de sus subprocesos (*hijo*) y el subproceso debe acceder a su entrada. En este caso las dos

[2] En caso de ser aceptadas claves repetidas son correctas varias ordenaciones, disponiendo el conjunto de estados finales de más de un elemento.

actuaciones han de excluirse mutuamente: hasta que no termina una de ellas no puede comenzar la otra. El orden en que sean ejecutadas provoca que el procedimiento evolucione por distintos estados, terminando en estados finales diferentes, si bien igualmente válidos y equivalentes.

Cualquier otro paralelismo o simultaneidad que se pueda producir no afecta en absoluto al procedimiento de ejecución.

4.5. Representación gráfica

La forma elegida para mostrar gráficamente el proceso de ejecución del meta-algoritmo descendente, es mediante la utilización de un grafo dirigido acíclico.

Sea el grafo $G(V, A)$, representando los procesos en ejecución para un instante determinado, así como la información que se intercambian entre ellos. Las características que presenta son las siguientes:

- Cada nodo $v \in V$ del grafo, representa un proceso en ejecución que dispone de un conjunto de elementos a tratar y un elemento *pivote* seleccionado de dicho conjunto.
- Cada arista $a \in A$ que une dos procesos, representa la posibilidad de intercambio de información entre ambos. En este meta-algoritmo la información siempre fluye de forma descendente, desde la raíz a las hojas del árbol de procesos creado.

La ejecución del meta-algoritmo se divide en dos fases, aunque la fase realmente interesante es la primera en la que es construido el árbol de procesos. Tal y como se ha comentado, la segunda fase es un mero trámite que permite recuperar el resultado generado.

4.5.1. Primera fase: creación del árbol de procesos

Al iniciarse la ejecución del meta-algoritmo se crea el proceso P raíz del árbol, que dispone del conjunto completo de elementos $\mathcal{E}$ a tratar. Si el conjunto de entrada es igual al conjunto vacío ($C_p = \emptyset$), el meta-algoritmo finaliza su ejecución devolviendo como salida el mismo conjunto vacío ($S_p = \emptyset$). En cualquier otra situación el proceso es creado sin procesos hijo, sin elemento pivote, y disponiendo en su entrada del conjunto completo de elementos a tratar. La figura 4.3 refleja la situación descrita.

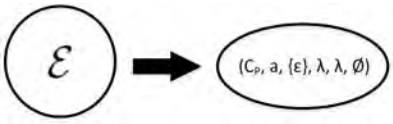


FIGURA 4.3: Situación inicial: único proceso

A continuación el meta-algoritmo realiza los pasos siguientes, siempre que existan elementos en el conjunto de entrada C_p del proceso raíz P.

1.- El proceso selecciona un elemento e cualquiera de su entrada. Al carecer de elemento *pivote*, coloca el elemento e como tal (la figura 4.4 muestra de forma gráfica la situación descrita).

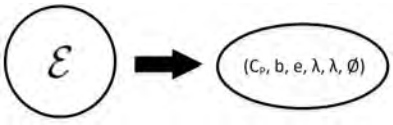


FIGURA 4.4: Proceso raíz una vez seleccionado el pivote

2.- El proceso raíz selecciona un nuevo elemento e' de su conjunto de entrada.[3]

3.- El proceso compara el elemento e' con el elemento seleccionado como *pivote* (La figura 4.5 representa esta situación).

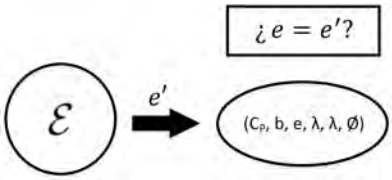


FIGURA 4.5: ¿Cómo es e' respecto al elemento pivote e?

4.- Si el elemento e' resulta ser menor que el *pivote* situado en el proceso es enviado al subárbol izquierdo. En caso de no existir subárbol izquierdo, previamente se crea un proceso P' como hijo por la izquierda del proceso P (se muestra en la figura 4.6).

[3]Siempre que el conjunto de entrada sea distinto del conjunto vacío. En caso contrario finaliza la ejecución del meta-algoritmo.

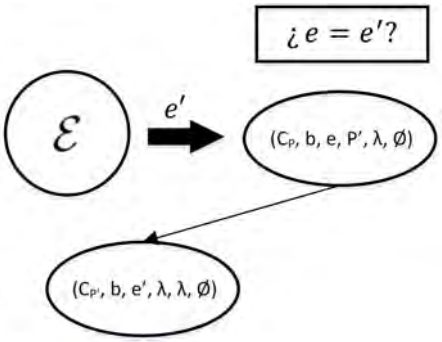


FIGURA 4.6: ¿El elemento *e'* es menor que el elemento *e*?

5.- Si el elemento e' es mayor que el *pivote*, de forma análoga al paso 4 se envía el elemento al subárbol derecho, si existe. Si no existe previamente se crea el proceso P' como hijo por la derecha del proceso (se muestra en la figura 4.7).

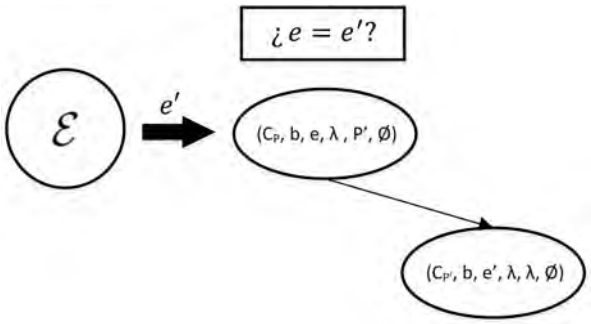


FIGURA 4.7: ¿El elemento *e'* es mayor que el elemento *e*?

6.- Transferido el elemento al hijo (derecho - izquierdo) correspondiente se repite el proceso de comparación volviendo al paso 3, hasta que al ser enviado el elemento a un subárbol tenga que crear proceso nuevo: en ese instante el elemento *e*‘ ocupará su posición definitiva como *pivote* del nuevo proceso creado.

7.- El meta-algoritmo finaliza cuando la entrada del proceso raíz se encuentre vacía y todos los elementos que se encontraban inicialmente en el conjunto de entrada, han sido colocados como *pivotes* cada uno en el proceso correspondiente.

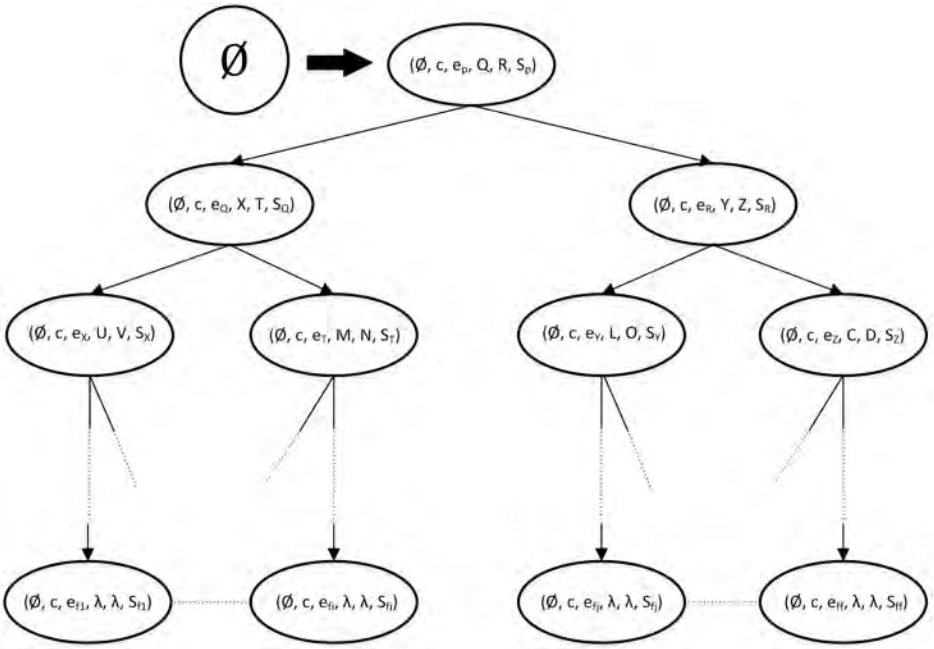


FIGURA 4.8: Árbol final del meta-algoritmo descendente

Finalizada la primera fase de ejecución del meta-algoritmo (la figura 4.8 representa esta situación) se garantiza que todos los elementos se encuentran distribuidos en el árbol de procesos como pivotes, pasando a la segunda fase o de recuperación de elementos.

4.5.2. Segunda fase: recuperación de elementos

Finalizada la construcción del árbol de procesos, los n elementos (si $\#\mathcal{C} \neq \emptyset$) del conjunto de entrada ($e \in C$) se encuentran situados como *pivotes* en cada uno de los procesos.

Para obtener la salida deseada es suficiente con realizar un recorrido en inorden del árbol de procesos, recuperando los *pivotes* que se encuentran en cada proceso. Se trata por tanto de una tarea rutinaria carente de relevancia.

Una vez finalizado el recorrido por el árbol, se obtiene en la salida el conjunto de elementos de la entrada cumpliendo la relación de orden deseada, por lo que se da por finalizada la ejecución del meta-algoritmo procediendo a la liberación de memoria ocupada por el árbol de procesos.

4.6. Versión formal

A continuación se define de una manera más formal[12] los componentes presentes en la ejecución del meta-algoritmo descendente:

- Un conjunto de entrada denominado $\mathcal{E}$ con los elementos a ordenar.
- Un proceso $P \in \mathcal{P}$ es una séxtupla $P = (\mathcal{C},\ e,\ v,\ Q,\ R,\ S)$ donde:
  - $\mathcal{C} \subset \mathcal{E}$ es un conjunto de elementos.
  - $e \in \{a, b, c\}$ representa el estado del proceso, con el siguiente significado:
    - a significa creado: sin pivote seleccionado aún.
    - b significa activo: el proceso P ya dispone de pivote.
    - c significa terminado: su entrada está vacía y no tendrá más elementos en el futuro.
  - $v \in \{\epsilon\} \cup \mathcal{E}$ es el pivote, si lo hay.
  - $Q, R \in \{\lambda\} \cup \mathcal{P}$ los subprocesos, si los hay.
  - $S \in \{\lambda\} \cup \mathcal{P}$ la salida que es una cola, y estará vacía salvo cuando el estado sea c.

Los diferentes estados del meta-algoritmo son alcanzados desde el estado inicial a través de las transiciones permitidas.

El estado inicial del meta-algoritmo descendente se corresponde con los siguientes valores $(\mathcal{C},\ a,\ \{\epsilon\},\ \lambda,\ \lambda,\ \emptyset)$. En la situación inicial, el conjunto $\mathcal{C}$ de elementos del proceso raíz se corresponde con el conjunto $\mathcal{E}$ de elementos a tratar. (La imagen que se corresponde con esta situación se encuentra reflejada en la figura 4.3).

Las transiciones aceptadas desde este estado son las que permiten la ejecución del meta-algoritmo y son las encargadas de acercarnos en cada paso hacia un estado final con solución.

Sea definido el proceso P de la siguiente forma $P = (\mathcal{C},\ e,\ v,\ Q,\ R,\ S)$. Las transiciones permitidas para P son:

1. Cuando se crea el proceso, no hay pivote y los enlaces a los hijos son nulos, por lo tanto si $e = a$, v ha de ser $\{\epsilon\}$, P y Q han de ser λ, y S ha de ser $\emptyset$, lo que se traduce en:
   - $\mathcal{C} \neq \emptyset$ y $v \in \mathcal{C}$
   - $\Rightarrow P' = (\mathcal{C} - \{v\},\ b,\ v,\ \lambda,\ \lambda,\ \emptyset)$

La imagen tras la elección del pivote se corresponde con la figura 4.4.

2. Proceso en estado activo, con elementos todavía por tratar (representado en las figuras 4.9 y 4.10). En este caso $e = b$, $\mathcal{C} \neq \emptyset$ y $w \in \mathcal{C}$, genera la siguiente transición:

   - Sea $w < v$.
   - $\Rightarrow P' = (\ \mathcal{C} - \{w\},\ b,\ v,\ Q,\ R,\ S)$
   - donde si $Q = \lambda$
     - $Q' = (\{w\},\ a,\ \{\epsilon\},\ \lambda,\ \lambda,\ \emptyset)$

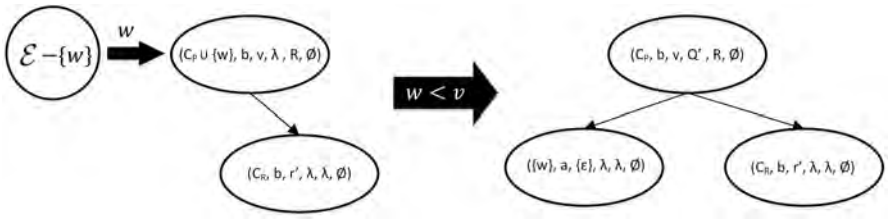


FIGURA 4.9: Transición si no existe proceso hijo izquierdo

   - y si $Q = (C',\ x,\ y,\ Z,\ T,\ S')$
     - $Q' = (C' \cup \{w\},\ x,\ y,\ Z,\ T,\ S')$

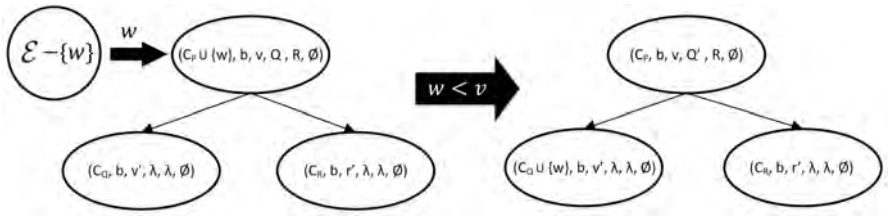


FIGURA 4.10: Transición si inicialmente existe proceso hijo izquierdo

3. De forma similar a la anterior sucede para el proceso R si $w > v$ (se muestra en las figuras 4.11 y 4.12). Análogamente $e = b$, $\mathcal{C} \neq \emptyset$ y $w \in \mathcal{C}$, generando la siguiente transición:

   - Sea $w > v$.
   - $\Rightarrow P' = (\ \mathcal{C} - \{w\},\ b,\ v,\ Q,\ R,\ S)$
   - donde si $R = \lambda$

- $R' = (\{w\},\ a,\ \{\epsilon\},\ \lambda,\ \lambda,\ \emptyset)$

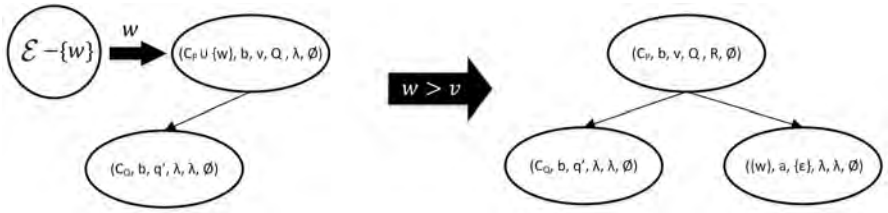


FIGURA 4.11: Transición si no existe proceso hijo derecho

- y si $R = (C',\ x,\ y,\ Z,\ T,\ S')$
  - $R' = (C' \cup \{w\},\ x,\ y,\ Z,\ T,\ S')$

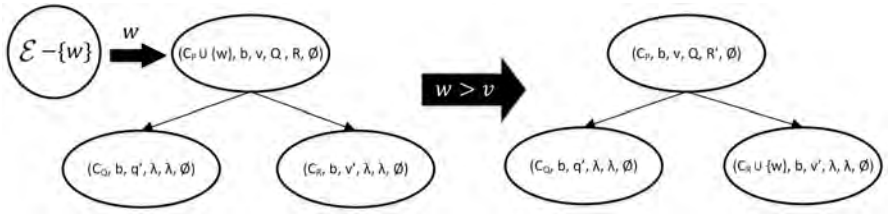


FIGURA 4.12: Transición si inicialmente existe proceso hijo derecho

4. Si el proceso raíz es $(\mathcal{C},\ b,\ v,\ Q,\ R,\ S)$ con $C = \emptyset$ (figura 4.13) la única transición posible es
   - $\Rightarrow (\emptyset,\ c,\ v,\ Q,\ R,\ S)$

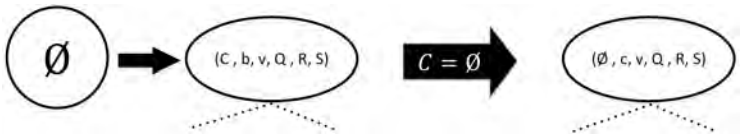


FIGURA 4.13: Transición desde el conjunto vacío

5. Si $\tilde{P} = (X,\ c,\ w,\ P,\ Q,\ S)$, X ha de ser $\emptyset$ y $P = (C,\ b,\ v,\ P',\ Q',\ S')$ con $C = \emptyset$ (figura 4.14)
   - $\Rightarrow P = (\emptyset,\ c,\ v,\ P',\ Q',\ S')$

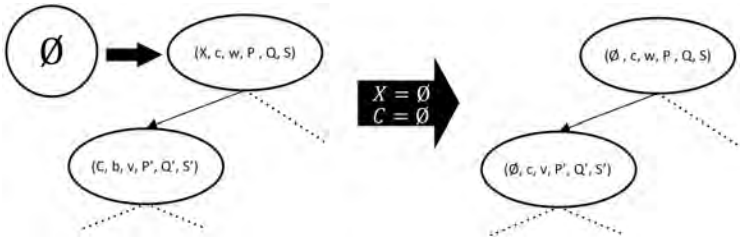


FIGURA 4.14: Transmisión señal de finalización al proceso izquierdo

6. Transición similar a la anterior para Q. Si $\tilde{Q} = (X,\ c,\ w,\ P,\ Q,\ S)$, X ha de ser $\emptyset$ y $Q = (C,\ b,\ v,\ P',\ Q',\ S')$ con $C = \emptyset$ (figura 4.15)

   - $\Rightarrow Q = (\emptyset,\ c,\ v,\ P',\ Q',\ S')$

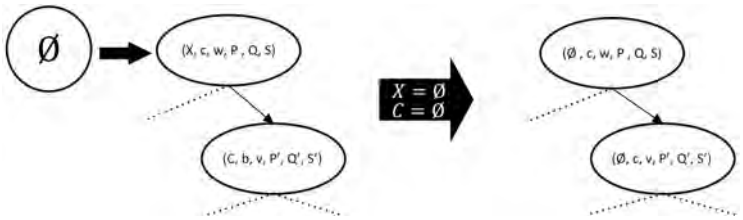


FIGURA 4.15: Transmisión señal de finalización al proceso derecho

7. Si $P = (\emptyset,\ c,\ v,\ Q,\ R,\ S)$ entonces:

   - Si $Q \neq \lambda$, se toman los elementos de la salida de Q para la salida de P (figura 4.18).

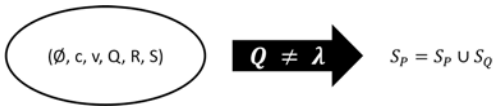


FIGURA 4.16: Transmisión de la salida proceso izquierdo hasta la raíz

   - Si $Q = \lambda$ y $v \neq \{\epsilon\}$, el pivote es llevado a la salida de P (figura 4.17).

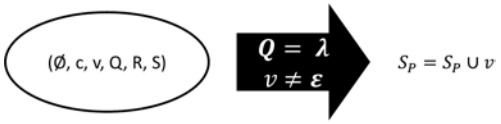


FIGURA 4.17: Transmisión del elemento pivote a la salida

- Si $Q = \lambda$, $v = \{\epsilon\}$ y $R \neq \lambda$, se toman los elementos de la salida de R para la salida de P (figura 4.18).

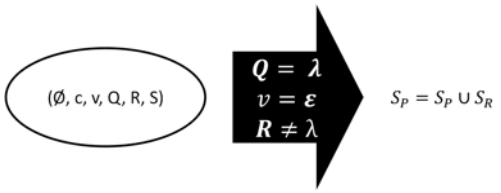


FIGURA 4.18: Transmisión de la salida proceso derecho hasta la raíz

- si $Q = \lambda$, $v = \{\epsilon\}$ y $R = \lambda$, el proceso P desaparece (figura 4.19).

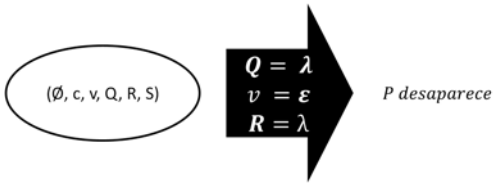


FIGURA 4.19: Desaparición del proceso P

8. Cuando la raíz es $(\emptyset,\ c,\ \lambda,\ \lambda,\ S)$, los elementos en S se encuentran en el orden deseado.

En resumen, un proceso se encuentra en un estado:

- a, cuando ha sido creado pero no se ha iniciado.
- b, procesa elementos de su entrada comparándolos con el pivote.
- c, recoge ordenadamente los resultados del subárbol con raíz en él dejándolos en la salida.

Un estado para el que no existe transición alguna posible es un estado final, y los elementos ordenados están representados por los pivotes de los procesos del estado final, recorriendo el árbol de procesos en *inorden*.

4.7. Corrección del meta-algoritmo descendente: versión informal

Para una correcta verificación de la corrección del meta-algoritmo descendente, al igual que la del resto de algoritmos, se deben cumplir las condiciones previamente definidas en el capítulo 2 apartado 2.1.

En primer lugar de una manera informal, se van a describir las características en las transiciones de los procesos del meta-algoritmo que son las que permiten asegurar su completa corrección.

La primera característica a cumplir, la definición sin ningún tipo de ambigüedad de los pasos sucesivos en la ejecución del meta-algoritmo, claramente se cumple a la perfección aplicando las únicas transiciones permitidas para cada proceso, transiciones descritas en el anterior apartado 4.6.

Con la definición realizada de procesos, todos ellos disponen de una entrada de elementos y de una salida en la que colocar el resultado tras la ejecución, pero en principio esto podría no ser suficiente, ya que no garantiza que el proceso finalice en algún momento.

Sin embargo según las descripciones previas, el proceso raíz termina cuando ha agotado su entrada: en ese punto todos los elementos a ordenar han sido considerados y se encarga de transferir esa información de finalización a sus procesos hijo, que a su vez se la transmitan a los suyos, reiterando este procedimiento hasta llegar a las hojas del árbol. Con ello se garantiza la finalización del algoritmo.

Cuando se crea cualquier proceso intermedio la entrada del mismo siempre es vacía, ya que así se ha definido en la transición: cuando el estado del proceso es igual a *creado (a)*, el conjunto de entrada es igual al conjunto vacío, el elemento pivote no existe (es $\{\epsilon\}$), no existe aún ningún proceso hijo y su salida es igual al conjunto vacío.

Para que un proceso pase de estado *creado* a estado *activo* ha de tomar de su entrada un elemento como *pivote* y no se desprende de él en todo el proceso de ejecución del algoritmo.

Cada una de las transiciones anteriormente descritas consume un elemento: bien se lo queda el propio proceso como pivote o bien lo transfiere a un nivel inferior. Esto garantiza que el proceso completo finaliza en algún momento: es finito.

La propiedad invariante de todo el procedimiento se puede ver de la siguiente forma: dentro de un subárbol, los elementos anteriores al pivote de la

raíz de dicho subárbol se encuentran en el subárbol izquierdo y los posteriores en el subárbol derecho, bien como pivotes o bien en la entrada de algún proceso.

Esta propiedad se cumple inicialmente y es conservada para todo el procedimiento por las transiciones definidas. Cuando el procedimiento completo finaliza no quedan elementos en ninguna entrada de ningún proceso. Los pivotes de cada proceso son tomados en *inorden*[4] para construir el resultado final del procedimiento de ordenación y con ello se obtiene el objetivo perseguido.

4.8. Corrección del meta-algoritmo descendente: versión formal

La corrección de un proceso exige la comprobación de que:

- El proceso termina al cabo de un número finito de transiciones.
- El proceso cumple una condición en su estado inicial, el invariante.
- Las transiciones conservar el invariante.
- El cumplimiento de la condición invariante en los estados finales implica la consecución del objetivo del proceso.

4.8.1. El proceso termina

Para llevar a cabo la corrección formal se describe el concepto *grado de un proceso*, que es representado por la abreviatura *gr*. Se podría intentar utilizar el grado de ignorancia, pero eso exige comprobar que cada transición lo disminuye, lo cual no siempre es directo, mientras que un grado por complejo que sea, más adaptado al meta-algoritmo, refleja más fácilmente las transiciones que se produzcan.

Para un estado concreto en el árbol de procesos se define su grado como:

$$gr = [(a_1, b_1)(a_2, b_2)...(a_k, b_k)]$$

donde b_i es el número de procesos de nivel i en el árbol de procesos con pivote (que han salido del estado inicial) y a_i el número de elementos en las entradas de los procesos de nivel i, siendo k el nivel máximo del árbol de procesos.

k será a lo sumo n (el número de elementos a ordenar) puesto que no se crean procesos de un nivel a menos que todos sus antecesores estén activos o terminados. La cadena más larga en el árbol de procesos ha de tener al menos

[4] Es en ese orden en el que se recogen los elementos cuando los procesos se encuentren en estado c (*finalizado*).

un elemento en cada nodo, como pivote en los primeros y como pivote o en la entrada en el último.

$\sum_{i=1}^{K}(a_i + b_i) = n$ puesto que todos los elementos están o bien como pivotes de un proceso $P \in \mathcal{P}$, o bien se encuentran en el conjunto de entrada $\mathcal{C}$ de algún proceso.

Una transición en un proceso es la encargada de mover un elemento. Dicho movimiento es:

1. Bien desde la entrada de un proceso a pivote de ese mismo proceso,
2. o bien desde la entrada de un proceso P a la entrada de otro proceso hijo P' (creando este si fuera necesario).

En ambos casos el grado se ve alterado con la siguiente variación:

- En el primer caso, al seleccionar un pivote para el proceso el grado cambia de
  - $gr = [(a_1,\, b_1)\,(a_i,\, b_i)\,...\,(a_k,\, b_k)]$ a
  - $gr = [(a_1,\, b_1)\,(a_i - 1,\, b_i + 1)\,...\,(a_k,\, b_k)]$
- En el segundo caso cuando un proceso cambia de estado "a" a estado "b", el grado cambia de
  - $gr = [(a_1,\, b_1)\,(a_i,\, b_i)\,...\,(a_k,\, b_k)]$ a
  - $gr = [(a_1,\, b_1)\,(a_i - 1,\, b_i)\,(a_{i+1} + 1,\, b_{i+1})\,...\,(a_k,\, b_k)]$

al comparar un elemento con el pivote que se encuentra en el proceso P y pasar dicho elemento a uno de sus procesos hijo.

Cuando comienza la ejecución del meta-algoritmo descendente el grado del estado inicial es $[(n, 0)]$.

Un estado final del meta-algoritmo se produce cuando en su grado $gr = [(a_1,\, b_1)\,(a_2,\, b_2)\,...\,(a_k,\, b_k)]$, todos los $a_i = 0$ (todas las entradas a los procesos se encuentran vacías).

Como todas estas transiciones mantienen constante la suma, que ha de ser n, desplazan valores hacia la derecha y $a_i + b_i \geq 1$, en $2\,n$ transiciones termina.

4.8.2. La condición invariante

En cada subárbol con raíz en un proceso con pivote v, todos los elementos que aparecen como pivotes o en las entradas de los procesos del subárbol, o bien se encuentran en el subárbol izquierdo y son anteriores al pivote v, o bien se encuentran en el subárbol derecho y son posteriores a dicho pivote.

4.8.3. El estado inicial cumple el invariante

Lógicamente en el estado inicial al estar todos los elementos en la entrada del proceso raíz no hay subárboles, por lo tanto dicho estado inicial cumple el invariante con rotundidad.

4.8.4. Las transiciones conservan el invariante

La transición de tomar un elemento y ubicarlo como pivote no cambia en absoluto el invariante: no se crea ningún proceso nuevo, únicamente el elemento transita por el proceso P desde el conjunto de entrada $\mathcal{C}$ hasta ocupar la posición destinada al pivote.

La transición de enviar un elemento de una entrada de un proceso P a la entrada de un proceso hijo P', se lleva a cabo transfiriéndolo al subárbol que le corresponde con la precisa finalidad de mantener el invariante.

4.8.5. En los estados finales el objetivo se satisface

El invariante aplicado a los estados finales (todas las entradas están vacías) se traduce en que:

- Todos los pivotes en el subárbol izquierdo son menores que el pivote de cualquier proceso.
- Todos los pivotes en el subárbol derecho son mayores que el pivote de cualquier proceso.

Por lo que el orden a construir se obtiene con un recorrido en *inorden* por los pivotes de los procesos del árbol.

4.8.6. Indeterminaciones

Las indeterminaciones de secuencia dinámica que no afectan a la validez del proceso y que se utilizan como grados de libertad en las implementaciones son:

- En la transición del estado "a" al estado "b", se escoge cualquier elemento de la entrada como pivote.

- Como los procesos sólo se comunican por las entradas y salidas, las posibilidades de paralelismo son múltiples con tal de que se respete la exclusión en el acceso a esos recursos que se comparten.

4.9. Optimizaciones

Hay una serie de optimizaciones aplicables a cualquier implementación, por ejemplo:

- Como las entradas y salidas tienen vidas activas (útiles) disjuntas (las entradas se utilizan en estado "*a*" [creado] y "*b*" [activo], mientras que las salidas se usan en el estado "*c*" [finalizado]), en cualquier implementación pueden compartir recursos físicos. Pero esas dos vistas han de respetar las exigencias de cada una: como entrada es un conjunto sin más condiciones, mientras que como salida ha de ser gestionada como una cola.

- A estas optimizaciones hay que añadir las susceptibles de ser aplicadas en cada implementación, por una parte restringidas por las decisiones sobre las estructuras de datos y control, y por otra aprovechando esas mismas restricciones.

4.10. Derivación de *Quicksort*

El algoritmo de ordenación clásica *Quicksort* (Apéndice J) constituye una derivación del meta-algoritmo descendente, a base de secuencializar la estructura de control.

Para llevar a cabo dicha secuencialización se establecen una serie de reglas en el orden de los procesos, siendo la principal que los procesos que se encuentran en la misma rama no pueden simultanearse. Por tanto, hasta que un proceso no alcanza el estado final, sus subprocesos no pueden ser iniciados.

Sin embargo esta secuencialización no es completa, quedan unos resto de indeterminación que se producen en el momento en que finaliza un proceso P y debe iniciarse uno de sus subprocesos (R o S). ¿Qué subproceso es iniciado?. Incluso podrían actuar ambos en paralelo al ser sus actividades mutuamente independientes.

La cristalización final del algoritmo establece que el proceso inicial trata todos los elementos, por lo que todos han de estar disponibles al inicio del procedimiento, siendo todos transferidos excepto el pivote a uno u otro de los subprocesos. Acto seguido, cada uno de esos subprocesos realiza idéntica acción con los elementos disponibles en su entrada.

Al disponer inicialmente de todos los elementos a tratar, y gracias a que el tratamiento de un proceso se ve como la partición de los elementos de la entrada en tres partes,[5] una estructura de datos secuencial (por ejemplo un vector) puede ser utilizada con algunas estructuras auxiliares para implementar los estados.

El algoritmo comienza su proceso seleccionando un elemento e del conjunto $\mathcal{E}$ a ordenar y lo compara con todos los demás ($n-1$ comparaciones), dividiendo el resto de elementos en dos subconjuntos: C_1 con todos los elementos menores que e, y C_2 con todos los mayores. Por tanto se genera una partición disjunta en la que se cumple que $\mathcal{E} = C_1 \cup e \cup C_2$ con la propiedad de que:

- $\forall\, a \in C_1,\ a < e$.
- $\forall\, b \in C_2,\ b > e$.

  de lo que se deduce por transitividad que $a < b$.

En las $n-1$ comparaciones que se producen se reduce el grado de ignorancia en esa misma cantidad. El algoritmo continua aplicando el mismo procedimiento a C_1 y C_2 recursivamente. Sus ejecuciones son óptimas cuando los subconjutos en que parte son del mismo tamaño ($\#C_1 \approx \#C_2$), en cuyo caso la complejidad es del orden de $n \log_2 n$. El peor caso posible se produce cuando en cada partición todos los elementos se van a un único subconjunto, quedando el otro vacío. En este caso la complejidad se va hasta el peor valor posible, llegando a $\dfrac{n(n-1)}{2}$: lógicamente en esta situación no se produce beneficio alguno en comparación por transitividad.

El elemento seleccionado como pivote establece en gran medida el comportamiento del algoritmo. Dicho comportamiento mejora al acercarse su valor al del promedio del conjunto, por lo que se puede pensar en una pequeña optimización (siempre que el coste sea razonable) que incluya la consideración de varios elementos del conjunto y el cálculo de la mediana de estos, con la esperanza de que el valor obtenido se acerque a la mediana del conjunto.

Por ejemplo, se toman cinco elementos del conjunto $\mathcal{E}$ y de ellos se selecciona el mediano. Con esto se garantiza que las particiones C_1 y C_2 tengan al menos dos elementos, eliminando de esta forma el peor caso. Se implementa habitualmente en memoria consecutiva intercambiando los elementos cuando

[5] Los elementos menores que el pivote, el propio pivote y los elementos mayores que el pivote.

se comparan si su posición relativa es contraria al resultado de la comparación.

Si el estado del meta-algoritmo es interpretado como un árbol de procesos, para este algoritmo todos los procesos del árbol que corresponden a nodos intermedios se encuentran en estado final, mientras que las hojas se encuentran en cualquiera de los tres posibles estados. Esto muestra claramente que la actividad sólo se produce en las hojas del árbol.

Formalmente, se parte de un estado inicial consistente I, con $I = (C,\ a,\ \{\epsilon\},\ \lambda,\ \lambda)$ donde $\mathcal{C}$ es el conjunto de elementos a ordenar, y después de la actividad de este proceso se llega al estado $I = (\emptyset,\ c,\ p,\ (C_1,\ a,\ \{\epsilon\},\ \lambda,\ \lambda),\ (C_2,\ a,\ \{\epsilon\},\ \lambda,\ \lambda))$, donde se cumple que para todo elemento $x \in C_1$ e $y \in C_2,\ x \leq p \leq y$.

En representación matricial, al final de esta primera fase suponiendo que el pivote es el elemento e situado en una posición media dentro del conjunto, las comparaciones quedan representadas como se muestran en la figura 4.20.

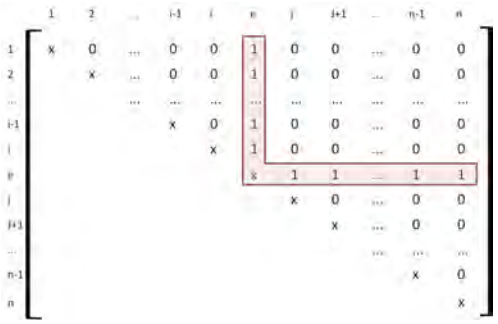


FIGURA 4.20: Matriz *Quicksort* tras la primera comparación directa

La información obtenida por la comparación directa se encuentra sombreada con un tono rojizo. Completando los elementos ya relacionados por transitividad de la propia relación, la matriz resultante es la que se muestra en la figura 4.21.

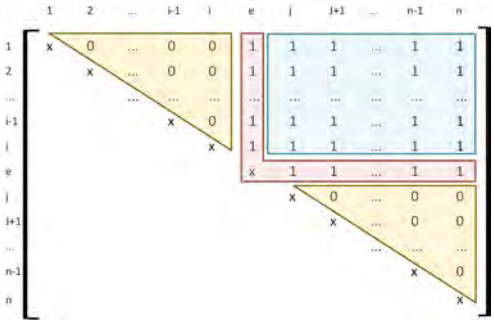


FIGURA 4.21: Matriz *Quicksort* tras añadir información por transitividad

La nueva información aparece ahora en la zona sombreada de color azul, y el problema se reconduce a ordenar los dos subconjuntos C_1 y C_2 que se corresponden en el gráfico con los dos triángulos de color amarillo.[6]

El proceso ha de ser reiterado hasta obtener todos los elementos de la matriz puestos a *uno*. Se eligen dos nuevos pivotes, $(i-1)$ y $n-1$, que son comparados con el resto de elementos contenidos en cada subconjunto. El elemento $i-1$ es comparado con los elementos situados en las posiciones $[1\,..\,i]$ (excluyendo al propio $(i-1)$), y el elemento $n-1$ es comparado con los elementos situados en las posiciones $[j\,..\,n]$ (excluyendo al propio $n-1$). Tras la comparación la matriz de incidencia presenta el aspecto que muestra la figura 4.22.

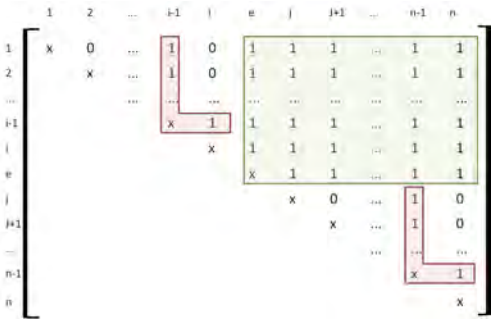


FIGURA 4.22: Matriz *Quicksort* tras seleccionar segundo y tercer pivote

Nuevamente la información sombreada en rojo es la obtenida por comparación directa de elementos con el pivote. (Cada comparación directa genera una "L" puesta a *unos* en la matriz). Completando la información conocida

[6] Se recuerda que por comodidad sólo se trabaja con los elementos situados en la mitad superior de la matriz, por encima de la diagonal.

por transitividad los valores matriciales pasan a ser los mostrados en la figura 4.23.

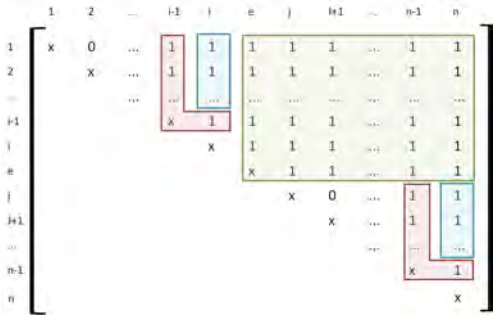


FIGURA 4.23: Matriz *Quicksort* añadida información por transitividad

La zona sombreada de azul es la información que se conoce y se añade por transitividad. El problema se reconduce a obtener la información desconocida que es representada por los elementos situados en los triángulos de color amarillo (figura 4.24).

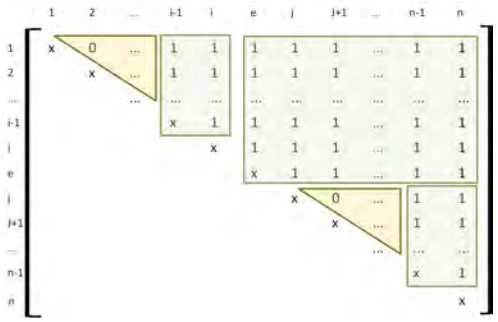


FIGURA 4.24: Matriz *Quicksort* con la que comienza la siguiente iteración

El proceso es repetido hasta obtener en la matriz todos sus valores puestos a *uno*.

En representación de grafos, las situaciones:

1.- Inicial,

2.- Posterior a la finalización del primer proceso,

3.- Posterior a que termine la actividad del primer subproceso,

son representadas como se muestra en la figura 4.25.

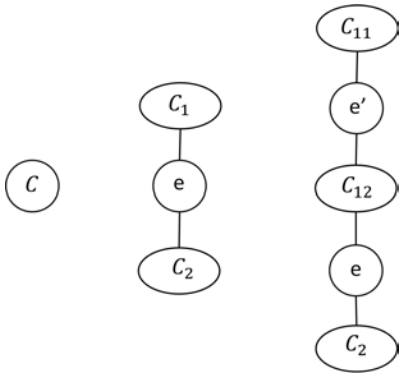


FIGURA 4.25: *Quicksort*: grafo tras las situaciones descritas

Por otro lado, si se representa gráficamente el árbol de procesos la transición es la que muestra la figura 4.26.

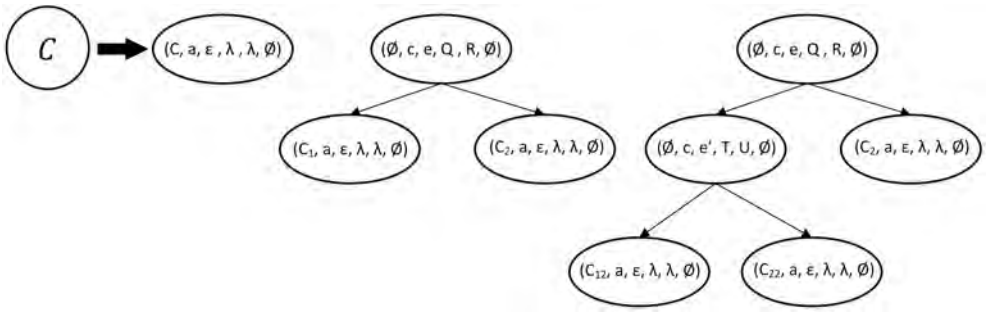


FIGURA 4.26: *Quicksort*: transición en el árbol de procesos

En el proceso de ejecución este algoritmo mantiene las dos siguientes indeterminaciones que han de ser resueltas por una implementación final secuencial:

1.- Al iniciarse un proceso, ¿qué elemento se toma como pivote entre los que se encuentran en la entrada de un proceso?.

2.- ¿Qué subproceso de entre todos (hojas del árbol de procesos) los que tienen la entrada no vacía se inicia?.

La primera indeterminación da lugar a árboles de procesos distintos, aunque en una implementación en memoria consecutiva el resultado final será el mismo.

La segunda indeterminación requiere más recursos en una implementación que ha de llevar un registro de los procesos pendientes de ser procesados.

Como ambas indeterminaciones pueden dar lugar a leves ineficiencias de tiempo o espacio, existen algunas variantes que tratan de minimizar las consecuencias, definiendo más minuciosamente la dinámica del algoritmo.

4.11. Derivación en *Binario*

El algoritmo *Binario* crea un árbol binario con un elemento en cada nodo, y con el invariante de que en todos los subárboles los elementos de los nodos del subárbol izquierdo son menores que el elemento del nodo raíz del subárbol, el cual a su vez es mayor que todos los elementos de los nodos del subárbol derecho.

El algoritmo se inicia tomando un elemento y se crea el nodo raíz del árbol a construir. Posteriormente se procesa el resto de elementos, siendo insertados en el árbol en construcción en el lugar que le corresponde. Se compara el nuevo elemento con el nodo raíz y:

- Si es menor y existe subárbol izquierdo, lo inserta en dicho subárbol.
- Si es menor pero no existe subárbol izquierdo, se crea un nodo con dicho elemento y se añade como subárbol izquierdo.
- Si es mayor, se realiza la tarea de forma simétrica pero con el subárbol derecho.

  El orden final se obtiene gracias a la propiedad del invariante, con sólo recorrer el árbol en inorden.

En oposición a lo explicado para el algoritmo *Quicksort*, el algoritmo *Binario* (Apéndice M) toma una decisión inversa a la anterior: asigna menor prioridad a un proceso que a cualquiera de sus subprocesos, lo que resulta en una secuencialidad caracterizada por el hecho de que son los datos los que definen la dinámica del procedimiento.

Una vez que el proceso inicial actúa sobre un elemento, ese elemento (salvo en la primera acción en que es tomado como pivote) es transferido a uno de sus subprocesos. Ese subproceso al no estar detenido por falta de tareas que realizar, tiene prioridad y actúa a la vez que el proceso padre. El elemento es transferido a sucesivos subprocesos hasta que se crea un nuevo subproceso que lo toma como pivote.

Este modo de funcionamiento no exige, al contrario que el *Quicksort*, que todos los elementos estén disponibles al inicio del procedimiento, lo que lo hace

preferible en situaciones en que se puede o debe simultanear la adquisición de elementos y su correspondiente tratamiento.

Sin embargo, la estructura de datos como árbol de procesos que contiene los pivotes debe mantenerse permanentemente. Pero la dinámica del procedimiento indica que salvo la entrada del proceso inicial, las entradas de los demás procesos estarán vacías siempre, salvo el breve intervalo entre la recepción de un elemento en la entrada y su inmediato proceso. Con esto, la estructura del árbol prescinde completamente del componente de entrada, debiendo sólo conservarse la entrada del proceso inicial que, al ser única, se trata de forma singular como la entrada al procedimiento completo.

4.12. Simetría o dualidad entre *Quicksort* y *Binario*

Entre estos dos algoritmos se da una cierta dualidad en la imposición de restricciones al meta-algoritmo. Ambos imponen condiciones a la estructura de control que en cualquier caso han de ser admisibles por el meta-algoritmo, optando por soluciones opuestas.

Para conseguir una estructura de control secuencial en una situación en que se admite una cierta indeterminación en el orden de actuación de distintos procesos, se ha de establecer un orden en esas actividades en forma de prioridades: si dos procesos pueden actuar: ¿en qué orden lo hacen?.

Desde este punto de vista, entre dos procesos cualesquiera de los que uno es subproceso del otro, *Quicksort* opta por dar menos prioridad al subproceso y además mantiene una cierta indecisión en el caso de dos procesos que se encuentren en otra posición relativa.

Como clara alternativa contrapuesta, *Binario* opta por la decisión contraria, estableciendo mayor prioridad al subproceso sobre su proceso padre. Esto es lo que induce que en este algoritmo, a lo sumo, dos procesos puedan actuar, el inicial[7] y quizás otro.

4.13. Optimizaciones específicas

4.13.1. *Binario*

Se prescinde de la entrada de los procesos puesto que un elemento que llegue a una entrada es extraído inmediatamente, y la entrada, excepto ese breve momento, está siempre vacía.[8]

[7] Si no ha terminado: en caso contrario la ordenación habría terminado.

[8] La excepción que confirma la regla se encuentra en el proceso raíz.

De la misma forma se prescinde de los estados: la transición del estado "*a*" al estado "*b*" es inmediata, y la transición al estado "*c*" se hace en todos los procesos en el momento que la entrada del proceso raíz se encuentre vacía.

También se puede prescindir de la salida, sustituyendo la segunda parte del procedimiento por un recorrido en inorden, recogiendo los pivotes de los procesos.

Como la eficiencia depende de que el árbol sea equilibrado, el proceso hace pausas para reorganizar y equilibrar el árbol, pero los siguientes elementos que se procesen pueden desequilibrar el árbol reorganizado, cuando, a lo mejor el equilibrio podría haberse alcanzado con sólo procesar esos elementos.

4.13.2. *Quicksort*

Aplicando una técnica similar a la del problema de la bandera holandesa, es posible utilizar una estructura de datos secuencial que permita el intercambio de elementos (es la única operación necesaria para el mantenimiento de esa estructura de datos y su significado).

Como un proceso actúa hasta que se detiene definitivamente, los estados "*a*" y "*b*" son innecesarios.

La entrada se produce por los elementos ya colocados en la estructura de datos. La salida es innecesaria, puesto que el estado final de la estructura secuencial representa perfectamente el orden final: es decir, la estructura lineal cubre las funciones de las entradas y salidas de los procesos.

En ambos casos se renuncia a ciertas posibilidades de paralelismo, a cambio de una simplificación de la estructura que de todas formas está presente o en la estructura del árbol binario o en la dinámica de la recursividad.

4.14. Consideraciones finales

No se utiliza el aspecto de los grafos porque no resulta productivo. Como lo que se hace en el fondo es insertar elementos en una cadena, la visualización del proceso con grafos es plana.

De hecho la estructura de árbol expresa:

- La lista lineal de los pivotes en *inorden* es la cadena.
- La estructura de árbol binario representa la estrategia de inserción binaria.

El darse cuenta de que se trata de inserción de elementos en cadena, llevará a un tipo particular de esquemas, aspecto que se trata en el capítulo 6.

Capítulo 5

Meta-algoritmo ascendente

Hay dos maneras de diseñar software: una es hacerlo tan simple que sea obvia su falta de deficiencias, la otra es hacerlo tan complejo que no haya deficiencias obvias.

C.A.R. Hoare

En el anterior capítulo 4 fue descrito el meta-algoritmo descendente, con las características intrínsecas del mismo y las ventajas que aporta su utilización frente a un algoritmo de ordenación clásico.

En este capítulo se describe su dual: el meta-algoritmo ascendente. Si el descendente basa sus principios de construcción de los algoritmos *Quicksort* y *Binario*, el ascendente unifica a los algoritmos clásicos *Fusión* (Apéndice L) y *Torneo* (Apéndice N) como extremos de un abanico de posibles implementaciones. La elección de ordenar subconjuntos de elementos por completo para posteriormente ser combinados frente a la posibilidad alternativa de obtener los primeros elementos lo antes posible, permiten pensar en dos estrategias dinámicas diferentes que asignan prioridades a subtareas de forma diametralmente opuesta.

5.1. Introducción

Al considerar los algoritmos clásicos de *Fusión* y *Torneo* se observan unas similitudes entre ellos que sugieren un enfoque que los engloba.

Ambos algoritmos construyen en una primera fase (explícita o implícitamente) un árbol binario de procesos distribuyendo en él los elementos a

ordenar. Posteriormente ambos comparan elementos consiguiendo resultados parciales que unen, hasta obtener la solución completa al problema.

Los elementos comparados son los primeros elementos asignados en los subárboles de cada proceso nodo. Esto es evidente en el algoritmo *Torneo*, mientras que en el algoritmo *Fusión* está latente en el sentido de que el árbol no está presente al completo en ningún momento; pero, si se prescinde de la evolución temporal, el árbol de procesos aparece. Los procesos se activan, se ejecutan hasta su terminación y desaparecen, pero su actividad, durante un cierto intervalo de tiempo, deja huella.

5.2. Versión intuitiva

El procedimiento consiste en la definición de una máquina abstracta cuyos estados son un árbol de procesos que actúan de forma independiente. El procedimiento se descompone en dos fases.

En la primera fase se crea un proceso inicial (raíz) de lo que será un árbol binario de procesos, y se le encarga la ordenación de los elementos: es el estado inicial.

Ahora cada proceso que tenga que ordenar más de un elemento, reparte estos en dos conjuntos no vacíos y crea dos subprocesos a los que encarga la ordenación de los elementos asignados.

Con la realización de esta tarea finaliza la primera fase, que se puede denominar de distribución o preparación.

En la segunda fase se realiza el proceso de ordenación propiamente dicho. Cada proceso debe proporcionar los elementos que le han sido encomendados a su proceso superior, en el orden deseado. Para ello dispone de una salida gestionada como una cola en la que va colocando el resultado de su tarea.

Si un proceso ha de ordenar un sólo elemento, da por realizada su tarea dejando el elemento en su salida y concluye su actividad.

Un proceso va considerando los primeros elementos de las salidas de sus subprocesos[1] y toma el menor de ellos para ponerlo en su salida. Si un subproceso tiene la salida vacía y está en estado final, es ignorado.

El proceso ha terminado su tarea cuando sus subprocesos han finalizado y sus salidas se encuentren vacías. Es entonces cuando cambia su estado a un

[1] Si una de esas salidas está vacía, o bien el subproceso ha terminado y entonces la salida del otro subproceso se copia en su salida, o bien el subproceso está detenido y todavía ha de procesar otros elementos, con lo que el proceso queda detenido y debe esperar a que le sean proporcionados en la salida correspondiente

estado final. Un proceso en estado final con la salida vacía puede ser eliminado del procedimiento puesto que ya no tiene actividad posible.

5.3. Representación gráfica

La forma de mostrar gráficamente la ejecución del meta-algoritmo ascendente se realiza mediante la utilización de un grafo dirigido acíclico.

Sea el grafo $G(V, A)$ representando los procesos en ejecución así como la información que se intercambian. Sus características son las siguientes:

- Cada nodo $v \in V$, representa un proceso en ejecución. Al menos se debe tener un proceso en la ejecución y su número máximo viene determinado por el número de elementos a tratar: a lo sumo, para n elementos se dispone de $2n-1$ procesos.
- Cada arista $a \in A$, representa la relación existente entre procesos, permitiendo el intercambio de información entre aquellos que se encuentren unidos mediante una arista.

Los siguientes apartados muestran las dos fases utilizadas por el meta-algoritmo para resolver el problema de ordenación. Para una mejor comprensión, se supone que la segunda fase (fusión) sólo es iniciada una vez finalizada la primera (distribución), pero no es necesario que esto ocurra así. De hecho, seleccionado un proceso (el raíz incluido), una de sus ramas puede encontrarse todavía en una fase de distribución de elementos cuando la otra rama comienza su proceso de fusión: son tareas independientes pero que no necesariamente han de realizarse en instantes diferentes en el tiempo.

5.3.1. Primera fase: Distribución de elementos

En la primera fase del meta-algoritmo ascendente se produce la división de cada conjunto de elementos $\mathcal{C}$ en dos subconjuntos C_1, C_2 ($\mathcal{C} = C_1 \cup C_2$), tratando de que la partición sea lo más equilibrada posible. La situación inicial del meta-algoritmo (Si $C \neq \emptyset$) se muestra en la figura 5.1.

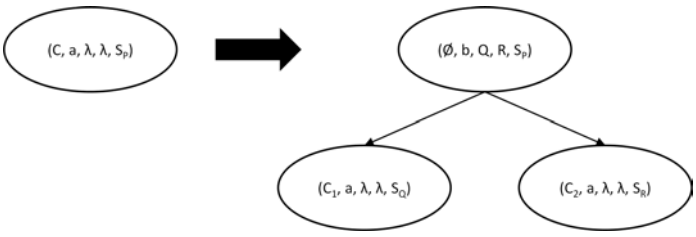


FIGURA 5.1: Situación inicial meta-algoritmo ascendente

Si el conjunto de entrada es vacío ($\mathcal{E} = C_i = \emptyset$), se devuelve como resultado de la ejecución el mismo conjunto vacío como salida del proceso raíz y se da por finalizada la ejecución ($S_f = \emptyset$). En caso contrario, el proceso raíz es el encargado de crear sus dos procesos hijo (Q y R) a los que transfiere los elementos correspondientes, respetando siempre la condición que impide crear un proceso con su conjunto de entrada vacío. (C_1 para $Q \neq \{\epsilon\}$ y C_2 para $R \neq \{\epsilon\}$).

El anterior procedimiento se repite en todos y cada uno de los procesos que se creen ($Q', R', \tilde{Q}, \tilde{R}...$), hasta que al crear el proceso correspondiente, este sólo reciba un elemento a tratar ($\#C_k = 1$), en cuyo caso lo sitúa directamente en su salida (C_k es transferido a S_k).

La figura 5.2 muestra el resultado del árbol de procesos (tratando de que cada división del conjunto de elementos se haga de forma equilibrada) tras la fase de distribución de elementos del meta-algoritmo ascendente, para el conjunto de elementos de entrada $\{G, A, B, R, I, E, L\}$, tomando como orden para el tratamiento la posición relativa que ocupa cada elemento dentro del conjunto.

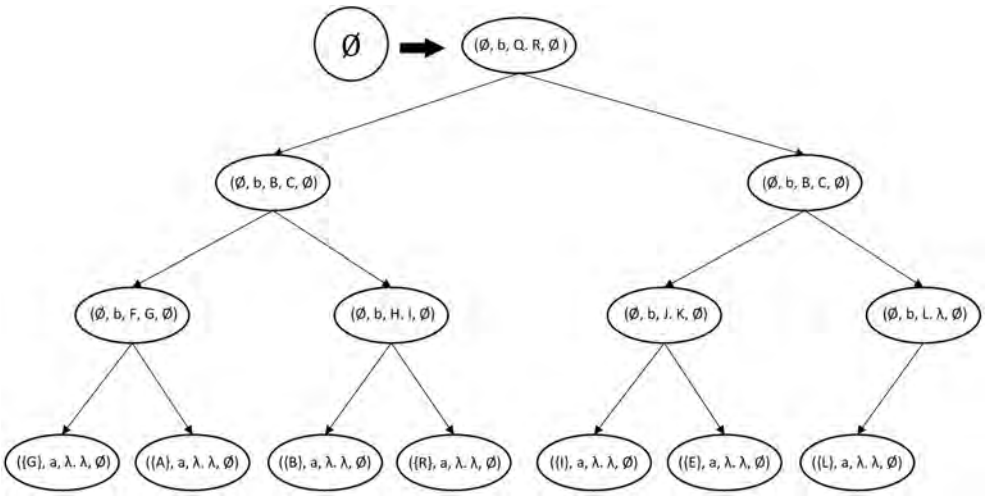


FIGURA 5.2: Ejemplo meta-algoritmo ascendente, entrada {G,A,B,R,I,E,L}

Llegados a este punto de la ejecución, la primera fase de distribución de elementos ha finalizado: cada proceso situado en las hojas del árbol dispone, a lo sumo, de un elemento en su conjunto de entrada. Recalcar por la importancia que de la decisión se deriva, que la división de cada conjunto de elementos

($\mathcal{C}$) en dos subconjuntos (C_1 y C_2) se realiza como mejor convenga. La eficiencia estadística recomienda utilizar la equipartición del conjunto, pero esas decisiones (particularizaciones) del meta-algoritmo se permiten alterar para cada elección, incluso siendo diferentes las elecciones para cada proceso del árbol. Se haga como se haga la distribución de los elementos, la primera fase finaliza para aquellos procesos cuyo conjunto de entrada está formado por un único elemento, que transfiere de forma inmediata a su salida correspondiente. Continuando con el ejemplo anterior, la figura 5.3 muestra el árbol de procesos al concluir la fase de distribución de elementos. (Podrían obtenerse otros árboles de procesos equivalentes en esta fase según el criterio que se utilice para distribuir los elementos).

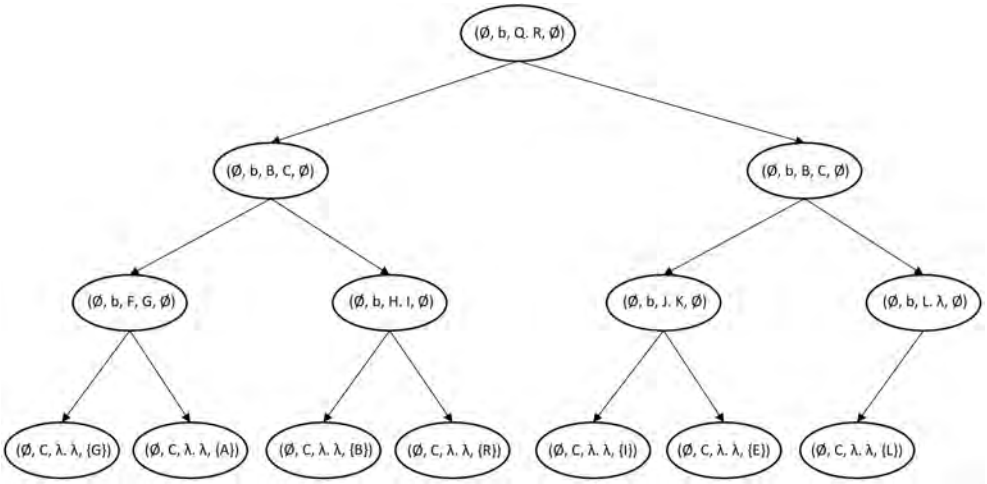


FIGURA 5.3: Meta-algoritmo ascendente finalizada la fase de distribución

5.3.2. Segunda fase: fusión de elementos

Finalizada la fase de distribución (e incluso como se comentó de forma simultánea, pero por simplicidad de explicación se desarrolla por separado) comienza la fase de fusión de elementos desde los procesos situados en las hojas, hasta obtener en la salida del proceso raíz los elementos en el orden deseado. Con estas características se parte como situación inicial de la imagen 5.3, que representa el estado de los procesos al finalizar la primera fase.

A partir de esta situación comienza el proceso de fusión. Supongamos un proceso $\tilde{P}$ cuyos procesos hijo $\tilde{R}$ y $\tilde{S}$ sean nodos hoja con un único elemento en sus respectivas salidas $S_{\tilde{R}}$ y $S_{\tilde{S}}$.

El momento en que comienza el proceso de fusión, depende en gran medida de las particularidades que se establezcan en la ejecución del meta-algoritmo. En la definición pura de este, el proceso $\tilde{P}$ en el momento que sus procesos hijo $\tilde{R}$ y $\tilde{S}$ disponen de su salida, compara sus elementos situados en las salidas $S_{\tilde{R}}$ y $S_{\tilde{S}}$ y transfiere el de menor valor a su propia salida $S_{\tilde{P}}$, posibilitando que su proceso padre P' la utilice para comparar dicho elemento y continuar transfiriendo el mismo hacia la salida del proceso raíz (se muestra la transmisión de elementos en la figura 5.4).

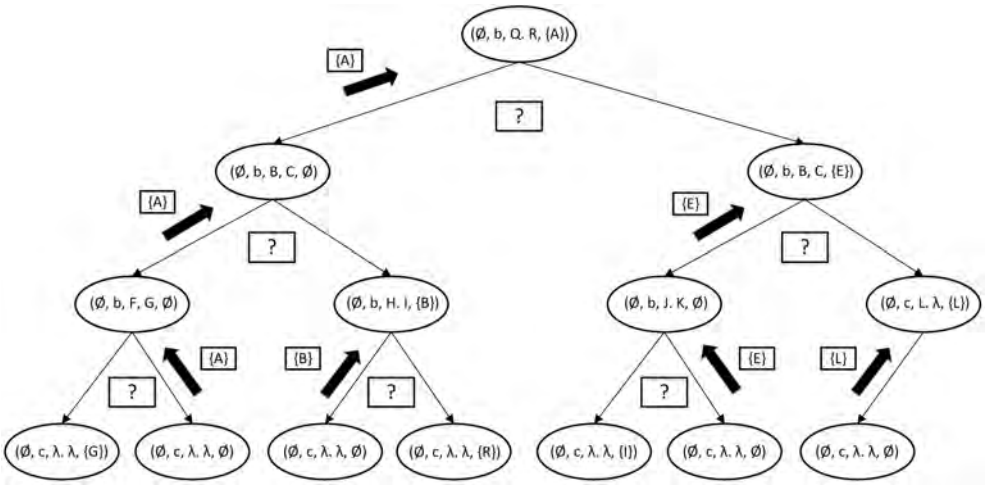


FIGURA 5.4: Meta-algoritmo ascendente en la fase de fusión

Esta mecánica es la utilizada por los procesos del árbol hasta finalizar todos y cada uno de ellos. Cuando un proceso transfiere la salida a su proceso padre y sus procesos hijo ya no disponen de elementos en sus salidas, el proceso puede ser eliminado del árbol.

Al finalizar la ejecución del meta-algoritmo, el proceso raíz dispone en su salida (gestionada como una cola) de todos los elementos que se encontraban en su entrada al comienzo de la ejecución, garantizando que se encuentran en la relación de orden deseada. El resto de procesos son eliminados liberando la correspondiente memoria ocupada. La figura 5.5 muestra el resultado al finalizar.

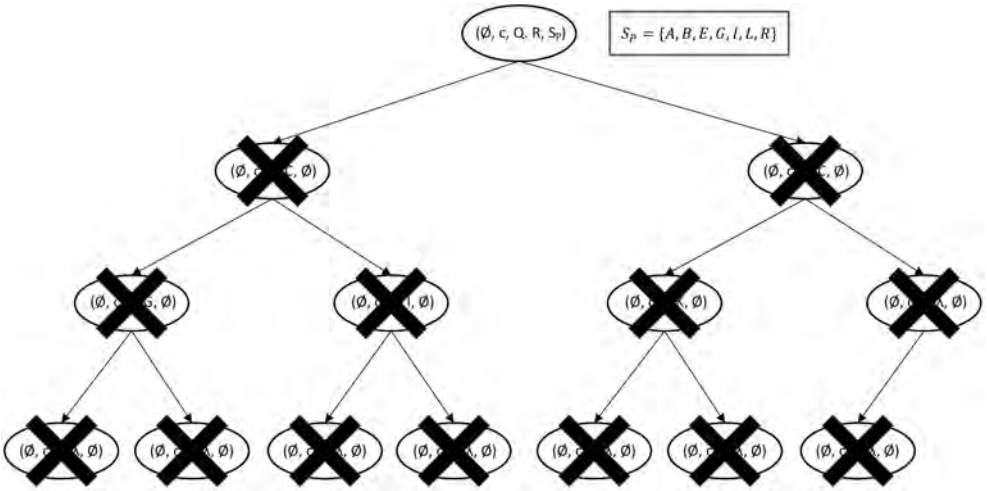


FIGURA 5.5: Meta-algoritmo ascendente finalizado

5.4. Versión formal

Para detallar formalmente el meta-algoritmo ascendente[12] se define un proceso $P \in \mathcal{P}$ como una quíntupla:

$P = (\mathcal{C},\ e,\ Q,\ R,\ S)$

El estado del meta-algoritmo ascendente consiste en un árbol binario de procesos definido de la siguiente forma:

- Se denomina $\mathcal{E}$ al conjunto formado por los elementos a ordenar.
- Un proceso $P \in \mathcal{P}$ es una quíntupla $P = (\mathcal{C},\ e,\ Q,\ R,\ S)$ donde:
  - $\mathcal{P} \subset \mathcal{E}$ es un conjunto de elementos.
  - $e \in \{a, b, c\}$ representa el estado del proceso, con idéntica descripción a la realizada en el capítulo 4 apartado 4.6[2].
  - $Q, R \in \{\lambda\} \cup \mathcal{P}$ los subprocesos, si los hay.
  - $S \in \{\lambda\} \cup \mathcal{P}$ se corresponde con la salida gestionada como una cola, que está vacía salvo cuando el estado del proceso sea finalizado.

Los elementos de entrada C de cada proceso P están contenidos todos en $\mathcal{E}$. Con estas definiciones la configuración inicial es $(\mathcal{C},\ a,\ \lambda,\ \lambda,\ \emptyset)$ (se muestra en la figura 5.6).

[2]Estado a=*creado*, b=*activo* y c=$finalizado$.

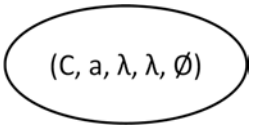


FIGURA 5.6: Proceso raíz meta-algoritmo ascendente ($\mathcal{C} = \mathcal{E}$)

Las transiciones permitidas desde este estado inicial son las que posibilitan el proceso de ejecución del meta-algoritmo a través de sus dos fases (distribución y fusión).

Las transiciones permitidas desde el estado inicial para el proceso P son las siguientes:

1. En la primera fase, se descompone el conjunto de elementos en dos subconjuntos siempre que la cardinalidad del mismo sea al menos dos. Si el proceso P dispone de los siguientes componentes $P = (\mathcal{C},\ a,\ \lambda,\ \lambda,\ \emptyset)$, siendo e no vacío y sea $\mathcal{C} = C_1 \cup C_2$ una partición de $\mathcal{E}$ con C_1 y C_2 no vacíos, entonces:

   - $\Rightarrow (\emptyset,\ b,\ (C_1,\ a,\ \lambda,\ \lambda,\ \emptyset),\ (C_2,\ a,\ \lambda,\ \lambda,\ \emptyset),\ \emptyset)$ La imagen del proceso se corresponde con la mostrada en la figura 5.1.

2. Si el proceso en estado activo sólo dispone de un elemento, lo coloca en su salida gestionada como una cola y pasa a un estado finalizado. La figura 5.7 muestra gráficamente la transición. Formalmente, si $P = (C,\ b,\ \lambda,\ \lambda,\ \emptyset)$, con $\#(C) = 1$, $g \in C$,

   - $\Rightarrow (\emptyset,\ c,\ \lambda,\ \lambda,\ (g))$

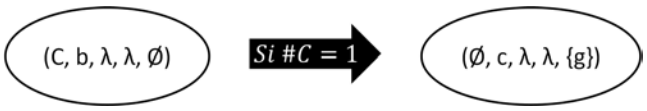


FIGURA 5.7: Cambio de estado de un proceso: de activo a finalizado

3. En una segunda fase de ordenación propiamente dicha, un proceso sin elementos en su entrada y con procesos hijo, coge el primer elemento que cada uno le entregue y los compara, eliminando dicho elemento de la salida de su proceso hijo y colocándolo en su propia cola de salida. Llevando a cabo una definición formal, si $P = (\emptyset,\ b,\ Q,\ R,\ S)$ con Q y R distintos de λ, y los componentes de salida de Q y R (S_1 y S_2) no están vacíos, entonces se toman q y r los primeros elementos de S_1 y

S_2 y se comparan (la transición se muestra en la figura 5.8). Si $q < r$ entonces:

- $\Rightarrow P = (\emptyset,\ b,\ \tilde{Q},\ R,\ S \oplus \{q\})$
  con $\tilde{Q} = (\emptyset,\ c,\ U,\ V,\ T_1)$,
  siendo $Q = (\emptyset,\ c,\ U,\ V,\ S_1)$ y $S_1 = \{q\} \oplus T_1$

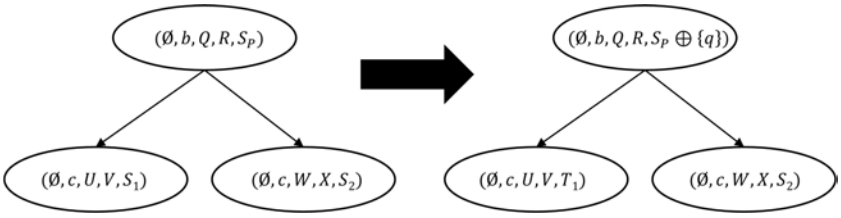


FIGURA 5.8: Fusión para $q < r$

Simétricamente si $q > r$ la transición (figura 5.9) pasa a ser:

- $\Rightarrow P = (\emptyset,\ b,\ Q,\ \tilde{R},\ S \oplus \{r\})$
  con $\tilde{R} = (\emptyset,\ c,\ W,\ X,\ T_2)$,
  siendo $R = (\emptyset,\ c,\ W,\ X,\ S_2)$ y $S_2 = \{r\} \oplus T_2$

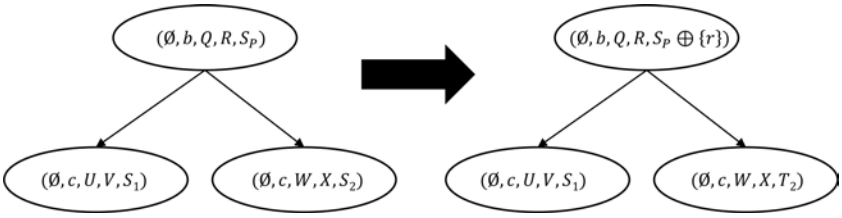


FIGURA 5.9: Fusión para $q > r$

4. Cuando para un proceso P sólo le restan por tratar elementos en la cola de uno de sus dos hijos, directamente traspasa la salida de ese hijo a su propia salida. Formalmente si $P = (\mathcal{E},\ b,\ Q,\ R,\ S)$ y alguno de sus procesos hijo Q o R (supongamos que esta situación ocurre en R) ya le ha entregado todos los elementos de los que disponía en su salida (por tanto salida $= \emptyset$), los elementos de la salida de Q son transferidos a la salida de P, de forma que

   - $\Rightarrow (\emptyset,\ b,\ \tilde{Q},\ \lambda,\ S \oplus (q))$
     con $(\tilde{Q}) = (\emptyset,\ c,\ U,\ V,\ T_1)$,

siendo $Q = (\emptyset,\ c,\ U,\ V,\ S_1) y S_1 = (q) \oplus T_1$

Si en lugar de ser el conjunto vacío ($\emptyset$) la salida del proceso R lo es la de Q, la situación es análoga, colocándose en este caso los elementos de la salida de R en la salida del proceso P (figura 5.10).

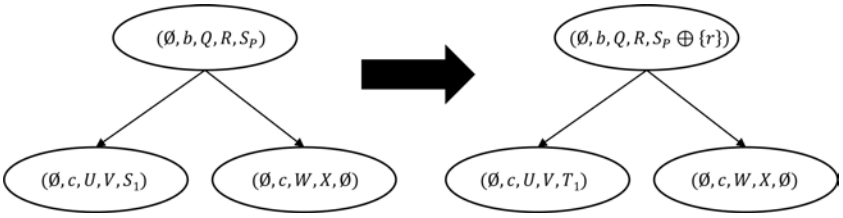


FIGURA 5.10: Transferencia de la salida cuando está vacía en un proceso

5. Cuando un proceso no dispone de elementos en su conjunto de entrada y sus procesos hijos tienen dirección igual a λ, pasa a estado finalizado y el conjunto de elementos ordenado está disponible en su salida (la figura 5.11 muestra la transición)
   Trascrito a lenguaje formal, cuando un proceso $P = (\emptyset,\ b,\ \lambda,\ \lambda,\ S)$, con S no vacío
   - $\Rightarrow (\emptyset,\ c,\ \lambda,\ \lambda,\ S)$

FIGURA 5.11: Paso de un proceso a estado finalizado

6. Cuando un proceso P no dispone de elementos en su entrada y su salida es vacía, el proceso es eliminado (figura 5.12).
   Sea el proceso $P = (\emptyset,\ b,\ \lambda,\ \lambda,\ \emptyset)$, su transición natural es
   - $\Rightarrow \lambda$, el proceso P es eliminado del árbol de procesos.

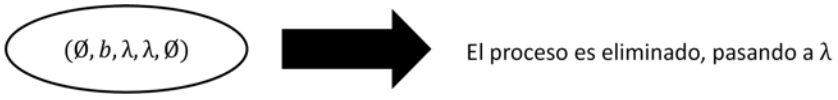


FIGURA 5.12: Estado en que el proceso es eliminado liberando memoria

Según los estados definidos para cada proceso P en el capítulo 4 apartado 4.6, trasladando dichos estados a las dos fases de las que dispone el meta-algoritmo ascendente, se establece que el estado a corresponde a la primera fase o fase de distribución del conjunto de elementos.

Cuando el estado de los procesos cambia a b, se puede afirmar que se corresponde con la segunda fase o fase de comparación entre los elementos de las salidas de los subprocesos.

Cuando uno de los subprocesos termina y su entrada se encuentra vacía (ese subproceso desaparece) la actividad se limita a copiar los elementos de la salida del subproceso hijo en la salida del proceso padre.

Cuando un proceso termina su actividad (no tiene subprocesos) queda en estado pasivo c hasta que su salida ha sido vaciada: entonces es eliminado y desaparece del árbol de procesos.

5.5. Corrección del meta-algoritmo: versión informal

El meta-algoritmo ascendente divide su trabajo en dos fases concretas a lo largo de las cuales ha de mantener el invariante para todo el proceso.

En la primera fase, o de *reparto* de los elementos a tratar, el proceso raíz dispone en su entrada de todos los elementos con los que trabajar. En cada paso sucesivo de esta primera fase, el conjunto de elementos es repartido en a lo sumo, dos subconjuntos, que pasan a ser la entrada de los subprocesos hijo con la condición de que ninguno de estos subconjuntos sea vacío. Con esta descripción el proceso raíz es el único que acepta como entrada el conjunto vacío: devuelve idéntico resultado en su salida y da por finalizada la ejecución del meta-algoritmo.

Cada proceso divide los elementos de su conjunto de entrada en dos subconjuntos no vacíos, lo que garantiza que cualquier proceso intermedio no es creado con su entrada vacía. Por otro lado, durante la fase de creación del árbol binario de procesos los elementos descienden desde la raíz a las hojas del árbol; la creación del árbol de procesos realizada de esta forma garantiza la imposibilidad de generar bucles infinitos, al no permitir la creación de procesos intermedios con entrada vacía.

Durante la fase de reparto de elementos las salidas de los procesos se encuentran vacías, y los elementos distribuidos son colocados en las entradas de los procesos situados en las hojas.

En esta primera fase descendente, cada paso que realiza el meta-algoritmo acerca el proceso a un estado final al dividir el conjunto de entrada en dos subconjuntos en los que cada uno contiene al menos un elemento. Por tanto si se

dispone en la entrada de n elementos a tratar, en el peor de los casos, cuando al dividir el conjunto de entrada en subconjuntos siempre la cardinalidad de uno de estos sea igual a *uno*, se finaliza la distribución de elementos en a lo sumo $n-1$ pasos.

En la segunda fase o fase ascendente, los elementos menores van trepando por el árbol de procesos hasta la raíz. Así realizada la declaración, en un instante concreto un elemento de la salida de un proceso es anterior a todos los que le siguen en la cola y a todos los que se encuentren en sus procesos descendientes. De forma análoga se garantiza que es posterior a todos los que le preceden en la cola de salida y a todos los que se encuentren situados en los procesos superiores.

A medida que se ejecuta el meta-algoritmo, la evolución del proceso hace que en esta segunda fase los elementos se desplacen hacia arriba por el árbol, lo que conlleva que en el estado final todos los elementos se encuentren en la salida del proceso raíz colocados según la relación de orden establecida.

Está garantizado el conjunto de elementos de entrada, el de elementos de salida, así como la finalización de la ejecución del proceso en un número finito de pasos, haciendo imposible que la ejecución caiga en bucles infinitos. Y por supuesto, dos ejecuciones diferentes del meta-algoritmo con los mismos elementos en su entrada, devuelven idéntico resultado salvo por la irrelevante ordenación de elementos repetidos vista en el capítulo 3 apartado 3.5.1.

5.6. Corrección del meta-algoritmo: versión formal

Para realizar una demostración formal de la corrección del meta-algoritmo ascendente, se define el concepto de grado de forma similar a la realizada para el meta-algoritmo descendente en el capítulo 4 apartado 4.8, pero más adecuado a la estructura ascendente que se pretende verificar. De esta forma se define el grado de una configuración cómo:

$(a_1, a_2, ..., a_i, ..., a_k)$ donde a_i es el número de elementos disponibles en las salidas de los procesos de nivel i.

Finalizada la fase de distribución el número de niveles es a lo sumo n, y en caso de equipartición es de $\log_2 n$.

Cada comparación y subsiguiente acción de un proceso de nivel i pasa de una configuración de grado

- $(a_1, ..., a_i, a_{i+1}, ..., a_k)$

a una de grado

- $(a_1, ..., a_i + 1, ai + 1 - 1, ..., a_k)$

De la anterior descripción se deduce que en un número finito y acotado de pasos se llega a una configuración con grado (n, .., 0, ..., 0). Dicho estado no tiene configuración siguiente posible, por tanto la configuración mostrada es la final, la cual dispone de todos los elementos en la salida del proceso raíz según la relación de orden establecida.

El invariante que cada configuración del proceso ha de cumplir y que las acciones han de conservar, se corresponde con las siguientes características:

- Los elementos en la salida de un proceso están en una cola ordenada, y todos ellos son anteriores a los elementos que estén en procesos inferiores.
- La configuración inicial cumple el invariante.
- Cada acción mantiene el invariante.
- La configuración final, al cumplir el invariante, satisface el objetivo final.

5.7. Indeterminaciones

Los grados de libertad que se pueden ejercer en este meta-algoritmo son:

- La forma de realizar las particiones en la fase de distribución, en cuanto a qué tamaño tienen los conjuntos resultado y cómo se distribuyen los elementos. La única condición a respetar es que los conjuntos resultantes sean no vacíos, para garantizar que no hay bucles infinitos en esta fase.
- El orden de ejecución de los distintos procesos que se pueden secuencializar de distinta forma o ejecutar con distintos grados de simultaneidad.

5.8. Optimizaciones genéricas

El reparto de tareas en la primera fase es óptimo si divide por iguales entre sus procesos hijos el conjunto de elementos a ordenar (Véase apéndice D).

Un reparto que encarga a un hijo la ordenación de todos los elementos y al otro hijo el conjunto vacío, podría dar lugar a un bucle infinito (un árbol de procesos infinito), por lo que dicha situación quede totalmente prohibida en cualquier momento de la ejecución.

Las entradas y salidas son de vidas disjuntas, por lo que una implementación puede hacer que compartan recursos físicos. No obstante las dinámicas son distintas, por lo que las dos vistas han de tenerlo en cuenta.

La equipartición mejora la eficiencia puesto que la fusión de dos cadenas de longitud p y q requiere entre un mínimo de $min(p, q)$ y un máximo de $(p+q-1)$, lo que extendido a todo el procedimiento y aplicando la equipartición en la fase de distribución resulta en una complejidad mínima de $\frac{n}{2} \log_2 n$ con un máximo de $n \log_2 n - n$.

Comparado con otros algoritmos de ordenación, resulta que si bien su mínimo es innegociable y no puede detectar y aprovechar situaciones especiales (cómo que los elementos ya estuvieran ordenados) su máximo es comparable con ventaja al de cualquier otro algoritmo.[3]

5.9. Derivaciones

La ejecución del meta-algoritmo permite derivar algoritmos secuenciales dando prioridad a unos procesos sobre otros, obteniendo los diferentes algoritmos clásicos en él contenidos y que son cristalizados mediante una implementación concreta y detallada.

Se hace necesario recordar en este punto que el árbol binario que se construye a lo largo de la primera fase de la ejecución, es un árbol binario de procesos, no de elementos. Eso permite la posibilidad de aplicar paralelismo entre los diferentes procesos que componen el árbol.

Es verdad que no siempre será posible llevar a cabo varias tareas en simultáneo o, por decirlo de otra forma, es posible que la aportación obtenida por el paralelismo no sea la deseada de no ser puesta en práctica con un análisis previo de la forma de funcionamiento del meta-algoritmo.

Con la descripción de la división del proceso de ejecución en dos fases[4] se pretende detallar el proceso a realizar, pero eso no implica que ambos procesos no puedan realizarse bajo unas determinadas circunstancias de forma paralela. Dicho de otro modo, no es necesario que finalice por completo la fase de distribución para que se inicie la fase de fusión.

Por ejemplo si se da prioridad a una de las ramas de árbol de procesos y una vez que se ha enviado un número determinado de elementos a dicha rama, se puede dar por finalizada la fase de distribución para la misma. Seguidamente comienza el proceso de fusión permitiendo adelantar la ejecución en la ordenación de los elementos dispuestos en ella. Esto no implica que no sea necesario tratar al menos una vez todos los elementos para saber cuál es el primero, pero sí que permite adelantar tareas del trabajo a realizar para que,

[3] Puesto que hay $n!$ posibles ordenaciones y cada comparación a lo más divide las posibilidades por 2, el número de comparaciones que un algoritmo debe hacer es, en general, $n \log_2 n - n \log_2 e$, usando la aproximación de Stirling $n! \approx n^n e^{-n} \sqrt{2\pi n}$.

[4] Primera fase de distribución y segunda fase de fusión.

en el momento en que llegue el último elemento a la entrada del proceso raíz se pueda responder con la mayor inmediatez posible acerca de los primeros elementos.

A continuación se muestran dos claros ejemplos donde se refleja como priorizar una u otra zona del árbol de procesos, conduce las tareas realizadas por el meta-algoritmo a una situación similar a la de uno u otro algoritmo clásico de ordenación.

5.9.1. Derivación de *Fusión*

El algoritmo clásico de *Fusión* se adapta perfectamente al árbol de procesos establecido para el meta-algoritmo ascendente. En primer lugar, *Fusión* se encarga de realizar una serie de divisiones del conjunto de elementos de entrada en varios subconjuntos y en una segunda fase obtiene el resultado de la ordenación por fusión de subconjuntos previamente ordenados.

Como se puede comprobar intuitivamente la adaptación a los procesos descritos es casi perfecta. Sin embargo hay una serie de pequeños detalles de implementación que hacen diferir su funcionamiento respecto al meta-algoritmo: en el algoritmo *Fusión*, para que el proceso padre realice la fusión de los elementos entregados por los procesos hijo, se impone la restricción de que ambos subprocesos hijo dispongan de todos sus elementos ya ordenados.

Realmente desde el momento en que se dispone de cada elemento de menor valor de cada subproceso el proceso padre del meta-algoritmo puede realizar la comparación, pero *Fusión* impone que se debe esperar a que cada uno de los subprocesos hijo haya concluido su tarea. Lógicamente esto son sólo detalles de implementación de un algoritmo concreto que nada tienen que ver con el problema real de obtener la relación de orden total del conjunto de elementos.

Si en el meta-algoritmo se da preferencia (prioridad) a los procesos situados en la parte baja del árbol (procesos inferiores situados en las hojas y sus cercanías) se obtiene como resultado que un proceso por encima de estos no actúa hasta que sus hijos no hayan finalizado con su tarea.

La visión obtenida es que cada proceso sólo realiza la fusión de las cadenas resultantes (en las salidas) del trabajo de sus hijos cuando ambos hayan finalizado, proporcionando una nueva cadena resultante de la fusión de estas a un proceso superior.

Establecida la prioridad de esta forma, el paralelismo se concentra en los procesos más cercanos a las hojas, ya que en los procesos superiores aunque se quiera actuar no es posible, al carecer de elementos que procesar hasta que no termine para cada proceso toda la tarea de sus 2 subprocesos hijo.

5.9.2. Derivación de *Torneo*

El algoritmo clásico de *Torneo* en cualquiera de las múltiples variantes que existen definidas también se encuentra incluido dentro del meta-algoritmo ascendente, pero realiza su tarea de forma un poco diferente a como lo hace *Fusión.*

En este caso la cristalización del algoritmo *Torneo* establece una prioridad para aquellos procesos que se encuentran situados en la zona superior del árbol. Por este motivo, el proceso raíz y sus allegados gozan de mayor prioridad.

Al dar prioridad a los procesos superiores se obtiene la siguiente visión: cada proceso espera que sus subprocesos hijo le proporcionen cada uno el primer elemento de su cola de salida lo antes posible para ser comparados y transferir el resultado a su proceso padre. Al contrario de lo que sucede con el algoritmo *Fusión,* no es necesario que cada proceso termine su tarea completa para transferir los elementos en el orden apropiado a su proceso padre. En el momento que obtiene el primer elemento lo transfiere, dando así prioridad a los procesos que se encuentran en la zona más alta del árbol.

La estructura encargada de representar la situación e información conocida en un momento determinado es la de un árbol binario. El nodo raíz se deshace de su elemento y a la mayor brevedad posible lo sustituye con el menor de los elementos de sus hijos, lo que desencadena la misma acción en el nodo del que ha tomado el elemento.

Haciendo una pequeña simetría entre ambos algoritmos, si *Fusión* funde cadenas, *Torneo* maneja árboles binarios.

Como observación adicional, el algoritmo *Heap* es una adaptación de este mismo procedimiento a una gestión particular de un tipo de estructura de datos concreta (secuencial interpretada como árbol binario).

5.10. Simetría o dualidad

Torneo da prioridad a los procesos superiores en el árbol mientras que *Fusión* da esa prioridad a los procesos inferiores.

En *Torneo* tiene prioridad el proceso superior, que como se detiene cuando uno de sus hijos tenga la salida vacía le pasa la prioridad a ese hijo, pero la recupera en cuanto ese hijo coloque un elemento en la salida; esto hace que la salida de un proceso (excepto el raíz) tenga un elemento solamente o esté vacía.

5.11. Optimizaciones específicas

5.11.1. *Torneo*

Las particularidades de implementación de los algoritmos contenidos en el meta-algoritmo permiten adaptar determinadas características del mismo, obteniendo como resultado un proceso más óptimo.

En el caso del algoritmo *Torneo* la salida de cada proceso (excepto para el proceso raíz) está muy limitada: la citada salida nunca tiene más de un elemento. O se encuentra vacía o dispone a lo sumo de un único elemento, en cuyo caso o bien es ocupada inmediatamente por elementos procedentes de un nivel inferior, o bien el proceso desaparece al haber finalizado su tarea. Por este motivo no es necesario que sea implementada como una cola.

5.11.2. *Fusión*

En el caso del algoritmo *Fusión*, al dar prioridad a los procesos situados en zonas inferiores del árbol un proceso no comienza a actuar mientras sus procesos hijo se encuentren activos.

Por ello se consideran como superfluos los estados de los procesos siendo simplificados, ya que el estado en que se encuentra cada proceso realmente queda determinado por la dinámica local.

Capítulo 6

Esquemas

> *[In mathematics, as] in many scientific research, we find two tendencies present. On the one hand, the tendency towards* abstraction *seeks to chrystalize the* logical *relations inherent in the maze of material that is being studied, and to correlate the material in a systematic and orderly manner. On the other hand, the tendency towards* intuitive understanding *fosters a more inmediate grasp of the objects one studies, a live* rapport *with them, so to speak, which stresses the concrete meaning of their relations.*
>
> Hilbert

6.1. Introducción

6.2. Esquemas

Los esquemas son un paso más de la abstracción. Si los meta-algoritmos prescinden de la implementación de la estructura de control definiendo sólo las acciones necesarias, los esquemas prescinden de la implementación de las estructuras de datos, limitándose a considerarlas de forma abstracta con las operaciones que son mínimamente necesarias y limitan la estructura de control a la mínima precedencia entre operaciones. En buena parte la presencia de la estructura de control perturba el punto de vista desviando la atención hacia la eficiencia real o pretendida, mientras que lo esencial que es la corrección queda desatendida.

Así el meta-algoritmo descendente y sus algoritmos derivados, operan insertando un elemento en una cadena que se va construyendo hasta llegar a

una cadena única final que representa el objetivo a alcanzar.

De igual manera el meta-algoritmo ascendente combina árboles y opera en el interior de estos hasta obtener un árbol final completamente degenerado o, lo que es lo mismo, una cadena.

Un esquema opera con una familia de estructuras de elementos que:

- Incluya la situación inicial: una colección de elementos simples.
- Incluya la situación final: una cadena única.
- Opere con las estructuras presentes para construir otras de la misma familia que están más próximas a la situación final.

Para ello las estructuras se restringen a familias de forma que:

- En una situación dada, sea fácil determinar las acciones adecuadas (efectivas) a realizar.
- Las estructuras inicial y final sean de la familia.
- Las acciones transformen las estructuras produciendo otras dentro de la misma familia.

Así se consideran distintos tipos de esquemas.

6.3. Germen de la idea

Llegados a este punto no es necesario recalcar la importancia que supone el concepto de abstracción para la realización del presente trabajo. Por este motivo y aplicando nuevamente abstracción sobre los meta-algoritmos (concretamente sobre la estructura de datos de los mismos) se va un paso más allá, todavía más cerca del concepto ordenación.

La idea de los esquemas está inspirada en los esquemas de análisis sintáctico de Sikkel [12] en los que unas mínimas estructuras de datos flotan en una *sopa primordial*[11] interactuando y transformándose hasta alcanzar el objetivo.

El concepto es sencillo: supongamos que se dispone de un plato de sopa de letras, en el que estas se pueden ir uniendo sin restricción alguna hasta obtener una determinada asociación entre ellas. Lógicamente si no se impone ningún tipo de restricción ni orden en el modo de combinarse, la estructura

de datos final será un grafo conexo, sin disponer de orden alguno, que contiene a todas las letras iniciales "entrelazadas"de una forma totalmente aleatoria.

Si se trasladan los conceptos a los algoritmos de ordenación, el plato representa el conjunto de entrada del algoritmo y las letras de la sopa los elementos a ordenar. Al tratarse de una sopa de letras se puede establecer una ordenación lexicográfica, por ejemplo.

Si se trasladan al meta-algoritmo esos conceptos, la operación de ordenación de elementos ha de ser previamente establecida puesto que la naturaleza de los elementos a ordenar puede ser dispar.

Volviendo sobre la sopa de letras, en la ordenación del conjunto de elementos se pueden establecer una serie de restricciones, sobre todo aquellas que vienen impuestas por la propia naturaleza del problema a tratar. No resulta interesante trabajar con un grafo de cualquier naturaleza por las propiedades intrínsecas de una relación de orden. Disponer de un grafo que presente ciclos no es más que un problema añadido grave que puede degenerar en un bucle sin fin para el esquema de resolución.

Por este motivo la máxima abstracción con la que se trabajará es el grafo dirigido acíclico, que representa perfectamente la naturaleza del problema de ordenación. Tal y como se comentó en el apartado 3.9, el diagrama de *Hasse* representa la mínima relación cuya clausura transitiva es el orden parcial conocido en un momento dado.

Además como apoyatura visual que permita tomar decisiones encaminadas a alcanzar un estado final, la matriz de incidencia es de gran utilidad.

Como desarrolla en los siguientes apartados, restringir las posibilidades de comparación de elementos a estructuras finales de tipo cadena o árbol es lo que permite obtener, a partir de esquemas generales, los dos tipos de meta-algoritmo.

Si la restricción no es tal pero sí que se piensa que las estructuras únicamente pueden ser lineales, entonces se encuentra la fusión de cadenas que no es más que un caso particular de la fusión de grafos en general. La fusión de cadenas no es más que una mera particularización de la fusión de árboles, que a su vez es una particularización de la fusión de grafos en general que se obtiene de la abstracción del meta-algoritmo ascendente.

Se comienza con la visión establecida para el meta-algoritmo descendente: fusión únicamente de elementos o fusión de un elemento con una cadena.

6.4. Esquemas generales

La familia sobre la que trabajan es la de grafos dirigidos y acíclicos, que representan el orden parcial construido hasta ese momento y que debe ser un suborden del orden total buscado. La restricción de grafos genéricos a la familia de grafos dirigidos acíclicos es válida ya que:

- Contiene toda la información conocida.
- Contiene a todos los elementos sueltos presentes en un estado inicial (nodos).
- Contiene a la cadena única que representa el estado final.
- Cualquier transición que se produce lo hace dentro de la propia familia de grafos. Se hace una pregunta y al añadir la nueva información obtenida (de forma directa y por transitividad) se mantiene dentro de la familia de grafos dirigidos acíclicos.

Sin embargo estos esquemas son difíciles de mantener y la determinación de las acciones eficaces puede ser demasiado compleja.

No obstante realizar el proceso sobre esta base se traduce en una búsqueda genérica de la solución al problema de ordenación de elementos (mínimo número de restricciones posible). No se establece ningún tipo de secuencia u orden (preferencia) en el momento de seleccionar los elementos a comparar, es un proceso completamente aleatorio. Lo que se debe garantizar al aplicar cualquier transición es que al añadir la nueva información tras la comparación se mantenga la estructura dentro de la familia original de grafos (en este caso acíclico y dirigido). Por este motivo lo que no se permite bajo ningún concepto es la formación de bucles en la estructura, pues puede conducir la situación a un bucle infinito indeseable.

La mejor forma de mostrar las diferentes alternativas que se pueden presentar es realizar un ejemplo valiéndonos para su desarrollo de la matriz de incidencia. Según la definición establecida en el capítulo 3 apartado 3.9.2, es suficiente trabajar con los valores de la matriz situados por encima de la diagonal.

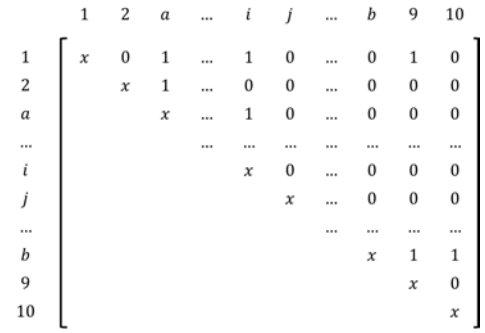


FIGURA 6.1: Situación intermedia de trabajo con esquema general

La matriz de incidencia representada en la figura 6.1 muestra una situación intermedia en el proceso de ordenación. Por los valores que contiene la matriz no representa un estado final del proceso, por lo que el siguiente paso es seleccionar dos elementos para comparar. Para ello basta con elegir uno de los *ceros* que aparecen y comparar esos elementos. Supóngase que se selecciona el 0 situado en la posición $M[a,\ b]$ (representado en la figura 6.2) realizando la comparación entre ambos elementos a y b. Según sea el resultado de la comparación las acciones a realizar son diferentes.

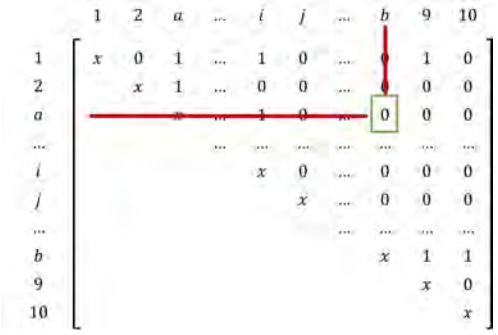


FIGURA 6.2: Se seleccionan 2 elementos a, b para continuar

6.4.1. Resultado de la comparación $a < b$

Si el resultado obtenido en la comparación muestra que el elemento a es menor que b, los elementos se encuentran en orden en la matriz, por lo que no es necesario intercambiarlos pero sí se debe incluir la nueva información obtenida tanto de forma directa como por transitividad.

Con la información obtenida de forma directa el resultado de la comparación introduce un 1 en la posición $M[a, b]$. A continuación se ha de trasladar la información obtenida al realizar la clausura transitiva. Todos aquellos elementos que se saben menores que a también lo son de b. De forma análoga todos los elementos mayores que b son mayores que a. Se traslada toda esta información a la matriz y las acciones a realizar finalizan. (La figura 6.3 muestra en el cuadro rojo la información obtenida por comparación directa y en verde la deducida de la clausura transitiva).

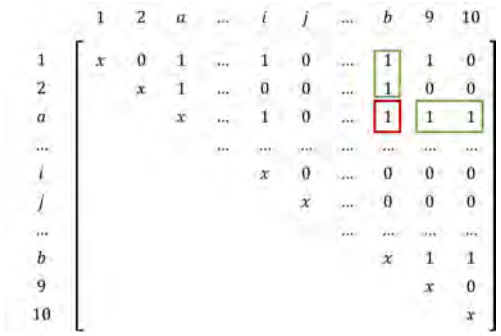


FIGURA 6.3: Se añade la información directa y la clausura transitiva

Si se alcanza un estado final el proceso de ordenación concluye. En caso contrario se debe seleccionar una nueva posición matricial que disponga de valor 0 y continuar con las comparaciones.

6.4.2. Resultado de la comparación $a > b$

En esta situación, dado que sólo se trabaja con la mitad superior de la matriz es necesario realizar una reordenación de los elementos. Existen múltiples posibilidades para realizar el intercambio, por lo que la pregunta a satisfacer es, ¿cuál es la mejor forma de trasladar toda la nueva información con la menor sobrecarga posible para el sistema?.

Previamente se define la situación antes de realizar la comparación. Se denomina $\mathcal{A}$ al conjunto de elementos que se saben que mayores que a y se encuentran situados entre las posiciones matriciales de los elementos a y b. Se denomina $\mathcal{B}$ al conjunto de elementos menores que b y que, igualmente, se encuentran situados entre las posiciones de ambos elementos. Al conjunto de los restantes elementos situados entre a y b de los que se desconoce toda relación se denomina $\mathcal{C}$.

La operación puede ser realizada de múltiples formas, pero aquella que emplea menos recursos es la que conserva para los elementos de $\mathcal{A}$, $\mathcal{B}$ y $\mathcal{C}$ su orden interno inicial.

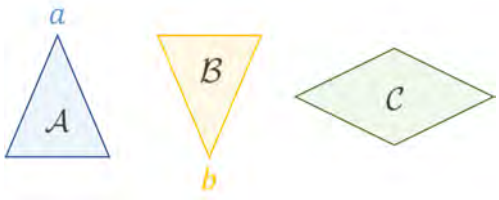


FIGURA 6.4: Conjuntos $\mathcal{A}$, $\mathcal{B}$ y $\mathcal{C}$ que se han de intercambiar

Lo que se debe hacer es permutar a y todo el conjunto $\mathcal{A}$ con b y todo el conjunto $\mathcal{B}$, dejando entre ambos al conjunto de elementos $\mathcal{C}$ de los que no se dispone información alguna que los relacione con a o b. La figura 6.5 muestra una posible situación intermedia, diferenciando con colores los elementos a y b así como los conjuntos anteriormente descritos.

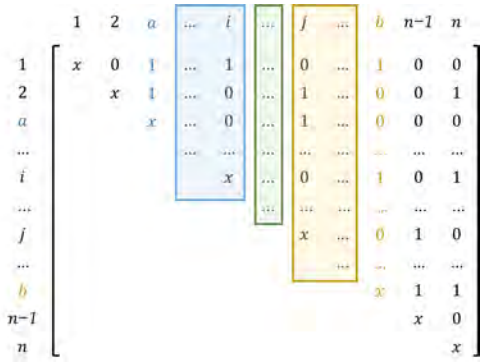


FIGURA 6.5: Diferenciación de conjuntos en la matriz de incidencia

Los elementos situados en posiciones anteriores a a o posteriores a b no se tocan, independientemente de la información que contengan. (En la figura 6.6 las posiciones dentro de la zona roja no se tocan. Las que se encuentran situadas dentro de la zona verde son las candidatas a realizar el intercambio).

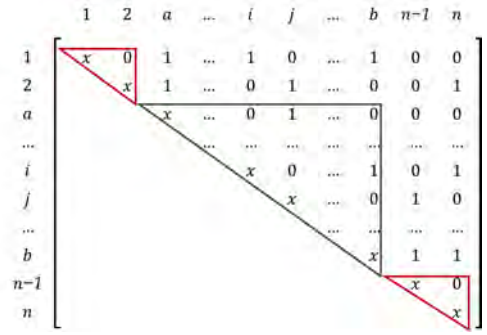


FIGURA 6.6: Zonas que no se tocan (rojo) y sobre la que se trabaja (verde)

Esta forma de trabajo puede sobrecargar el sistema: realizar la permutación "arrastrando intermedios" es una operación laboriosa y con cierta complicación. No obstante representa la esencia más pura para la resolución del problema de ordenación sin imponer ningún tipo de restricción al proceso.

Al realizar la comparación de los elementos a y b, el resultado obtenido es $a > b$, por tanto tras añadir la nueva información y reorganizar los elementos la matriz obtenida presenta el aspecto que se muestra en la figura 6.7.

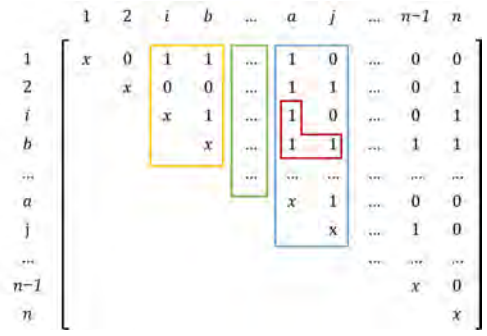


FIGURA 6.7: Elementos tras la comparación y reorganización

Los diferentes colores de la figura 6.7 representan los conjuntos anteriormente descritos: los valores en la zona azul contienen la permutación del elemento a y todo el conjunto $\mathcal{A}$ (se conocía que el elemento j era mayor que a). La zona amarilla contiene la permutación del elemento b y todo el conjunto $\mathcal{B}$ (se conocía que el elemento i es menor que b). La zona verde contiene el conjunto $\mathcal{C}$. Por último los elementos en rojo representan la nueva información obtenida por comparación directa y por transitividad, tras el resultado que se

obtiene al comparar los elementos a y b.

Por resumirlo de forma breve la situación de la que se parte presenta los conjuntos ordenados según muestra la figura 6.8.

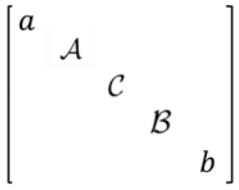


FIGURA 6.8: Situación de elementos y conjuntos antes de la comparación

Tras la comparación el nuevo orden de elementos y conjuntos es el mostrado en la figura 6.9.

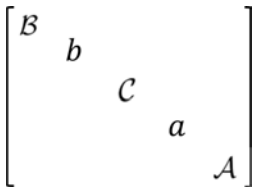


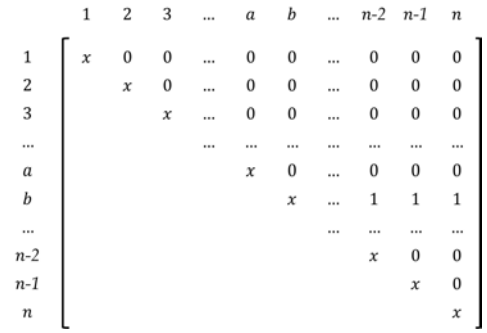
FIGURA 6.9: Situación de elementos y conjuntos tras la reorganización

6.4.3. Algoritmo *Burbuja*

Una buena elección para avanzar hacia la resolución del problema al trabajar con esquemas generales es realizar comparaciones entre hermanos adyacentes.

Curiosamente como esquema general el algoritmo *Burbuja* es la elección que realiza, ya que sólo compara elementos que se encuentran consecutivos en la estructura en que sean almacenados. La restricción de comparar elementos adyacentes simplifica el mantenimiento.

Supóngase que nos encontramos en la situación intermedia que aparece en la figura 6.10, en la que el elemento b ha sido el ganador de las últimas comparaciones.

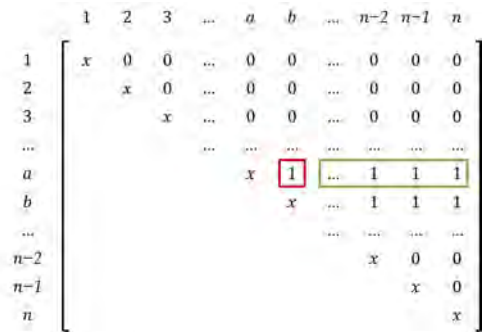


	1	2	3	...	a	b	...	n-2	n-1	n
1	x	0	0	...	0	0	...	0	0	0
2		x	0	...	0	0	...	0	0	0
3			x	...	0	0	...	0	0	0
...				...	...	...	...	...	...	...
a					x	0	...	0	0	0
b						x	...	1	1	1
...							...	...	...	...
n-2								x	0	0
n-1									x	0
n										x

FIGURA 6.10: Algoritmo *Burbuja*: situación intermedia

El siguiente elemento con el que realizar la comparación es el elemento a. Las posibilidades que pueden presentarse son las siguientes:

- Tras la comparación $a < b$. Se añade la información obtenida por comparación directa y además se completa por transitividad la información conocida de que todos los elementos menores que b lo son también de a (La nueva distribución matricial se representa en la figura 6.11).



	1	2	3	...	a	b	...	n-2	n-1	n
1	x	0	0	...	0	0	...	0	0	0
2		x	0	...	0	0	...	0	0	0
3			x	...	0	0	...	0	0	0
...				...	...	...	...	...	...	...
a					x	1	...	1	1	1
b						x	...	1	1	1
...							...	...	...	...
n-2								x	0	0
n-1									x	0
n										x

FIGURA 6.11: Si $a < b$, se añade toda la información conocida

- Si tras la comparación $a > b$, entonces ambos elementos se permutan añadiendo la nueva información obtenida tras la comparación (la nueva situación se refleja en la matriz de incidencia representada en la figura 6.12).

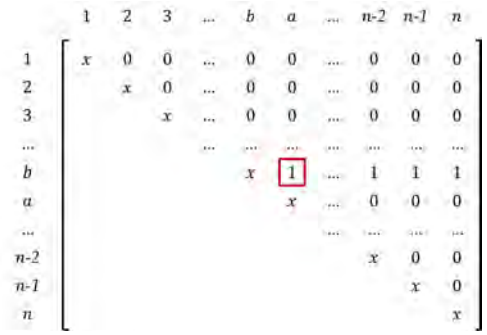


FIGURA 6.12: Si $a > b$ se permutan elementos

6.5. Esquemas de árboles

Para que la tarea sea realizada mediante operaciones más sencillas se restringe el universo de posibilidades de forma que sin afectar en absoluto a la resolución del problema y manteniendo un elevado grado de libertad en la toma de decisiones, permita que las estructuras intermedias manejadas sean más sencillas que la situación precedente que manejaba grafos.

Los esquemas con árboles restringen las posibles comparaciones a esta familia que contiene toda la información conocida en un determinado instante. Puesto que los árboles son relativamente fáciles de manejar, solamente se han de determinar las acciones que son eficaces y que una vez añadida la información obtenida tras cada comparación mantengan la estructura dentro de esta familia.

Esto se puede obtener limitando las comparaciones a elementos que figuren como hermanos dentro de un árbol (o a raíces de árboles que pueden verse como hermanos de primer nivel de un árbol global nocional del que todos los árboles son subárboles) y la incorporación del resultado de la comparación que consiste en podar un subárbol injertándolo como hijo de su hermano[15], mantiene la estructura de árbol conservando toda la información.

La situación final en que ya no hay hermanos que comparar es la de una cadena (árbol degenerado) que representa el objetivo.

El meta-algoritmo ascendente representa una situación particular de trabajo con esquemas de árboles. Si obviamos la estructura de datos utilizada por dicho meta-algoritmo, el proceso de fusionar árboles o cadenas de elementos[1]

[1]Una cadena de elementos no es más que una particularización de un árbol genérico que se encuentra completamente degenerado.

no es más que una abstracción del propio meta-algoritmo.

6.5.1. Ejemplo con elementos almacenados de forma secuencial

Cuando el almacenamiento de la información se encuentra en dispositivos secuenciales, la forma de trabajo con árboles ha de ir encaminada a minimizar el impacto de las operaciones de intercambio de elementos, pues su coste es muy elevado para el proceso.

Si se utiliza memoria encadenada el mantenimiento de la estructura de árbol es sencillo: se basa en realizar sucesivas podas e injertos que al trabajar directamente con memoria pueden ser realizadas en cualquier lugar. Insertar elementos al final o al principio del árbol presenta la misma complejidad, aunque suele ser preferible realizarlos sobre el primer hijo de cada árbol por tener almacenada la dirección de dicho elemento.

En una organización secuencial de árboles en general es necesario disponer de una información estructural adicional (almacenar el número de hijos y el número de descendientes o la dirección del último descendiente, son formas sencillas de disponer de la información con un mantenimiento reducido). Mantener el número de descendientes (o la posición del último descendiente) son pocas instrucciones más que pueden resultar muy útiles según la situación que se pretenda resolver. Son operaciones muy sencillas que mantienen la información redundante y consistente.

La organización en que se dispone el árbol es secuencial en preorden (cada nodo es menor que todos sus hijos que están secuenciados,[2] pero no tienen por qué encontrarse ordenados).

Para almacenar la estructura se exige que cada nodo tenga como información estructural adicional:

- El número de hijos.
- El número de descendientes (número de nodos [incluido él] en el subárbol).
- La dirección del último descendiente.

Ambas estructuras con la información son redundantes y pueden ser deducidas (construidas) una a partir de la otra. Según el caso en que sea necesaria su aplicación, una u otra opción facilitan la operación en curso (como

[2] Árbol secuenciado: en lenguaje de árboles es ordenado, pero al ser confuso se utiliza la palabra secuenciado.

sus mantenimientos son poco elaborados, disponer de 2 e incluso de las 3 opciones no degrada la eficiencia de mantenimiento y mejora la del proceso).

La búsqueda de candidatos para comparar comienza desde la raíz hasta encontrar un nodo que tenga al menos 2 hijos, que son candidatos para ser comparados. La comparación se realiza entre hermanos (sean o no consecutivos). Elegir hijos consecutivos del nodo raíz no es obligatorio, pero facilita enormemente las cosas: puede asegurarse que es una buena elección.

A continuación se muestra un ejemplo con las posibilidades que se pueden presentar. Supongamos la estructura secuencial que se muestra en la figura 6.13.

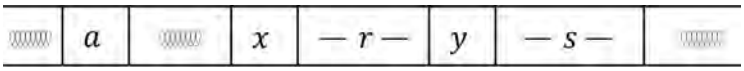


FIGURA 6.13: Ejemplo de organización secuencial

La estructura con forma de árbol que representa dicha situación se corresponde con la imagen de la figura 6.14.

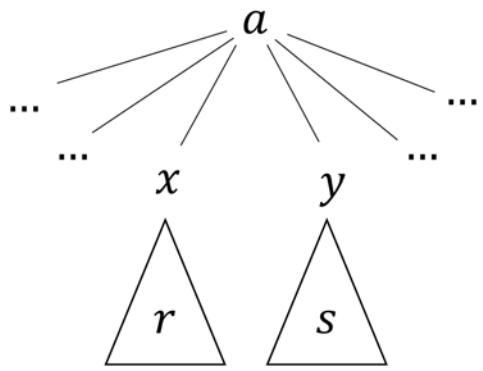


FIGURA 6.14: Organización secuencial expresada en forma de árbol

Se va a proceder a comparar para el árbol con raíz en a entre sus hijos consecutivos los elementos x e y (la información contenida en ellos). Se denomina r al subárbol que se encuentra bajo x y s al subárbol que se encuentra bajo y.

6.5.1.1. Primera posibilidad: $x < y$

Si al realizar la comparación de los elementos x e y se obtiene que $x < y$, se ha de pasar a una situación en que el elemento y y todo su subárbol s pasan

a ser hijos de x, ya que se ha de mantener la relación en preorden. Para ello el padre de y (el nodo a) pierde un hijo y el nodo x lo gana. El elemento y es añadido como último hijo de x por comodidad, pero puede ser añadido en cualquiera de las posiciones que ocupan actualmente los hijos de x.

Al tomar la decisión de que sea el último hijo de x, en secuencial no es necesario mover ningún elemento. Basta con actualizar la información estructural de los hijos.

- a.NumeroDeHijos := a.NumeroDeHijos - 1
- x.NumeroDeHijos := x.NumeroDeHijos + 1

  Si además se lleva la dirección del último hijo:

- x.UltimoDescendiente := y.UltimoDescendiente

  Si se lleva el número de descendientes, entonces:

- x.NumDesc := x.NumDesc + y.NumDesc

Tras realizar esta acción, la información secuencial almacenada no ve alterada su posición[3] y por tanto será idéntica a la mostrada en la figura 6.13.

Únicamente se produce una actualización de la información estructural, lo que transforma la situación del árbol mostrado en la figura 6.14 a la nueva situación que se muestra en la figura 6.15.

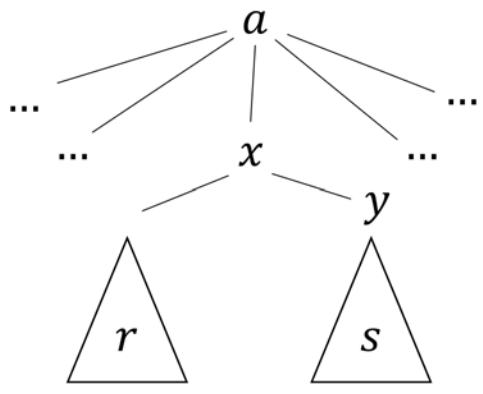


FIGURA 6.15: Organización del árbol tras incorporar la información $x < y$

[3]Esto es consecuencia de haber insertado el elemento y como último hijo de x. En cualquier otra posición es necesario realizar trasposiciones de elementos.

6.5.1.2. Segunda posibilidad: $x > y$

En este caso tras realizar la comparación se obtiene que el menor elemento de los dos es y, por lo que para proceder a introducir la nueva información en la estructura se hace necesaria una reorganización de los elementos secuenciales: x con todo su subárbol ha de pasar a ser hijo de y.

Volviendo a la situación inicial, supongamos que la información almacenada se dispone según representa la figura 6.13. Al ser el elemento x mayor que el elemento y, x (junto con todo su subárbol) ha de pasar a ser hijo de y. En esta situación se dispone de dos posibles alternativas, como muestra la figura 6.16. En la primera alternativa x junto a todo su subárbol pasan a ser el último hijo de y. Por contra, en la segunda alternativa pasa a ser el primer hijo de y.

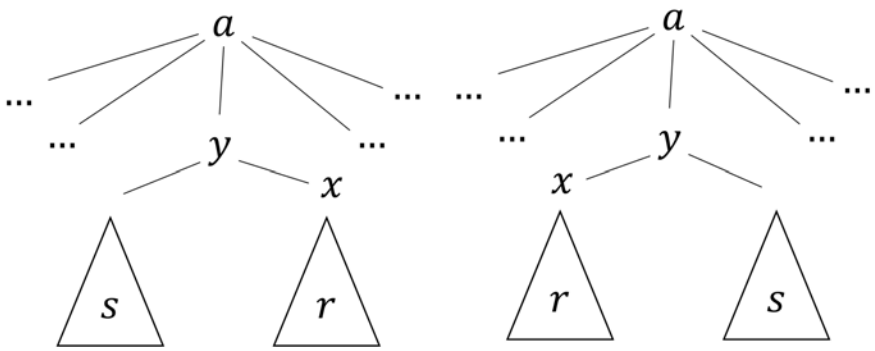


FIGURA 6.16: Posibles situaciones del árbol tras incorporar $x > y$

Si se utiliza la primera alternativa, en la información secuencial almacenada se ha de permutar el elemento x con todo su subárbol r, con el elemento y y todo su subárbol s. La imagen de la situación final de la información tras la permutación se muestra en la figura 6.17.

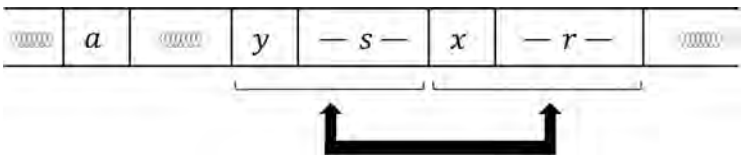


FIGURA 6.17: Información: situación final utilizando la primera alternativa

La situación final de los elementos utilizando la segunda alternativa necesita alterar la posición del elemento y, que ha de pasar a estar situado justo

antes que el elemento x. El resto de elementos (y, subárbol r y subárbol s) han de ser desplazados hacia la derecha tantas posiciones como ocupe el elemento y (la figura 6.18 muestra de forma gráfica la situación descrita).

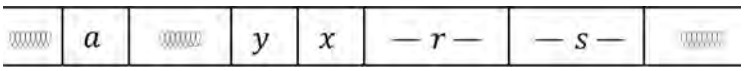


FIGURA 6.18: Información: situación final utilizando la segunda alternativa

En ambas situaciones se mantiene la estructura secuencial. El mantenimiento de la información estructural ha de ser retocado, actualizando el número de hijos en cada caso, en la forma que proceda.

Para ambas situaciones el nodo a pierde un hijo, hijo que en este caso gana el elemento y. Por tanto la actualización de la información estructural es común a las dos situaciones:

- a.NumeroDeHijos := a.NumeroDeHijos - 1
- y.NumeroDeHijos := y.NumeroDeHijos + 1
- y.NumDesc := y.NumDesc + x.NumDesc

Si la información que se lleva almacenada contiene la posición del último descendiente, en la primera situación:

- y.UltimoDescendiente := x.UltimoDescendiente

Para la misma información pero basado en la alternativa de la segunda situación:

- y.UltimoDescendiente no varía ya que el elemento x es colocado como primer hijo de y.

6.5.2. Conclusiones

El trabajo con esquemas de árboles no tiene por qué ser complicado ni costoso. El mantenimiento que se ha de realizar en los esquemas incluso genéricos no tiene por qué ser especialmente oneroso, como se ha mostrado.

Realizar el almacenamiento de la información de la estructura arbórea en preorden no es una condición necesaria: se puede utilizar cualquier otro tipo de orden y la estructura sigue siendo igualmente válida. Se ha elegido preorden por comodidad al considerarse el más intuitivo para realizar el análisis.

6.6. Esquemas de cadenas

Si se realiza una restricción más y se limita la consideración de estructuras únicamente a cadenas (que no son más que árboles completamente degenerados) y elementos simples (vistos como cadenas unitarias), la visión que se obtiene se aproxima (realmente es idéntica) a la fase de fusión del meta-algoritmo ascendente (y al propio algoritmo de *Fusión*).

De hecho una vez finalizada la primera fase del meta-algoritmo ascendente, su tarea para la segunda fase consiste en realizar fusiones sucesivas de cadenas en una cadena más larga hasta llegar al proceso final, que contendrá una única cadena formada por un número igual de elementos (eslabones) a los que contuviese el conjunto de entrada $\mathcal{E}$.

La gestión de estos esquemas es muy simple y si se mira la eficiencia, la forma de realizar esas fusiones (su descomposición en operaciones más elementales) es la que determina el rendimiento final.

6.7. Esquemas de cadena única

Maneja una única cadena y una colección de elementos simples. La determinación de esta estructura de datos da lugar a los meta-algoritmos descendentes.

Los elementos son insertados en la cadena hasta alcanzar la cadena final que los contiene a todos cumpliendo la relación de orden.

Son los esquemas más simples de todos. Los algoritmos *Binario* y *Quicksort* pueden considerarse dentro de este grupo.

En el algoritmo *Binario* la cadena se va construyendo a medida que se construye el árbol de búsqueda, y cada nuevo elemento que se inserta realmente lo que está haciendo es insertar un elemento en la cadena ya construida.

En el algoritmo *Quicksort* el proceso es idéntico aunque su visión no es tan clara como en *Binario. Quicksort* construye la cadena mediante la elección de los pivotes. En un instante determinado la cadena en la que insertar nuevos elementos está formada por todos los pivotes seleccionados hasta ese instante, y los elementos que restan por ordenar (cada uno de los subconjuntos) son los eslabones que se insertan en la cadena. Estos son insertados bien al ser elegidos como pivotes, o bien cuando sólo reste un elemento por ordenar, instante en el que pasa a formar parte de la cadena construida.

6.8. Una jerarquía

Esta serie de esquemas constituye una jerarquía en el sentido de que cada uno es un caso particular más restringido del anterior:

Grafos $\supset$ árboles $\supset$ Cadenas $\supset$ Cadena-simple

Y la inclusión no es sólo de las estructuras que maneja, sino también sus acciones, que son un caso particular de las acciones más generales de la familia que la incluye.

6.9. Acciones

Las acciones tienen como objetivo principal incorporar al esquema la información obtenida por una comparación. Por lo tanto cada comparación que se produce no ha de haberse realizado con anterioridad, ni ha de poder ser deducida (por transitividad) de las comparaciones ya realizadas. La violación de la primera condición puede dar lugar a un bucle sin salida, mientras que la violación de la segunda da lugar a acciones perfectamente inútiles.

La primera condición garantiza que el objetivo final se alcanza con un número finito de acciones, mientras que la segunda sólo pretende que cada acción aproxime la situación actual al objetivo final.

Refiriéndose al índice de ignorancia, una pregunta de respuesta no deducible disminuye el índice de ignorancia. Y una pregunta es tanto más eficiente cuanto más disminuya ese índice. No obstante esa eficiencia es local en el sentido de que un beneficio puntual puede impedir un óptimo global. La situación es compleja en el sentido de que no garantiza que un óptimo global esté formado por óptimos locales.

Las preguntas inútiles no son sólo un despilfarro de tiempo. Existe la posibilidad de que una sucesión de acciones inútiles se constituya en un bucle del que sea imposible salir.

En los algoritmos que incluyen estas acciones se sufre (además de la pérdida de eficiencia) la necesidad de justificar que esos bucles no se producen, recurriendo a razonamientos absolutamente singulares y anecdóticos.

Capítulo 7

Miscelánea

La simplicidad es un prerrequisito
para la fiabilidad.

Edsger W. Dijkstra

7.1. Dualidad entre meta-algoritmos

Como se ha descrito en capítulos precedentes, cualquier proceso de ordenación de los algoritmos clásicos conocidos está contenido en uno de los anteriores meta-algoritmos descritos.

Aunque ambos se han definido y presentado como si se tratase de dos abstracciones de algoritmos de ordenación independientes (puede pensarse por la nomenclatura que incluso antagónicos), en la práctica los puntos divergentes que se encuentran en sus diferentes formas de trabajar se convierten precisamente en puntos en común, si son vistos bajo un prisma que trate de buscar la simetría o dualidad entre ambos procedimientos.

Desde el punto de vista único y exclusivo de una implementación final, poco parece que tengan en común el algoritmo *Burbuja* con el algoritmo *Quicksort*, y si se va un paso más allá, menos todavía con el *Fusión*. Sin embargo son muchos más los puntos comunes que les unen que aquellos que les separan: son más las similitudes o simetrías que muestran sus procesos de ordenación que la separación que se produce en la cristalización de cada implementación.

7.1.1. Dualidad en el propio proceso

A modo de breve resumen de los meta-algoritmos descritos en los capítulos 4 y 5, se recuerda que la tarea común a realizar para ambos es la de hallar

una relación de orden total entre los elementos del conjunto de entrada, asociando a cada elemento una posición final en la citada relación. Los procedimientos locales han de responder a uno de estos dos planteamientos:

- Dado un elemento del conjunto, hallar el lugar que le corresponde.
- Dada una posición, encontrar el elemento que ha de ocuparla.

Su forma de trabajo, lejos de ser opuesta, se puede calificar como de simétrica o dual: el meta-algoritmo descendente fija el elemento y busca su posición final. El meta-algoritmo ascendente fija la posición y busca el elemento que ha de ocuparla.

Para ambos meta-algoritmos se distinguen tres acciones sucesivas comunes y hasta cierto punto solapadas en el proceso de ejecución:

- La acción de distribuir elementos.
- La acción propiamente dicha de ordenar elementos.
- La acción que recoge el resultado.

A continuación se analizan estas fases para cada uno de los meta-algoritmos.

7.1.2. Descendente

En el meta-algoritmo descendente los procedimientos que resuelven el problema son descritos como de descomposición de este en otros problemas menores. Cada paso nos acerca a la solución final dividiendo un problema de gran envergadura en otros de menor tamaño y mayor facilidad de resolución. Puede decirse que se sitúan encuadrados dentro de una dinámica de bifurcación y progreso hacia acciones menores. Las diversas formas de materializar dichas acciones es lo que da lugar a la variedad de representaciones finales.

Realizada la descomposición del problema, cada subproblema se trata de forma independiente. Esto permite elegir entre diversas estrategias de paralelismo o su secuencialización. La característica principal de estos procedimientos es, por tanto, la bifurcación.

El procedimiento completo consta de dos fases:

- En una primera fase se solapa la distribución con la ordenación de elementos.
- En la segunda fase se realiza la recuperación de los resultados.

7.1.3. *Quicksort* y *Binario*

Las semejanzas subyacentes entre *Quicksort* y *Binario* son manifiestas.

Ambos algoritmos seleccionan un elemento y lo comparan con todos los demás, dividiéndose el resto de elementos en dos subconjuntos: los menores al elemento seleccionado (subárbol izquierdo) y los mayores que este (subárbol derecho).

La única diferencia radica en que *Quicksort* realiza todas esas comparaciones antes de poder continuar, mientras que *Binario* procesa un elemento y hasta que este no encuentra su lugar no pasa a procesar el siguiente elemento.

Otra forma de analizar esta situación es considerar a *Quicksort* como recursivo y a *Binario* como iterativo. Curiosamente *Quicksort* es clasificado como de intercambio (a veces de partición) y *Binario* es clasificado como de inserción.

7.1.4. Ascendente

De manera análoga este meta-algoritmo construye soluciones parciales locales que deben ser combinadas de forma ascendente. Al combinar problemas menores que se tratan de manera independiente, la posibilidad de paralelismo se presenta en la fase ascendente y su principal característica es la de sincronización.

En los primeros pasos no es posible llevar a cabo el paralelismo, ya que para iniciar la composición de soluciones parciales es necesario que previamente estas hayan sido construidas.

Al igual que el descendente, el procedimiento completo también consta de dos fases:

- En la primera fase realiza la distribución de elementos.
- En una segunda fase se solapan la ordenación y la recuperación de resultados.

7.1.5. Dualidad en la forma de abordar el problema

El pequeño resumen realizado sobre la forma de trabajo de ambos meta-algoritmos muestra de forma clara una simetría o dualidad en la metodología utilizada para resolver el problema.

En ambos aparece reflejada la máxima de *divide y vencerás*: en el caso del meta-algoritmo descendente se alcanza el objetivo mediante sucesivas bifurcaciones. En el meta-algoritmo ascendente mediante fusiones, lo que muestra una dualidad en la forma de trabajar: *bifurcación vs fusión*, dos metodologías análogas (duales) que persiguen el mismo fin.

Dicho de otra forma, todos los algoritmos que se incluyen dentro del meta-algoritmo ascendente sincronizan subproblemas, mientras que aquellos que se incluyen dentro del meta-algoritmo descendente bifurcan en subproblemas.

7.1.5.1. Asimetrías

Continuando con un análisis exhaustivo sobre la forma de trabajo de ambos meta-algoritmos, en este apartado se muestran las asimetrías que se producen en ambos procesos de ejecución: aspectos fundamentales que muestran como el punto fuerte de un meta-algoritmo se convierte en el punto débil de su homólogo, y viceversa.

7.1.5.2. Inserción de elementos

Cuando el proceso de ordenación se encuentra en plena ejecución, con independencia del meta-algoritmo seleccionado para realizar el trabajo, es muy probable que en la entrada aparezcan nuevos elementos para tratar que han de ser incorporados al proceso.

En el caso del meta-algoritmo descendente no afecta para nada a la complejidad del procedimiento. Los nuevos elementos son tratados y distribuidos en el árbol de procesos en el lugar que les corresponda, lo que muestra una alta adaptabilidad a las nuevas circunstancias.

Por contra, en el meta-algoritmo ascendente sí que afecta este nuevo aspecto a la ejecución, puesto que los nuevos elementos es posible que hagan que la "equipartición" no sea la deseada y por lo tanto afecta de forma directa a la eficiencia del proceso.

7.1.5.3. Fase de realización del trabajo

En el meta-algoritmo descendente la totalidad del proceso presentará más eficiencia cuanto más equilibrada sea la distribución de elementos o, lo que es lo mismo, cuanto más equilibrado sea el árbol de procesos. Pero ese equilibrio viene condicionado por el orden inicial de los elementos y no es controlable. Y si bien se puede proceder a reequilibrar el árbol en pleno proceso, no hay garantías de que perdure y no sea destruido por los elementos que aparezcan a continuación.

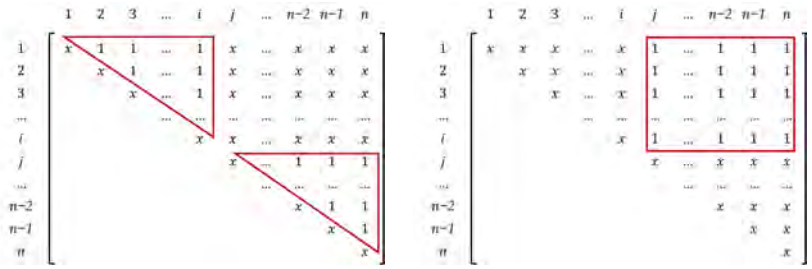
Por contra, en el meta-algoritmo ascendente la primera fase o *fase descendente* es la encargada de realizar la división del conjunto de entrada. Por tanto en este caso la eficiencia depende de la equipartición, que en cierta medida es controlable por el proceso y es lo que hace que la distribución de la

complejidad sea más concentrada.[1]

El meta-algoritmo ascendente requiere una distribución inicial de los elementos, mientras que el meta-algoritmo descendente requiere una recopilación final para obtener el resultado. Son pequeños detalles que muestran como la forma de trabajo de ambos, aunque parezca alejada, realiza tareas duales en diferentes momentos de la ejecución según la situación que se aborde.

Si se centra la atención en una cristalización definitiva de los meta-algoritmos comparando *Fusión* (ascendente) con *Quicksort* (descendente), se puede afirmar que *Fusión* es mejor que *Quicksort* si se analiza únicamente el número de comparaciones que se realizan. Sin embargo no es menos cierto que el éxito siempre se lo ha llevado *Quicksort*. Se justifica esta falta de utilización de *Fusión* en que para su implementación (por norma general) se necesita el doble de memoria, lo que hace que no sea deseable su uso en entornos con recursos limitados (algo habitual porque cuando se dispone de innumerables recursos en el sistema informático es frecuente que el número de elementos a tratar sea proporcionalmente mayor).

La prueba de esa asimetría (con cariz de dualidad) entre ambos algoritmos se muestra en la matriz de incidencia en dos momentos opuestos de su respectiva ejecución. En la figura 7.1, la imagen de la izquierda representa la matriz de incidencia del algoritmo *Fusión* justo antes de realizar la última fusión. La imagen derecha representa la situación para el algoritmo *Quicksort* justo después de realizar la primera división.



(A) (B)

FIGURA 7.1:
(A) *Merge* antes de la última fusión
(B) *Quicksort* tras la primera división

[1] Se elige el punto por el que realizar la división de cada conjunto en dos subconjuntos de menor tamaño. Siempre que se tenga posibilidad de elección la mejor alternativa es seleccionar el punto medio, como se demuestra en el apéndice D

Si nos fijamos en ambas imágenes se puede afirmar con rotundidad que ambos algoritmos trabajan de forma dual, de forma simétrica en momentos antagónicos de su ejecución.

7.2. Un ejemplo práctico: concordancias

En este apartado se va a realizar un análisis de un problema real de los más habituales, en el cual se hace necesario llevar a cabo la ordenación de un conjunto de elementos cuyo tamaño en el instante inicial es desconocido. Se trata de un ejemplo de adaptación desde una forma "pura" a una circunstancias particulares.

No es infrecuente que en numerosas tareas de cualquier sistema de información sea necesario un procedimiento de ordenación que atienda a unas circunstancias especiales, las cuales presentan exigencias particulares.

Dichas exigencias en unas ocasiones imponen una carga adicional al sistema que es más eficiente incorporar al propio proceso de ordenación, y en otras ocasiones simplifican en parte la tarea.

7.2.1. Análisis inicial

El problema a abordar (problema muy habitual) es el de las concordancias. ¿En qué consiste la tarea a realizar?. Supongamos que se facilita un documento de texto de la naturaleza y tamaño que sea, y se pretende realizar un análisis y obtener una serie de datos sobre el mismo: orden de las palabras, número de repeticiones de cada una, posición que ocupa cada palabra en el documento, etc. ¿Cómo se aborda la cuestión?.

El punto de vista más abstracto consiste en considerar que hay claves repetidas (la palabra) y que lo que se ordena es el conjunto de los conjuntos de elementos con la misma clave. Por ejemplo en *Binario* el pivote es el conjunto de las apariciones de la misma palabra que se inicializa con la primera aparición y se va construyendo con las apariciones sucesivas de la misma. Los detalles particulares a partir de este punto han de satisfacer los propios requisitos particulares.

En un primer análisis del problema se identifican los siguientes elementos:

- Como productor, el texto que se facilita (del que se obtiene el conjunto de elementos de entrada).
- Como consumidor, aquel proceso que solicita la información adecuadamente ordenada.

- En medio de ambos, el algoritmo de ordenación encargado de tratar el conjunto de entrada (facilitado por el productor) y enviar los resultados esperados al consumidor.

7.2.2. Situaciones

Unas situaciones típicas que se suelen dar son:

- La creación de un índice de un texto, en que el algoritmo va leyendo el texto detectando y generando los elementos a ordenar compuestos de (palabra, lugar de aparición) y que han de ser ordenados por la palabra.
- Calcular las frecuencias de las palabras de un texto: situación parecida a la anterior, pero en esta caso una palabra tiene asociado un contador de frecuencia y el módulo de ordenación debe mantener esos contadores, así como generar la lista ordenada de las palabras con su frecuencia.
- Crear unas concordancias: similar al primer ejemplo pero tratando todas (o casi) las palabras del texto.

Para cualquiera de estos casos, llevar a cabo la solución por fuerza bruta es poco eficiente en tiempo y desde luego completamente ineficiente en espacio. Si se piensa en textos de varias páginas, la solución por fuerza bruta pudiera incluso funcionar con las citadas limitaciones, pero si se piensa en analizar una biblioteca completa entonces la fuerza bruta se califica simplemente como inaceptable.

Para abordar este problema se debe violar la regla de unicidad de claves vista en el apartado 3.5.1, puesto que puede haber varios elementos iguales en la lista a ordenar. Aunque la condición al menos en las dos primeras situaciones es adaptable:

- En la primera situación, cuando se necesita crear el índice de un texto el elemento repetido simplemente es ignorado.
- En la segunda situación, cuando es necesario conocer la frecuencia de cada palabra, lo realmente relevante es incrementar el contador de frecuencia con cada repetición encontrada.

Lógicamente el tercer caso planteado necesita diferenciar las palabras repetidas, puesto que si se pretende crear un índice con todas las palabras del texto no se admite ningún tipo de concesión.

7.2.3. Posibilidades de solución

En un primer análisis del problema, parece lógico no pensar en la utilización de estructuras rígidas dada la naturaleza de la tarea a realizar. Es una buena idea la utilización de estructuras dinámicas y flexibles al ser desconocido el tamaño del conjunto de elementos de entrada $\mathcal{E}$, así como la temporalidad con la que estos son "entregados" al algoritmo de ordenación.

Por idéntico motivo, aquellos procesos que requieran para comenzar su ejecución de todos los elementos a tratar en su entrada, no parecen una buena alternativa de solución al problema. Además no es mala idea permitir en el mayor grado posible el concepto de paralelismo, lo que parece recomendar no decantarnos por aquellos algoritmos que restringen las comparaciones entre elementos a los meramente adyacentes.

Igualmente parece claro que ante la posibilidad de un conjunto de elementos a tratar de tamaño desconocido, aquellos algoritmos que requieren mayor espacio de almacenamiento para llevar a cabo el proceso no son los idóneos.

Por otro lado utilizar una cristalización de meta-algoritmo descendente como *Quicksort* o similar, impone que el proceso inicial no da paso a sus procesos hijo hasta haber tratado por completo todos los elementos de la entrada. Nadie garantiza que al inicio de la ejecución se disponga de todos los elementos a tratar, por lo que utilizar cualquier algoritmo que imponga esta restricción representa un lastre para el proceso.

El análisis de la situación nos dirige a pensar como posible solución en el algoritmo *Binario* o el meta-algoritmo descendente. Ambos candidatos se adaptan perfectamente a este tipo de situaciones con las siguientes adaptaciones:

1. En primer lugar, las comparaciones han de distinguir los tres casos de menor, igual y mayor. Los casos menor y mayor siguen el modelo original.

2. En el caso de las frecuencias, al pivote se le asocia un contador que se inicializa a 1 cuando se toma el elemento como pivote y se incrementa cada vez que es comparado con un elemento igual.

3. En el resto de los casos, al pivote se le asocia una lista con los lugares de aparición que se actualiza con la incorporación de un nuevo lugar de aparición cuando el pivote sea comparado con un elemento que posea idéntica palabra.

Por norma general la lista construida se ha de proporcionar ordenada, tarea que se puede hacer posteriormente o (lo más habitual) si los elementos se generan con los lugares de aparición ordenados, basta con procesarlos por

ese orden[2] y la lista de lugares de aparición se actualiza añadiendo el nuevo lugar al final de la misma.

7.2.4. Conclusiones

Para resolver un problema de ordenación de elementos, en principio, se puede utilizar cualquier algoritmo puesto que todos ellos cumplen eficazmente con la tarea encomendada. Lógicamente la eficiencia y utilización de recursos (así como su disponibilidad) no resulta ser la misma en unos y otros casos.

Con este ejemplo se pretende mostrar como circunstancias externas ajenas al problema a resolver y completamente independientes del concepto de ordenación en sí mismo, dan preferencia a un par de algoritmos concretos permitiendo obtener una solución más clara y sencilla tanto en el proceso de ejecución como en el mantenimiento de las estructuras de datos.

Para resolver el problema de las concordancias es necesario adaptar la estructura del propio meta-algoritmo descendente incorporando información del problema a resolver. Esta es la mejor solución posible y la que sin lugar a dudas introduce menor sobrecarga en el sistema, evitando incluso la utilización de recursos extra de almacenamiento.

Si se analiza el procedimiento desde el punto de vista de la eficiencia, con toda seguridad una ejecución de procesos mediante el meta-algoritmo ascendente con una estructura de control similar a la empleada por el algoritmo *Fusión* produce la ordenación más eficiente de todas, pero su excesivo requerimiento de espacio de almacenamiento así como el mantenimiento que se deriva hacen que sea descartado al no compensar su eficiencia los costes de almacenamiento.

7.3. Análisis exhaustivo de un algoritmo: *Burbuja*

El método *Burbuja* es probablemente el más conocido de los algoritmos de ordenación debido a su simplicidad, lo que hace de él un candidato idóneo para ser utilizado como ejemplo o ejercicio en los cursos de introducción a la programación.

Su atractivo se complementa con la facilidad de verificación de corrección. Sin embargo no puede decirse que destaque por su eficiencia.

El algoritmo se basa en la utilización de una estructura de datos secuencial (un vector por ejemplo) y en una operación única muy simple: comparar elementos adyacentes que eventualmente son intercambiables.

[2] En el meta-algoritmo descendente las entradas a los procesos son ahora colas.

7.3.1. Versión clásica

La descripción del algoritmo clásico *Burbuja* junto con el pseudocódigo se encuentran en el apéndice G.

Utilizando el lenguaje de programación *ADA*, la codificación del pseudocódigo para la versión clásica es la siguiente:

```
Entrada: Conjunto de n elementos.
Salida:  n elementos cumpliendo una relación de orden total.
for i in 1..n-1 loop
     for j in reverse i..n-1 loop
          if elemento(j) > elemento(j+1) then
               swap(elemento(j), elemento(j+1));
          end if;
     end loop;
end loop;
```

ALGORITMO 1: Algoritmo clásico *Burbuja* en *ADA*

Su gran sencillez de explicación e implementación se ve contrarrestada por su ineficiencia, que se considera baja puesto que realiza $\frac{n(n-1)}{2}$ operaciones frente a otros algoritmos que sólo realizan $\log_2 n$ operaciones.

Esta baja eficiencia se debe a que:

1.- Hace preguntas inútiles de respuesta conocida o deducible. Se puede afirmar que muchas de las preguntas que realiza son poco eficaces.

2.- La inserción de elementos que realiza es secuencial.

3.- Las posibles comparaciones se encuentran limitadas a elementos adyacentes.

Por ejemplo, si la lista se encuentra inicialmente ordenada, de las $\frac{n(n-1)}{2}$ preguntas que realiza sólo son necesarias $n-1$ siendo el resto superfluas.

También puede ocurrir que en algún momento del bucle se alcance el orden total con lo que las restantes comparaciones son innecesarias. Esto sugiere una versión mejorada que se detiene cuando detecta que ya ha alcanzado una situación de orden total.

7.3.2. Versión mejorada

El pseudocódigo de la versión mejorada se encuentra disponible en el anexo G. No obstante dado que los siguientes análisis se realizan sobre dicho código, se muestra a continuación el código *ADA* introduciendo las mejoras.

```
Entrada: Conjunto de n elementos.
Salida:  n elementos cumpliendo una relación de orden total.
for i in 1..n-1 loop
     detener:=true;
     for j in reverse i..n-1 loop
          if elemento(j) > elemento(j+1) then
               swap(elemento(j), elemento(j+1));
               detener:=false;
          end if;
     end loop;
     exit when(detener);
end loop;
```

ALGORITMO 2: Algoritmo *Burbuja* mejorado en *ADA*

Cuando en el bucle interior no se realiza ningún intercambio se detecta el haber alcanzado la relación de orden total, por lo que cualquier acción posterior es superflua: lógicamente la mejor alternativa es detener la ejecución del algoritmo al tener disponible la salida deseada.

7.3.3. Análisis

En cada ejecución del bucle interior el elemento menor de los que están en las posiciones $(i\,..\,n)$ se intercambia con su inmediato anterior hasta situarse en la posición i.

Las condiciones de verificación son así fáciles de definir:

1.- El bucle externo en cada iteración con índice i, tiene como condición inicial que los elementos de $1\,..\,i-1$ son los menores y están ordenados.

2.- El bucle interno sitúa el menor elementos de los que se encuentran en las posiciones $(i\,..\,n)$ en la posición i.

3.- Al finalizar el bucle interno para el índice i, los elementos de $1\,..\,i$ son los menores y están ordenados.

7.3.4. Análisis gráfico

La simplicidad del método se obtiene al precio de una cierta ineficiencia al ignorar las respuestas a las comparaciones que no proporcionen un beneficio inmediato.

En la versión básica se realizan comparaciones que ya se han comprobado en ejecuciones precedentes del bucle interno. Y por otra parte, si en la ejecución

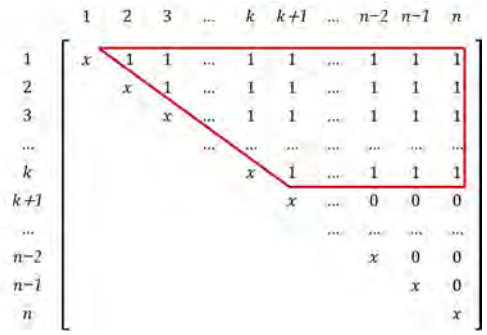


FIGURA 7.2: *Burbuja*: única información almacenada tras k iteraciones

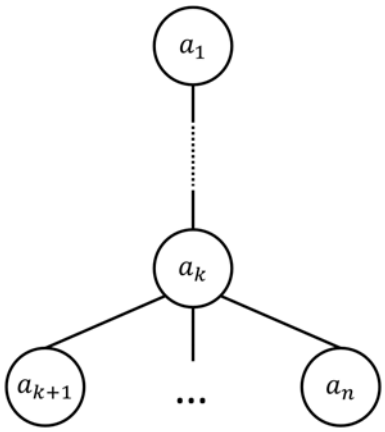


FIGURA 7.3: *Burbuja*: grafo que representa la situación tras k pasadas

para el índice i se obtienen no sólo los primeros i elementos sino algunos más, en ejecuciones posteriores del bucle se realizan comparaciones innecesarias.

El método clásico considera que la información de que dispone al cabo de k ejecuciones del bucle exterior es la que se representa en el gráfico mediante la matriz de incidencia de la figura 7.2.

Realizando la representación como grafo (árbol) la imagen es la representada en la figura 7.3.

Mientras que la información deducible por comparaciones representada en la matriz de incidencia, se muestra en la figura 7.4.

Y el árbol que representa la misma información se corresponde con el de la imagen 7.5.

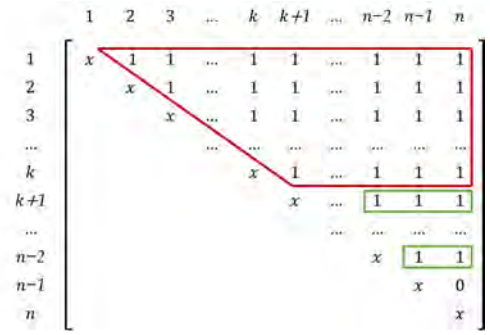


FIGURA 7.4: *Burbuja*: información real obtenida tras k comparaciones

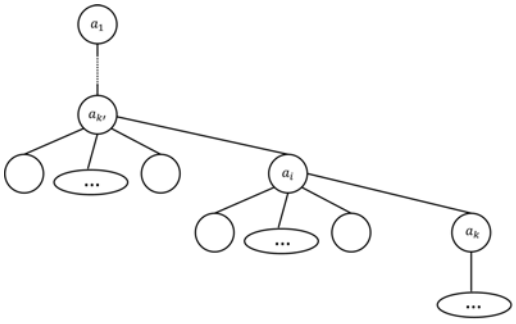


FIGURA 7.5: *Burbuja*: grafo que refleja toda la información obtenida

con $k' \geq k$ como árbol.

Nótese que la ubicación secuencial corresponde a un recorrido del árbol por niveles. O lo que es lo mismo, el orden inverso del preorden. Y que la relación de adyacencia en la estructura secuencial es entre hermanos consecutivos, o entre un nodo y su hijo mayor.

7.3.5. Versión optimizada

La inclusión de toda la información se obtiene mediante una pequeña adición a la estructura de datos utilizada. Se trata de asociar a cada elemento un estado con dos valores posibles (por ejemplo un único bit con valores 0 y 1 respectivamente) que indican para cada elemento si es o no el *hermano más pequeño del nivel del árbol,* en cuyo caso cumple la propiedad de que es menor que todos los elementos posteriores a él (con esta definición el elemento situado en la última posición está siempre en estado 1).

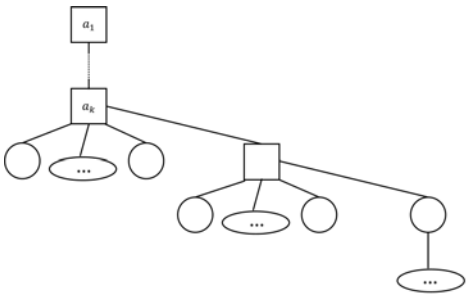


FIGURA 7.6: *Burbuja* optimizada: grafo de una situación intermedia

El gráfico 7.6 refleja una situación en un momento determinado de la ejecución del algoritmo. Los elementos han sido distinguidos de la siguiente forma: aquellos elementos que poseen un estado igual a 0, se representan mediante un ◯ y aquellos cuyo estado es igual a 1 mediante un □.

Al comenzar la ejecución el estado inicial de todos los elementos es 0. El estado cambia a 1 en el momento en que en una comparación resulta ser el menor, estado en el que permanece en adelante puesto que sólo permuta con elementos mayores que él.

La organización secuencial que impone el algoritmo, junto con la condición de comparar elementos consecutivos por orden decreciente de posición (lo que garantiza el mantenimiento de la especial estructura del árbol) muestra las comparaciones inútiles que hace el algoritmo en su versión básica. Por lo tanto sugieren las comparaciones que tiene sentido realizar y aquellas que se han de evitar por innecesarias.

Si se recorre el árbol en el orden exigido por el algoritmo, los elementos que se comparan pueden estar ambos en el estado 1, en cuyo caso la respuesta es conocida de antemano y por lo tanto la comparación es inútil. La otra posibilidad es que se compare un elemento en estado 0 con uno en estado 1. Si el primero es menor se le cambia a estado 1 y si es mayor los elementos son permutados.

En la representación gráfica la figura 7.7 muestra el estado antes de realizar la comparación de Y con Z.

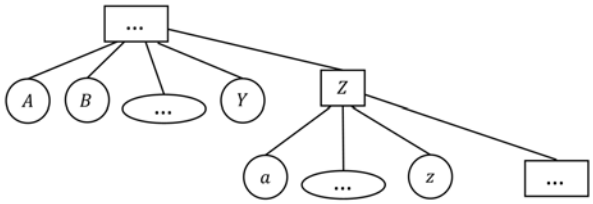


FIGURA 7.7: *Burbuja* optimizada: se compara Y con Z

Si el elemento Y es menor que el elemento Z, el resultado tras la comparación es el mostrado en la figura 7.8.

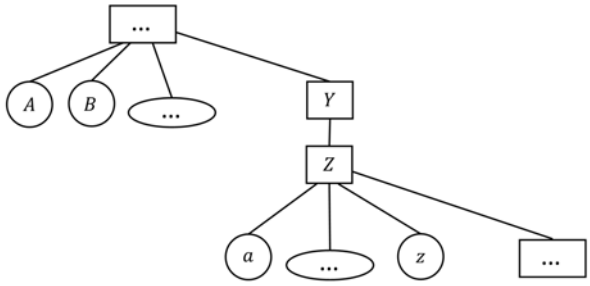


FIGURA 7.8: *Burbuja* optimizada: nueva situación si $y < z$

Por contra si el elemento Y es mayor que el elemento Z, la nueva situación la representa el grafo de la figura 7.9.

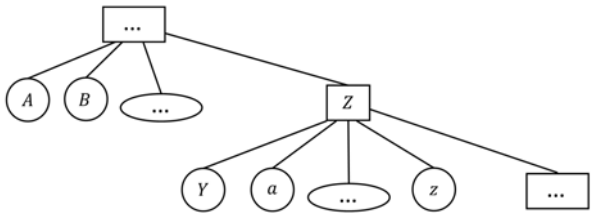


FIGURA 7.9: *Burbuja* optimizada: nueva situación si $y > z$

Con los cambios propuestos, ahora el núcleo del algoritmo es el siguiente:[3]

[3]En el algoritmo, la variable `ultimo` es el último elemento a considerar en un recorrido, ya que los elementos desde la posición 1 hasta `ultimo` ya están ordenados y en su posición.

```
Entrada: Conjunto de n elementos.
Salida:  n elementos cumpliendo una relación de orden total.
for i in 1..n loop
     estado(i):=0;
end loop;

estado(n):=1;
ultimo:=1;
loop
     for j in reverse ultimo..n-1 loop
          if (estado(j)=0) and (estado(j+1)=1) then
               if (elemento(j) > elemento(j+1) then
                    estado(j):=1;
               else
                    swap(elemento(j), elemento(j+1));
                    swap(estado(j), estado(j+1));
               end if;
          end if;
     end loop;
     estado(n):=1;
     z:=ultimo;
     for j in ultimo..n-1 loop
          if estado(j)=0 then
               z:=j;
               exit;
          end if;
          exit when (z=ultimo);
     end loop;
     ultimo:=z;
end loop;
```

ALGORITMO 3: *Burbuja* sin pérdida de información: código en *ADA*

Visto como árbol:

- El elemento raíz y los siguientes, mientras sean hijos únicos, están en el estado 1.
- El hijo menor de cada elemento está en estado 1.
- Los hijos restantes están en estado 0.

Y puesto que el proceso va tratando niveles de abajo hacia arriba, resulta que:

- El primer elemento en estado 0 define el límite de las comparaciones a hacer en cada ejecución del bucle.
- Sólo se hacen comparaciones útiles (de las que se desconoce la respuesta) cuando se compara un elemento en estado 0 con un elemento (consecutivo) en estado 1.
- El proceso termina cuando no hay elementos en estado 0 y por tanto no hay comparaciones para realizar.

Otra posible forma de resumen es la siguiente: sólo se compara un elemento que sea el hermano menor de una fraternidad con su hermano inmediato mayor (si lo hay).

7.3.6. Vista como meta-algoritmo descendente

Gracias a los estados declarados la estructura de datos se maneja como un árbol: los elementos en estado 1 tienen como hijos los elementos inmediatos posteriores en estado 0 hasta incluir un elemento en estado 1.

Los árboles así obtenidos son un de un tipo particular: los hijos de un elemento carecen de hijos a su vez, salvo el menor de ellos.

La consideración como meta-algoritmo descendente se interpreta de la siguiente forma:

- Los elementos en estado 1 son considerados como los procesos del meta-algoritmo descendente.
- El elemento a tratar se considera el elemento pivote del proceso.
- La entrada de un proceso son los hermanos mayores (todos en estado 0) gestionados no como un conjunto de elementos, sino como una cola.

La estructura global es una cadena de procesos en forma de árbol lineal en la que los pivotes de los procesos son los elementos en estado 1 en el orden en que se encuentran dentro de la lista.

La dinámica de este meta-algoritmo resulta ser un tanto anómala. Cuando un elemento es procesado, los dos elementos comparados son:

- Por un lado, un elemento de la entrada del proceso en estado 0.
- Por otro lado, el propio pivote del proceso que realiza la comparación.

Si el pivote es menor que el elemento de la entrada (los elementos se intercambian) se transfiere el elemento a la entrada del proceso hijo (al final de la cola).

Si el pivote es mayor el elemento adquiere el estado 1, lo que se interpreta como que se crea un proceso con ese elemento como pivote, adquiere como entrada los elementos que pudieran quedar en la cola (sus hermanos mayores en estado 0) y ese proceso se intercala en la cadena de procesos como anterior al que se estaba ejecutando.

7.3.7. Vista como esquema de cadena

En el meta-algoritmo descendente anterior, los pivotes de la cadena de procesos (los elementos del árbol que son los hijos menores o que están más a la derecha en cada nivel) son vistos como una cadena en la que se van insertando los elementos restantes.

Cada elemento a insertar ha de encontrar su posición en la subcadena que empieza en su hermano menor y continúa por los hijos menores hasta el último elemento de la lista.

7.3.8. Vista como esquema de árbol

Desde el punto de vista de los esquemas, a partir del primer recorrido por la lista se considera que los elementos están organizados como un árbol que representa toda la información del orden parcial determinado hasta ese momento.

Inicialmente se puede ver así también, como un árbol en que una raíz nocional tiene como hijos todos los elementos a tratar.

En este esquema, las únicas comparaciones permitidas son entre dos hermanos en que el segundo tenga hijos o sea el último de la lista.

Este punto de vista permite ver este método como una implementación particular de un esquema de árboles.

7.3.9. Posibilidades de paralelismo

Si se considera el método *Burbuja* como una variante del meta-algoritmo descendente, todos los procesos con entrada no vacía pueden ejecutarse en paralelo. Es posible comparar elementos adyacentes en diferentes partes de la estructura, que no interfiere en absoluto a la hora de obtener el resultado final y conserva las condiciones de corrección del algoritmo que se trata de paralelizar.

7.3.10. Selección de prioridades

Si por algún requerimiento externo se desea obtener los primeros elementos lo más rápido posible, como la necesidad de ir procesando los elementos obtenidos para que una tarea (proceso) posterior no esté detenida, se modifica el algoritmo mejorado para que en cada bucle interior en vez de iniciar el proceso desde el final de la estructura, lo haga desde el primer lugar posible.

Se ejecuta el proceso de comparación e intercambio desde el primer elemento en estado 1 que esté precedido de un elemento en estado 0 y se repite ese tratamiento hasta que no haya elementos en estado 0.

Visto como árbol el proceso estándar hace las comparaciones desde el final del árbol hacia arriba, mientras que esta modificación corresponde a procesar el nivel superior.

Visto como meta-algoritmo descendente, se da prioridad al proceso que está más arriba en la cadena de procesos.

7.4. Otra versión: *Burbuja* inversa

Hay otra versión bastante difundida del método *Burbuja*, en la que si bien se mantiene el invariante de que después de la ejecución del bucle para el índice i los i primeros elementos están ordenados, no son necesariamente los menores. Y en cada ejecución del bucle se toma el siguiente elemento y se le va permutando con los anteriores mientras sea menor.

```
Entrada: Conjunto de n elementos.
Salida:  n elementos cumpliendo una relación de orden total.
for i in (2..n) loop
     for j in reverse (1..i-1) loop
          if Elemento(j+1) < Elemento(j)
               then swap(Elemento(j), Elemento(j+1));
          end if;
     end loop;
end loop;
```

Algoritmo 4: Algoritmo *Burbuja* inverso: código en *ADA*

En realidad va construyendo una cadena ordenada en los primeros i lugares con los primeros i elementos y va tomando nuevos elementos e insertándolos en esa cadena.

Si la inserción de un elemento en una cadena de longitud p requiere de media $^{n}/_{2}$ comparaciones, la complejidad de esta variante es en media de $\frac{n(n-1)}{4}$, con un mínimo de $n-1$ para el caso de una configuración inicial ya ordenada y un máximo de $\frac{n(n-1)}{2}$ para configuraciones iniciales próximas al orden inverso.

No hace preguntas inútiles, pero al hacer la inserción de los elementos sucesivos en la cadena parcial de forma secuencial (exigido por la restricción sobre las comparaciones permitidas) la eficiencia ideal se degrada.

Pruebas experimentales muestran que la eficiencia de la versión optimizada es equivalente a la de esta versión, del orden de $\frac{n(n-1)}{4}$, lo que no es sorprendente puesto que ambas versiones insertan elementos en cadenas secuencialmente.

7.5. Comparaciones e intercambios

En el análisis de los algoritmos de ordenación se consideran únicamente las comparaciones que se hacen.

Si se desean computar los intercambios en el método *Burbuja*, tanto si se utiliza una versión u otra del algoritmo los intercambios a hacer son los mismos: dos elementos que en la configuración inicial estén invertidos (el menor está en una posición posterior) más pronto o más tarde se enfrentan y son intercambiados, y sólo en ese caso se realizan intercambios.

Por este motivo las variantes que se puedan definir, mientras respeten las restricciones básicas, pueden diferir en el número de comparaciones, pero el número de intercambios es invariable para una configuración inicial dada.[4]

El número de intercambios a realizar es igual al número de inserciones en la permutación representada por la configuración inicial.

7.6. Sus ventajas

Al consistir su operación básica en la comparación y eventual intercambio de elementos almacenados en posiciones consecutivas, los posibles fallos de páginas en la estratificación de memoria se reducen al mínimo, con lo que la sobrecarga del sistema externo se reduce y el rendimiento práctico puede superar las desventajas de un método que no aprovecha al máximo la información obtenida hasta el momento al hacer preguntas no "optimales". Si además se aplican algunas de las modificaciones anteriores, el resultado puede llegar

[4] Concretamente en terminología matemática es el número de inversiones.

a ser mejor que los métodos "tradicionales", sobre todo si los elementos son voluminosos.

7.7. Resumen de las variaciones posibles

Con sólo llevar constancia de los elementos que son menores que todos sus posteriores, información que se obtiene de haber "ganado" una comparación y que en el fondo es la aplicación de la clausura transitiva:

1.- Se evitan comparaciones inútiles, entendiendo como tales aquellas comparaciones ya realizadas o cuyo resultado es deducible (por transitividad) de las realizadas con anterioridad.

2.- Si se desea dar prioridad a la determinación de los primeros elementos, basta con dar preferencia a las comparaciones factibles entre elementos con posición más baja.[5]

3.- Si se dispone de la posibilidad de definir varias tareas para producir un cierto grado de simultaneidad, cada nivel en el árbol es un candidato a ser procesado por una tarea distinta; o en el lenguaje de los meta-algoritmos, dos procesos con la entrada no vacía pueden actuar simultáneamente.

Cualquiera de las versiones admite que se añadan elementos al final de la estructura de datos, lo que permite una simultaneidad con la producción de elementos.

Pero la versión optimizada se adapta a la exigencia de proporcionar los primeros elementos lo antes posible. A este requisito atienden algo peor las versiones básica y mejorada, mientras que la versión inversa no es aceptable puesto que al considerar los elementos por orden, podría incluso proporcionar el primero al final del todo.

7.8. Análisis de la eficiencia de los algoritmos clásicos

Todos los algoritmos de ordenación clásicos realizan su tarea de forma eficaz: dado un conjunto de elementos como entrada del algoritmo, en un tiempo finito obtienen como resultado el propio conjunto de elementos en el orden deseado.

Que un algoritmo realice su trabajo de forma eficaz es una cuestión necesaria, Que su trabajo sea realizado además de forma eficiente, es una cuestión

[5] Son aquellas que se encuentren en posiciones más arriba en el árbol.

deseable. Cuando se trata de comparar la eficiencia entre algoritmos hay que fijarse detenidamente en muchos factores.

En la mayoría de las bibliografías disponibles, para evaluar la eficiencia de un algoritmo se realiza un análisis que se fija exclusivamente en dos conceptos básicos:

- Por un lado, se cuenta el número de comparaciones que realizan (tiempo).
- Por otro, se calcula el espacio necesario tanto para la ejecución del algoritmo como para el almacenamiento de los datos[6] (espacio).

Pero no son sólo estos factores los que influyen en la eficiencia de un algoritmo. Existen una serie de factores externos ajenos a la naturaleza intrínseca del propio algoritmo y del conjunto de elementos a ordenar, que condicionan la eficiencia definitiva de la ejecución.

La arquitectura en la que este sea ejecutado, encargada de la gestión de los recursos disponibles (gestión de memoria, gestión de periféricos...) condiciona en gran medida la preferencia por uno u otro algoritmo. Si además se introduce el concepto de paralelismo, es necesario tener en cuenta la concurrencia de procesos sobre el mismo conjunto de elementos y el intercambio de información entre diferentes procesadores. Todos estos requisitos impuestos por el entorno nada tienen que ver con la propia naturaleza del proceso de ordenación.

La eficiencia, al igual que otras respuestas a las exigencias externas del entorno, no son el objetivo del análisis sino de la implementación. Este objetivo se ve modelado por la simplicidad en tanto en cuanto una implementación más compleja ve comprometida su seguridad (verificación de corrección) y su flexibilidad (generalidad).

Por otro lado, una implementación más compleja impone excesivas restricciones que dificultan (imposibilitan) adaptaciones (modificaciones) posteriores para responder a cambios en los condicionantes externos.

En la mayoría de los análisis, con excesiva asiduidad no se valora el mantenimiento de las estructuras utilizadas. Dicho mantenimiento se ha de llevar a cabo cuando se producen:

- Intercambios entre elementos.
- Incorporación de una respuesta (nueva información) a la estructura.
- Actualizaciones de la estructura.

[6] Tanto los resultados finales como los parciales.

La estructura de datos en la que se almacena la información debe posibilitar la inclusión de toda la información recogida (no sólo de forma directa sino también la obtenida por transitividad) y con esa información se ha de sugerir que comparaciones útiles realizar a continuación.

Dicho de otra forma, la estructura de datos utilizada ha de ser lo más fácil de mantener. Almacenar toda la información recogida de los procesos de comparación es fundamental, pero su incorporación y su posterior mantenimiento han de tener el menor coste posible. Además es fundamental que en cada paso se reduzca la incertidumbre y se acerque un paso más al estado final. Para ello es deseable la mayor cantidad de información sobre la siguiente posible comparación.

7.9. Análisis de una situación: paralelismo

Como se comentó en el ejemplo del apartado 7.2 (concordancias en un texto) no es frecuente que la ordenación de un conjunto de elementos sea un fin en sí mismo. Por norma general, es más habitual que esa tarea sea una fase de un proceso en el cual un módulo precedente va generando los elementos que, después de ser ordenados, son procesados por un módulo posterior del que se exige que los trate en un orden determinado.

Esta situación en que el módulo de ordenación está intercalado entre un módulo productor de elementos y un módulo consumidor de los mismos, sugiere que parte de las tareas de ordenación se pueden simultanear con la ejecución de esos módulos.

Cuando sólo se dispone de un procesador, la secuencia de ejecución entre los módulos es la más simple de todas las posibles y tan eficiente o más que cualquier otra disposición que se nos pueda ocurrir.

Pero si se pueden ejecutar con algún grado de paralelismo, la construcción y organización de los módulos aprovecha la simultaneidad para alcanzar una eficiencia global superior, aún a costa de una menor eficiencia en algún módulo interno.

En los procesos que nos atañen, las limitaciones lógicas a las que tiene que acomodarse el módulo de ordenación son:

- El módulo (proceso) productor que genera los elementos en un orden cualquiera.
- El módulo consumidor espera que se le proporcionen los elementos en un orden determinado.

Las consecuencias de estas limitaciones son evidentes: el módulo consumidor espera el primer elemento. Sin embargo este no es determinado hasta que el módulo productor haya analizado todos, puesto que ese elemento puede aparecer el último en el proceso de generación. Esto provoca una exclusión entre los módulos productor y consumidor: no se puede iniciar el consumo hasta que haya terminado la producción.

La simultaneidad posible se reduce a:

- La generación de elementos y parte del proceso de ordenación, o a
- Parte de la ordenación de elementos y su consumo.

Por tanto existen situaciones en que los requisitos externos imponen unas condiciones que dan preferencia a unos procedimientos sobre otros al poder atender esos requisitos con más facilidad.

En la anterior situación descrita en que la tarea que "ordena" ha de proporcionar los elementos ordenados a otra, es deseable que esta segunda tarea no tenga que esperar a que se complete el proceso de ordenación. Mientras la segunda tarea procesa los elementos que recibe, la primera realiza la ordenación. Para esta situación, lo deseable es que los elementos sean proporcionados por orden lo antes posible y poder así iniciar su tratamiento posterior.

A continuación se realiza un análisis de esta posible situación, tratando de obtener una simultaneidad en los principales algoritmos que han sido descritos o analizados.

7.9.1. *Quicksort*

El algoritmo de ordenación clásico *Quicksort*[16] ha de comparar todos y cada uno de los elementos con el pivote inicial, por lo que en este caso no se puede simultanear el proceso de producción con la ordenación.

En este algoritmo el proceso raíz encargado de dividir el conjunto de entrada en dos subconjuntos puede comenzar su tarea aún cuando no disponga en la entrada de todos los elementos, pero no pueden iniciar la tarea de ordenación sus procesos hijo hasta que no haya sido repartida por completo la entrada del proceso raíz. La consecuencia de esta situación es que se retrasa el proceso de ordenación, como mínimo, a la completa recepción de elementos procedentes del módulo productor.

Una vez concluida la recepción de elementos e iniciada la ordenación, se puede dar prioridad al tratamiento del subconjunto con los elementos menores en cada uno de los pasos del proceso, con el fin de que los primeros estén disponibles lo antes posible para el consumidor. Por tanto, para el consumidor

es apropiado en parte a costa de adaptar el procedimiento renunciando a ciertas optimizaciones.

Esto implica que la pila de recursividad no es susceptible de ser optimizada, con lo que en el peor de los casos se pasa de una pila[7] de tamaño $\log_2 n$ a ser necesario disponer de una pila de tamaño $^n/_3$.

Si se analiza el nivel de entendimiento del algoritmo con los módulos productor y consumidor, este algoritmo tiene un entendimiento mínimo con el módulo productor y limitado con el módulo consumidor.

7.9.2. *Binario*

El algoritmo *Binario*, por las características intrínsecas en su proceso de ejecución, puede simultanear el trabajo de ordenación con el módulo productor[17]. Esto es debido a que procesa los elementos según los va recibiendo, por lo tanto no es necesario que disponga al inicio de la ejecución del conjunto completo de elementos a tratar.

Iniciado el proceso de ordenación, si la velocidad del productor es lenta este algoritmo es el más adecuado, puesto que en el tiempo que transcurre entre la llegada de elementos el proceso coloca en su lugar a cada elemento que va tratando. En el momento que se produce la llegada del último elemento y es colocado donde le corresponde, este algoritmo devuelve la salida completamente ordenada de todo el conjunto de elementos.

Por las características descritas es posible afirmar que el entendimiento de este algoritmo con el módulo productor es máximo, mientras que con el módulo consumidor es nulo ya que cuando recibe el último elemento y lo procesa, no entrega sólo el primer elemento o conjunto de elementos menores sino que entrega la salida completa y ordenada, por lo que no existe ningún tipo de posible paralelismo entre ambos. Para cuando entrega el primer elemento ya no le queda nada por hacer.

7.9.3. Meta-algoritmo descendente

El meta-algoritmo descendente disfruta de las ventajas de *Quicksort* y *Binario*. Obtener la ventaja de *Binario* parece claro que no requiere de explicación ya que en su ejecución tiene un comportamiento similar a este, por tanto posee la ventaja de iniciar su proceso aún cuando no disponga de todos los elementos.

La ventaja de *Quicksort* también se puede obtener. Para ello lo único que se ha de hacer es dar prioridad a los procesos que se encuentran más a la izquierda del árbol, que son los que van produciendo los primeros elementos.

[7] Pila de tareas pendientes en la versión iterativa y pila de llamadas recursivas en la versión recursiva.

Es decir, las prioridades se ordenan por: en primer lugar la raíz, después su subproceso izquierdo, posteriormente el subproceso izquierdo de este[8]... y así hasta concluir en las hojas del árbol.

Por ello es posible afirmar que el entendimiento con el módulo productor es máximo, aprovechando las ventajas que proporciona una ejecución similar a la de *Binario*, y limitado con el módulo consumidor aprovechando en este caso las mismas ventajas que *Quicksort.*

7.9.4. *Torneo* y *Fusión*

Estos dos algoritmos contenidos en el meta-algoritmo ascendente no son simultaneables en su forma pura, pero como que van construyendo un árbol de forma ascendente se pueden modificar para ser adaptados a las nuevas necesidades.

En la primera fase en que ambos distribuyen el conjunto de elementos a tratar requieren de la completa disponibilidad de los elementos en la entrada del proceso raíz: hasta que no se hayan distribuido todos y cada uno de los elementos a tratar no comienza la fase de comparaciones entre elementos.

Este es el motivo por el que el entendimiento de ambos algoritmos con el módulo productor es limitado. Mientras el módulo productor genera elementos, ambos algoritmos pueden distribuir estos en subconjuntos por lo que pueden trabajar en paralelo, pero únicamente distribuyendo elementos. Como una adaptación para ambos que permita proporcionar al menos un entendimiento limitado con el productor, es posible crear con los elementos que se van recibiendo un árbol colateral que, en un momento determinado, es incorporado al resto.

En cuanto a la fase de comparación y fusión de subconjuntos de elementos, se debe diferenciar el entendimiento con el módulo consumidor para cada uno de ellos:

- *Torneo*: Posee un entendimiento máximo con el módulo consumidor, ya que en el instante en que todos los elementos han sido distribuidos en el árbol de procesos comienzan las comparaciones para obtener el elemento menor y siguientes lo antes posible.
- *Fusión*: Tiene un entendimiento limitado (o nulo en la mayoría de los casos) con el módulo consumidor. Hasta que el proceso raíz no inicie la última fusión el proceso consumidor se encuentra a la espera de recibir elementos.

[8] Prioridad por preorden.

7.9.5. Meta-algoritmo ascendente

En el caso del meta-algoritmo ascendente, al poder establecer prioridad de unos procesos sobre otros en el árbol es posible obtener mejor entendimiento con productor y consumidor que los algoritmos *Torneo* y *Fusión*.

Para obtener mejor entendimiento con el productor basta con ir añadiendo los elementos nuevos que lleguen como elementos finales de la estructura e irlos procesando. De esta forma, el anterior entendimiento nulo de los algoritmos precedentes incluidos en este meta-algoritmo es mejorado.

Mejorar el entendimiento con el módulo consumidor también es posible. Al terminar la generación de elementos se establece mayor prioridad a aquellos procesos que se encuentran situados en la zona superior del árbol: estos son los que proporcionan los primeros elementos. De esta forma el meta-algoritmo entrega elementos para que el consumidor comience su tarea y simultáneamente continuar el proceso de ordenación en su árbol de procesos. No obstante, sólo la última fusión es simultaneable con el consumidor.

7.10. Observaciones

Es posible que en determinadas ocasiones y con el fin de no mantener detenida la tarea consumidora, se permita que esta consuma elementos aunque no sean los primeros, pero prefiriéndolos. Un posible ejemplo se encuentra al tratar de gestionar una cola de tareas con prioridades.

En otras situaciones en que se necesita obligatoriamente que los elementos sean tratados en orden, a modo de resumen, los algoritmos clásicos se comportan del modo que se describe a continuación.

7.10.1. El primero lo antes posible

Si lo que se desea es disponer del primer elemento para ser consumido lo antes posible, el análisis de los algoritmos clásicos visto bajo este prisma es el siguiente::

- **Binario:** No es apropiado, porque sitúa los elementos uno a uno y hasta que no ha leído y procesado todos los elementos no proporciona el primero.
- **Fusión:** No es apropiado, hasta que no inicia la última fusión no proporciona el primer elemento.
- **Quicksort:** Apropiado en parte a costa de adaptar el procedimiento renunciando a ciertas optimizaciones.

- **Torneo:** Es el mejor, porque proporciona los primeros elementos lo antes posible.

En media, a modo de resumen, el primer elemento lo proporcionan después de realizar un cierto número de comparaciones, concretamente:

- *Binario*: $n \log_2 n$ comparaciones.
- *Fusión*: $n \log_2 \frac{n}{2}$ comparaciones.
- *Quicksort*: $2 \times n$ comparaciones.
- *Torneo*: $n-1$ comparaciones.

7.10.2. Los elementos llegan de tarde en tarde

Existen otras situaciones en las que el módulo productor es lento, por lo que no se dispone de todos los elementos desde un principio. Es entonces cuando se desea que el proceso de ordenación se inicie lo antes posible para ir quemando etapas. Por descontado que el proceso sólo se termina cuando todos los elementos se han recibido.

En esta situación el comportamiento de los algoritmos clásicos es el siguiente:

- **Binario:** Es el mejor de todos, puesto que mientras espera que lleguen elementos le da tiempo a realizar su tarea de ordenación para, en el momento que llegue el último elemento, entregar de golpe el conjunto completo ordenado al consumidor.
- **Quicksort:** Esta situación es catastrófica para este algoritmo, ya que el proceso raíz no entrega el control a sus procesos hijo hasta haber analizado por completo los elementos de su entrada.
- **Fusión y Torneo:** A ambos la situación les es perjudicial. Aunque en esos tiempos muertos en que no llegan elementos pueden distribuir el resto para luego fusionarlos, no comienza el proceso de ordenación hasta que se encuentre en la entrada del proceso raíz el último elemento a tratar.

Capítulo 8

Conclusiones

8.1. Conclusiones

Se ha definido una nueva taxonomía de algoritmos de ordenación basada exclusivamente en las características y detalles del concepto ordenación, alejada de cualquier detalle impuesto por las implementaciones. Las bases fundamentales para su definición se encuentran en el concepto de abstracción, a base de eliminar en los algoritmos clásicos de ordenación la implementación de la estructura de control y la estructura de datos.

En primer término, al eliminar la implementación de la estructura de control aparecen los meta-algoritmos con dos formas de trabajo diferentes pero duales.

Por un lado se ha definido un método descendente: los elementos de un conjunto $\mathcal{E}$ cualquiera son ordenados mediante un reparto de estos en subconjuntos disjuntos, simultaneando las tareas de distribución y ordenación. En último término se realiza un recorrido rutinario sobre la estructura de árbol de procesos generada, recuperando los elementos según la relación de orden total preestablecida.

Por otro lado ha sido definido un método de trabajo antagónico (pero curiosamente dual) ascendente: en una primera fase (descendente) los elementos del conjunto $\mathcal{E}$ únicamente son distribuidos en un árbol de procesos, donde cada nodo hoja contiene un único elemento del conjunto a tratar. En una segunda fase (ascendente) los elementos son fusionados hasta obtener en el proceso raíz el conjunto de elementos $\mathcal{E}$ en el orden deseado.

Si se analiza la forma de realizar la ordenación de cualquiera de los algoritmos clásicos conocidos (*Burbuja*, *Inserción*, *Shell*, *Quicksort*, *Heap*, *Fusión*, *Binario*, *Torneo*...) se puede afirmar con rotundidad que el proceso seguido por cualquiera de ellos para realizar su tarea está contenido dentro de uno de

los dos meta-algoritmos definidos.

Yendo un paso más allá en el proceso de abstracción, a base de eliminar la implementación de la estructura de datos a los meta-algoritmos, aparecen los esquemas: estos engloban en un único concepto los meta-algoritmos definidos mostrando el proceso de ordenación en su estado más puro.

Su forma genérica de trabajo se muestra como un grafo dirigido acíclico, y las diferentes particularizaciones (restricciones) que nos acercan a las implementaciones finales conocidas trabajan sobre esquemas de árboles (particularización de grafo), esquemas de cadenas (particularización de árbol) y esquemas de cadena única (eslabones, particularización de cadena).

Por último, en los apéndices se encuentran las demostraciones sobre el mejor lugar por el que realizar la inserción de un eslabón en una cadena, la ventaja de utilizar inserción binaria frente a inserción secuencial, la optimalidad de la equipartición de *Merge* o una propuesta de mejora sobre su proceso de fusión, al demostrarse que es preferible la fusión de dos cadenas por el punto medio de ambas que como lo realiza *Merge* por sus cabezas.

8.2. Desarrollos futuros

Todo el trabajo se apoya en la consideración de acciones efectivas: no se realizan actividades que no sean eficaces hacia el objetivo. Así se evitan comparaciones cuyo resultado es conocido o deducible de las acciones anteriores.

Al considerar algún tipo de paralelismo, los recursos pueden sacrificarse en aras de alcanzar un resultado que reduzca el tiempo de ejecución.

Las ineficiencias pueden ser:

a.- Preguntas redundantes cuya respuesta es necesaria en varios procesos y se renuncia a detener un proceso hasta obtener la respuesta obtenida en otro proceso.

b.- Procesos detenidos esperando una sincronización con otros procesos, lo que se puede reflejar en

c.- Las sobrecargas de distribución de tareas, recogida de resultados y su composición y eventual comunicación entre procesos.

Una posible prolongación de este trabajo consiste en determinar aquellas acciones que a pesar de conocerse su ineficiencia, contribuyen a reducir o eliminar las esperas en aquellos procesos que han de coordinarse. Se trata de

enfrentar los costes de espera con procesador inactivo con los costes de retraso global. (Un ejemplo de la aplicación es la utilización del paralelismo para realizar la inserción de elementos mediante el uso de un procesador vectorial, que es el tipo de paralelismo más simple).

Otra posible continuación es la adaptación a situaciones en que el número de procesadores es reducido. Se trata de distribuir las tareas de la forma más eficaz posible para conseguir reducir los tiempos muertos, teniendo en cuenta la sobrecarga de la distribución de tareas propiamente dicha. (Un ejemplo donde aplicar esta adaptación se encuentra en la asignación de procesadores al meta-algoritmo ascendente de forma que se obtenga una reducción de los tiempos muertos.)

Todo ello adaptándose al tipo de paralelismo disponible.

La eficiencia y el tiempo son dos parámetros independientes, por lo que en función de la valoración del grado de satisfacción de otros condicionantes externos se preferirá una alternativa u otra. Se trata de un caso de valoración con múltiples factores.

En cuanto a la posible existencia de un óptimo general hay una imposibilidad lógica de reducir el tiempo con el uso de más procesadores: si se exige que la eficiencia sea máxima a lo largo del proceso, algunos de los procesadores necesariamente estarán ociosos en algún momento.

Es decir, se puede reducir el tiempo con el uso de varios procesadores y la renuncia a la máxima eficiencia: parte de los procesadores estarán parados durante más o menos tiempo.

Apéndices

Apéndice A

Inserción de un elemento en una cadena

Al desgranar los principios fundamentales de cualquier algoritmo de ordenación a lo largo de su proceso de ejecución, este realiza la inserción de un elemento $x \in \mathcal{E}$ dentro de una colección de elementos que previamente ya se encuentran ordenados en lo que se supone como una cadena de elementos C.

La inserción del elemento x en la cadena, en principio y al no disponer de más información, se puede realizar por cualquier punto de la misma. La pregunta a responder es, ¿qué elemento de la cadena se elige para realizar la comparación hasta encontrar la posición de x en esta?.

Supongamos que se dispone de una cadena C formada por p elementos (longitud de $C = p$) cumpliendo una relación de orden. Se pretende insertar un elemento x en dicha cadena sin tener preferencia por ningún elemento, por lo que se selecciona un elemento $y \in C$ que se encuentra en la posición s. La figura A.1 muestra de forma gráfica la situación descrita.

Para conocer si el elemento x ha de ser insertado en el lugar que ocupa y, se deben comparar ambos elementos x e y. Cuando finalice la inserción del elemento x (donde corresponda) la longitud de la cadena C se verá incrementada en *un elemento*, por lo que su longitud final será igual a $p+1$.[1]

Por tanto, para realizar la inserción de x se dispone de $p+1$ posibilidades. Al comparar x con y la respuesta divide las $p+1$ posibilidades en s elementos por un lado (se recuerda que s es la posición que ocupa y en la cadena C) y $(p+1)-s$ elementos por otro. Por teoría de la información, la división es óptima si ambos valores son iguales. Se igualan ambos términos y se resuelve la ecuación.

[1] Y debe seguir conservando la relación de orden.

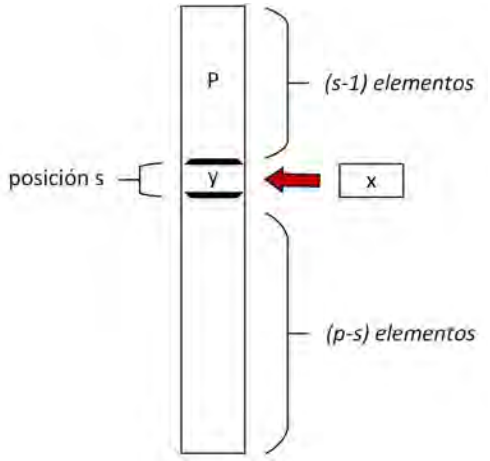


FIGURA A.1: ¿Con qué elemento y se compara el elemento x?

$$s = p + 1 - s$$

$$2s = p + 1$$

$$\boxed{s = \frac{p+1}{2}}$$

El resultado obtenido parece que coincide con lo que sugiere la intuición: el mejor elemento $y \in \mathcal{E}$ de la cadena C con el que realizar la comparación es el que se encuentra situado en la posición s, siendo el valor de s la posición central de la cadena C.

Apéndice B

Inserción binaria vs inserción secuencial: número medio de comparaciones

Cuando se desea insertar un elemento dentro de una cadena, si no existe una imposición externa que obligue a la utilización de un método determinado se puede optar por insertar el elemento de forma secuencial o binaria.

Supongamos que se dispone de una cadena de elementos C cumpliendo una relación de orden formada por p elementos, por tanto de longitud p. Se pretende insertar en dicha cadena un nuevo elemento $x \in \mathcal{E}$. Cuando se produce la inserción del elemento en la cadena C en el lugar que corresponda la longitud de la cadena se incrementa hasta el valor $p+1$.[1]

A continuación se realiza un análisis comparativo de ambas posibilidades, de manera que se establezca en términos estadísticos cual de las dos alternativas es preferible poner en práctica en caso de disfrutar de libertad de elección.

B.1. Inserción binaria

Como su propio nombre indica se trata de insertar el elemento x en la cadena C de forma binaria, comparando dicho elemento en cada iteración con aquel que ocupa la posición media del trozo de cadena que reste por comparar.

Cuando se realiza la inserción del elemento x en la cadena C compuesta por p elementos el número de comparaciones que se ha de realizar es fijo, siendo su valor igual a $\log_2(p+1)$. Cuando la longitud de la cadena C es grande el

[1] Y continua manteniendo la relación de orden inicial establecida.

valor que se obtiene es similar al que presenta la función de logaritmo en base 2.

B.2. Inserción secuencial

Consiste en comparar el elemento x comenzando desde el inicio (o fin) de la cadena C de forma secuencial con todos y cada uno de los elementos hasta encontrar la posición que ha de ocupar.

Supongamos que se comienzan las comparaciones por la cabeza. Si hay suerte, con una única comparación el elemento pasa a ocupar la posición que le corresponde. De no ser así es comparado con el segundo elemento, por lo que se realizan dos comparaciones. Si se ha encontrado el lugar que le corresponde se finaliza. De no ser así se itera el proceso hasta comparar el elemento con el que ocupa la posición p de la cadena: se ha tenido mala suerte, y para colocar el elemento x en el lugar que le corresponde de la cadena se han necesitado p comparaciones.

En términos estadísticos el número medio de comparaciones se corresponde con los siguientes cálculos:

$$\frac{\sum_{i=1}^{p+1} i - 1}{p+1} = \frac{\frac{(p+1)(p+2)}{2} - 1}{p+1} = \frac{(p+1)(p+2) - 2}{2(p+1)} \approx \frac{p+2}{2}$$

El resultado obtenido para el número medio de comparaciones es una función lineal. Para una longitud de la cadena C grande el número medio de comparaciones presenta un valor similar a la función lineal $\frac{p+2}{2}$.

B.3. Conclusiones

El número medio de comparaciones en la inserción binaria presenta una función logarítmica, mientras que la misma situación realizada mediante inserción secuencial presenta unos valores próximos a una función lineal: por tanto el número medio de comparaciones es mucho menor en la inserción binaria que en la secuencial como se aprecia en el gráfico comparativo B.1 para ambas funciones.

p	$\log_2(p+1)$	$(p+2)/2$
1	1,00	1,5
2	1,58	2
3	2,00	2,5
4	2,32	3
5	2,58	3,5
6	2,81	4
7	3,00	4,5
8	3,17	5
9	3,32	5,5
10	3,46	6
11	3,58	6,5
12	3,70	7
13	3,81	7,5
14	3,91	8
15	4,00	8,5
16	4,09	9
17	4,17	9,5
18	4,25	10
19	4,32	10,5
20	4,39	11
21	4,46	11,5
22	4,52	12
23	4,58	12,5
24	4,64	13
25	4,70	13,5

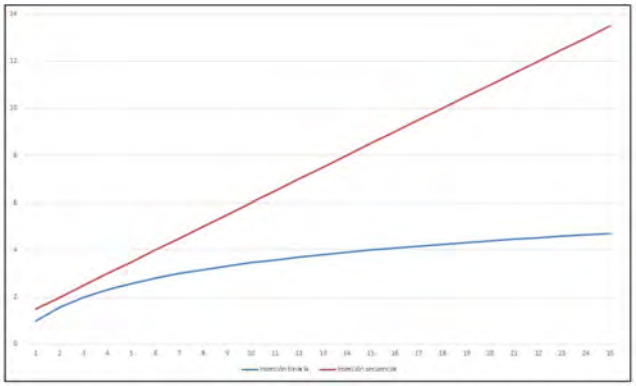


FIGURA B.1: Numero medio de comparaciones para cada tipo de inserción

Los resultados obtenidos nuevamente confirman lo que la intuición parece sugerir al abordar el problema analizado.

Apéndice C

Inserción por la cabeza vs inserción central: información obtenida con cada pregunta

C.1. Introducción

Cuando se necesita realizar la inserción de un elemento en una cadena de elementos la cual posee una relación de orden, en principio si no se dispone de más información que facilite la selección, el elemento elegido de la cadena con el que realizar la comparación puede ser cualquiera.

Por tanto la cuestión que se plantea es: de todos los elementos (posiciones) que forman la cadena en la que insertar el nuevo elemento, ¿existe algún elemento (posición) que proporcione una mayor información que la del resto de elementos que forman dicha cadena?.

Para responder a esta pregunta, en los siguientes apartados se desarrollan con ayuda de la teoría de la información una serie de cálculos basados en el principio de *entropía*, también denominado entropía de la información o entropía de Shannon.

En síntesis el supuesto que se aborda realiza la inserción de un elemento x en una cadena C, que dispone de una relación de orden entre sus elementos y cuya longitud es p. En principio no se dispone de ninguna información adicional que permita inclinar la balanza hacia un elemento concreto sobre el resto. La elección del elemento $y \in C$ es por tanto aleatoria.

Los cálculos realizados están basados en dos situaciones diferentes: en primer lugar la más habitual consistente en iniciar el proceso de inserción en la cadena por la cabeza (o por la cola) e ir comparando de forma secuencial con el resto de elementos hasta encontrar la posición adecuada. En segundo

lugar la que en principio parece mejor alternativa, consistente en seleccionar para cada comparación el elemento situado en la posición central de la cadena (o lo que reste de esta).

C.2. Principio de entropía de Shannon

La entropía se define como la cantidad de información promedio que contienen los símbolos utilizados. Aquellos símbolos cuya probabilidad es menor son los que más información aportan a la incógnita planteada en un problema.

El concepto se entiende mejor mediante la utilización de un ejemplo: si se considera como sistema de símbolos las palabras que conforman un texto, aquellas de mayor frecuencia como artículos, preposiciones o conjunciones, aportan poca información al lector. Otras palabras cuya aparición tiene un índice de aparición en frecuencia claramente inferior, como por ejemplo nombres, verbos o adjetivos, aportan más información al receptor. Si de un texto cualquiera son eliminados todos los artículos determinados que aparecen, con una gran probabilidad sigue siendo comprensible. Sin embargo si se borran varios verbos del texto, su comprensión pasa a ser difícil e incluso carente de sentido.

Cuando la aparición de todos los símbolos posee la misma probabilidad (distribución de probabilidad plana) todos aportan la misma información relevante, resultando en esta situación una entropía máxima.

La *entropía* de un mensaje x denotado por $H(x)$, es el valor medio ponderado de la cantidad de información de los diversos estados del mensaje. La fórmula que permite realizar su cálculo es la siguiente:

$$H(x) = -\sum_i p(x_i) \log_2 p(x_i)$$

C.3. Inserción de un elemento por la cabeza

Cuando se realiza la inserción de un elemento x en una cadena C de longitud p la longitud de dicha cadena pasa a ser p+1. Al realizar la inserción de dicho elemento por la cabeza, la cadena C queda dividida en dos subcadenas, C_1 y C_2, siendo la longitud de C_1 de un elemento y la longitud de C_2 de p elementos.

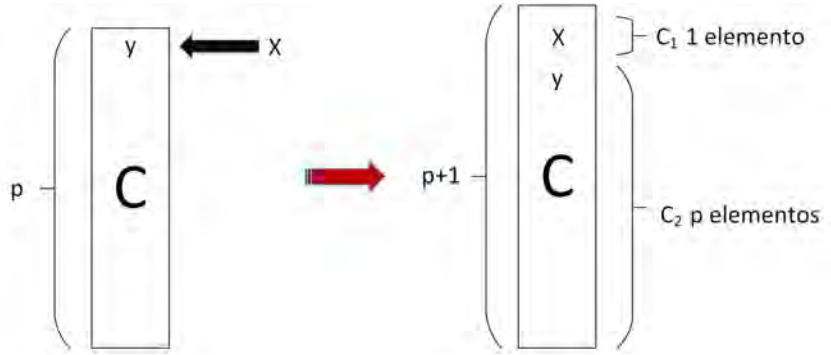


FIGURA C.1: Inserción de un elemento por la cabeza de una cadena

Por tanto la información que se obtiene en cada una de las subcadenas es la siguiente:

$$C_1 \Rightarrow \frac{1}{p+1}$$

$$C_2 \Rightarrow \frac{p}{p+1}$$

Según el principio de entropía de Shannon:

$$H = \frac{1}{p+1}\log_2(p+1) + \frac{p}{p+1}(\log_2(p+1) - \log_2 p) =$$

$$\log_2(p+1) - \frac{p}{p+1}\log_2 p =$$

$$\boxed{\log_2(p+1) - \log_2 p + \frac{1}{p+1}\log_2 p}$$

Analizando el resultado obtenido se puede afirmar que:

$$\log_2(p+1) - \log_2 p \approx 0$$

Por lo que resta por analizar únicamente la segunda parte del resultado:

$$\frac{1}{p+1}\log_2 p$$

Para cadenas grandes (valores de p grandes) el valor de p+1 crece mucho más rápido que la función logarítmica, por lo que el resultado del valor de la función H calculada se aproxima a 0. El resultado obtenido, desde luego, no es demasiado esperanzador.

C.4. Inserción de un elemento por el centro de la cadena

El planteamiento es idéntico al anterior pero en lugar de comparar el elemento a insertar x con el primer (o último) elemento de la cadena C, se compara con un elemento que esté situado en una posición central. Al igual que en el caso anterior, al realizar la inserción del elemento la longitud total de la cadena pasa a ser de p+1 elementos, por lo que el número de posibles posiciones en las que insertar x es igual a dicho valor. La figura C.2 muestra de forma gráfica la situación descrita.

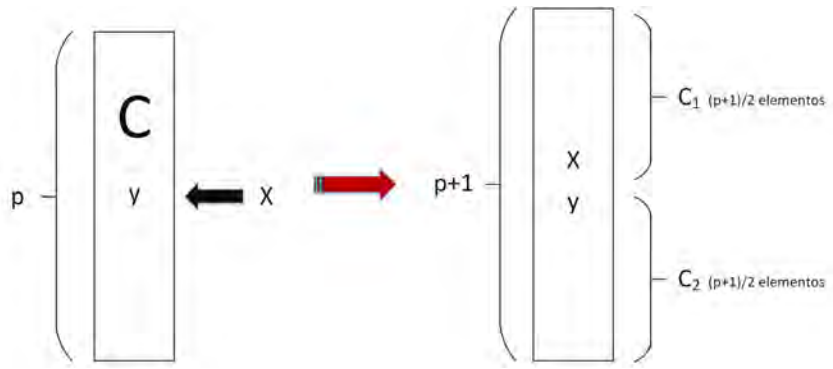


FIGURA C.2: Inserción de un elemento por el punto medio de una cadena

Al realizar la comparación con un elemento $y \in C$ situado en una posición central, la partición de la cadena C se produce en dos subcadenas de longitud aproximada $\frac{p+1}{2}$ cada una. Por tanto la información que se obtiene de C_1 y C_2 es igual a:

$$C_1 \Rightarrow \frac{\frac{p+1}{2}}{p+1} = \frac{1}{2}$$

$$C_2 \Rightarrow \frac{\frac{p+1}{2}}{p+1} = \frac{1}{2}$$

Aplicando el principio de entropía de Shannon:

$$H = \frac{1}{2}\log_2 2 + \frac{1}{2}\log_2 2 = \boxed{1}$$

En esta situación la información obtenida en el proceso de comparación es mucho mejor que la situación anterior, ya que el resultado nos da 1 bit de información.

C.5. Conclusiones

Realizar la inserción por la cabeza es equiparable a realizar la inserción de forma secuencial y realizar la inserción eligiendo un punto medio de la cadena a la inserción binaria. Utilizando dichos términos, se afirma que realizar la inserción binaria es un proceso óptimo mientras que realizar el proceso utilizando inserción secuencial, pésimo.

Por los resultados obtenidos parece claro que siempre que sea posible y no existan circunstancias externas que lo impidan, se recomienda realizar la inserción por el punto medio de la cadena. Desde el punto de vista de la teoría de la información esta es siempre la mejor opción.

Como nota aclaratoria al proceso, destacar que las situaciones analizadas sólo permiten obtener dos posibles alternativas (respuestas), por lo que parten el conjunto original en dos subconjuntos. En general un experimento es más informativo cuando:

- En más partes divida la respuesta el resultado (en este caso el valor es siempre igual a 2).
- Más uniformes sean las partes en que se divide el conjunto inicial. Esta afirmación se cumple perfectamente en este caso: la peor situación se produce cuando el conjunto inicial que dispone de $p+1$ posibilidades de inserción se divide en dos subconjuntos, uno de tamaño p y otro formado por un único elemento. En la mejor situación ambos conjuntos poseen un tamaño similar, disponiendo cada subconjuto de $\frac{p+1}{2}$ elementos.

Apéndice D

Optimalidad de la equipartición en *Fusión*

El proceso completo que realiza el algoritmo de ordenación *Fusión* se divide en dos fases claramente diferenciadas:

- Primera fase: Repartir el conjunto de elementos de entrada en subconjuntos de menor tamaño.
- Segunda fase: Fusionar los subconjuntos de elementos hasta obtener un único conjunto cumpliendo una relación de orden total, donde se encuentren todos los elementos de la entrada en el orden deseado.

Algoritmos como *Quicksort* o *Binario* dejan al azar el reparto de los elementos: la elección del pivote en el primer caso y el orden de aparición de los elementos en el segundo, determinan la perfecta o nefasta eficiencia de estos algoritmos.

En el caso del algoritmo *Fusión* dicho proceso es controlable por diseño. A continuación se muestra cual es la mejor decisión posible a la hora de repartir un conjunto de elementos en dos subconjuntos que posteriormente deben ser fundidos.

Supongamos que se dispone de una cadena C de elementos que en una primera fase se ha de dividir en dos subcadenas P y Q, cuyas longitudes tras la división son p y q respectivamente. En el proceso ambas subcadenas son ordenadas y en una segunda fase del proceso deben ser fundidas, lo que requiere un mínimo de $[p,\, q]$ comparaciones y un máximo de $(p+q-1)$.

Se sabe que el proceso de ordenación de un conjunto compuesto por n elementos, requiere del orden de $n \log_2 n$ comparaciones. Así pues, la función

de la que se debe calcular el mínimo al partir un conjunto de elementos en dos subconjuntos C_1 y C_2 de tamaños:

- C_1: compuesto por x elementos
- C_2: compuesto por $(n - x)$ elementos

respectivamente es la siguiente:

$$F(x) = n \log_2 n + (n - x) \log_2 (n - x) + K$$

El mínimo se produce cuando $F'(x) = 0$. Realizando los cálculos necesarios para obtener el resultado:

$$F'(x) = \log_2 n + 1 - \log_2 (n - x) - 1$$

$$\log_2 n + 1 - \log_2 (n - x) - 1 = 0$$

$$\log_2 n = \log_2 (n - x)$$

$$x = n - x$$

$$\boxed{x = \frac{n}{2}}$$

Como demuestran los cálculos, la mejor decisión posible en el reparto de elementos se obtiene, como bien aprovecha *Fusión*, mediante la equipartición del conjunto de entrada en dos subconjuntos de idéntico tamaño (o lo más iguales posible).

Apéndice E

Inserción binaria de la cabeza vs inserción binaria del elemento central

La cuestión a abordar es la siguiente. Supongamos que se pretende realizar la fusión de dos cadenas P y Q ambas ordenadas y cuyas longitudes iniciales son p y q respectivamente. Para completar la tarea de fusión se ha de iterar la acción de insertar cada elemento de la cadena Q en P (la inserción de P en Q presenta idéntico razonamiento) hasta que no queden elementos en Q por tratar. La tarea completa se describe de la siguiente forma:

1.- Seleccionar un elemento $x \in Q$ eliminándolo de la cadena.

2.- Insertar el elemento x en el lugar que le corresponda de la cadena P.

3.- Repetir las acciones de los pasos 1 y 2 (selección e inserción) con los $(q - i)$ elementos restantes de la cadena Q hasta completar la fusión de ambas (instante en que longitud de la cadena Q pasa a ser 0).

Como se demuestra en el apéndice C, la mejor opción para realizar la inserción de un elemento en una cadena es comenzar por su punto medio. Por tanto la inserción del elemento x en P se ha de realizar de forma binaria.

Yendo un paso más allá, la pregunta que se plantea es: al seleccionar un elemento x de la cadena Q, ¿tiene importancia la posición que ocupa dicho elemento x en la cadena inicial Q?. La respuesta se obtiene con los cálculos mostrados en los siguientes apartados.

Para satisfacer el planteamiento de la cuestión se distinguen dos posibles situaciones A) y B) a lo largo de la demostración, que se diferencian en la selección del elemento x a insertar:

A) Se selecciona de la cadena Q el elemento situado en su primera posición (cabeza).

B) Se selecciona de la cadena Q un elemento que se encuentre en su posición central (por tanto en una posición aproximada a $\frac{q}{2}$)

E.1. Reducción de la incertidumbre

Un primer análisis pretende conocer la reducción de incertidumbre que se obtiene en cada una de las situaciones anteriores. La incertidumbre inicial para ambas es idéntica: $p \times q$.

Realizar la acción de insertar necesita en primer lugar seleccionar el elemento x que corresponda (según la anteriores situaciones descritas A o B) de la cadena Q, para después realizar de forma binaria la inserción de x en la cadena P.

Según la situación en que se encuentra, tras la inserción del elemento x se producen las siguientes particiones:

A) De la cadena Q desaparece el primer elemento quedando los $(q-1)$ elementos siguientes al seleccionado. La inserción en P se realiza en una posición en la que hay r elementos anteriores a x y s elementos posteriores.

B) De la cadena Q desaparece un elemento situado en una posición media de la cadena, por lo que quedan aproximadamente $\frac{(q-1)}{2}$ elementos anteriores y posteriores al x seleccionado en la cadena Q. De forma idéntica a la situación A, la inserción en P se realiza en una posición en la que hay r elementos anteriores a x y s elementos posteriores.

La figura E.1 muestra de forma gráfica las dos situaciones que se plantean.

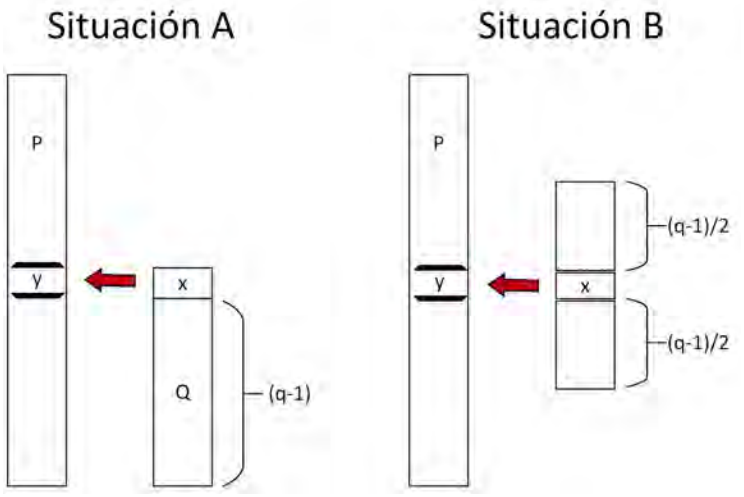


FIGURA E.1: Situaciones A y B: elemento cabecera vs elemento medio

Una vez insertado el elemento x en P, cada una de las situaciones requiere la fusión de los siguientes elementos:

A) Fusionar s elementos de P con $(q-1)$ elementos de Q.

B) Fusionar r elementos de P con $\dfrac{(q-1)}{2}$ elementos de Q, más la fusión de s elementos de P con los restantes $\dfrac{(q-1)}{2}$ de Q.

En la figura E.2 se muestra de forma gráfica las fusiones que han de realizarse.

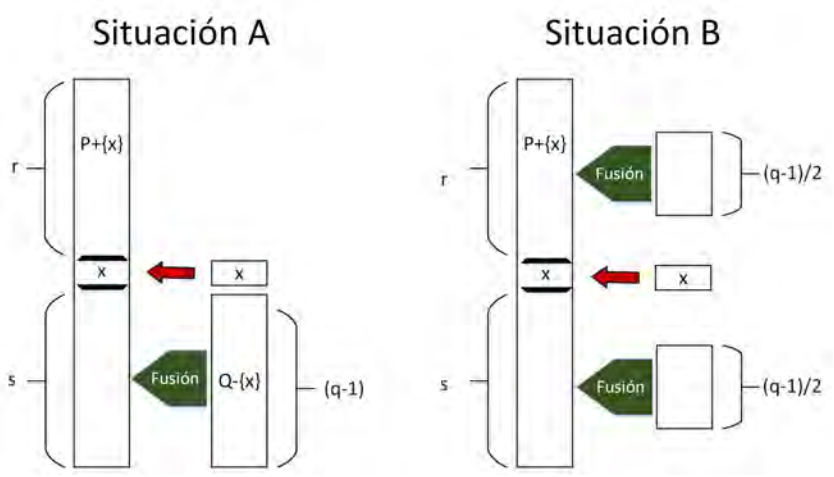


FIGURA E.2: Q: fusión elemento cabeza vs fusión elemento medio

Haciendo unos pequeños cálculos, al finalizar la inserción la reducción de la incertidumbre en cada situación se reduce a:

Situación A.-

$$s \cdot (q-1)$$

Situación B.-

$$r \cdot \frac{(q-1)}{2} + s \cdot \frac{(q-1)}{2} = p \cdot \frac{(q-1)}{2}$$

En término medio se puede decir que $r \approx s \approx \frac{p}{2}$ por lo que ambas situaciones A y B son equivalentes en:

1.- Reducción de la incertidumbre.

2.- Esfuerzo necesario para realizar la selección de x y su inserción en P.

Según el resultado obtenido parece que nada tiene que ver la posición en la que se encuentre x en la cadena Q, al menos desde el punto de vista de la teoría de la información.

E.2. Dispersión de elementos

A la vista de los resultados obtenidos, dado que en esta ocasión las operaciones matemáticas y lo que la lógica parecía indicar al plantear el problema

no coinciden, se decide abordar la cuestión desde otro punto de vista: la dispersión de elementos que se produce en cada caso.

Manteniendo las anteriores situaciones definidas A y B, se realizan una serie de cálculos matemáticos para obtener la gráfica producida en lo que a dispersión de elementos se refiere para cada una de las situaciones.

Los cálculos se basan en determinar, una vez realizada la inserción del elemento x en la cadena P, a qué reconduce en cada situación la fusión de los elementos restantes para finalizar la tarea completa. Como se aprecia en la figura E.2, una vez insertado en elemento x, la situación A reconduce a fusionar s elementos de P con q–1 elementos de Q y la situación B reconduce a fusionar r elementos de P con $\frac{q-1}{2}$ elementos de Q por un lado, y s elementos de P con los restantes $\frac{q-1}{2}$ elementos de Q.

El número de posibilidades de inserción, en base al número de elementos existentes desde la posición en la que x ha sido insertado hasta el final de la cadena P, se calcula con los siguientes números combinatorios según las situaciones descritas:

Situación A.-

$$\binom{Elementos\ hasta\ fin\ de\ P + q - 1}{q - 1}$$

Situación B.-

$$\binom{P - elementos\ hasta\ fin\ de\ P + \frac{q-1}{2}}{P - elementos\ hasta\ fin\ de\ P} \cdot \binom{Elementos\ hasta\ fin\ de\ P + \frac{q-1}{2}}{Elementos\ hasta\ fin\ de\ P}$$

Elementos hasta fin de P	Situación A	Situación B
0	1	0
1	5	5
2	15	30
3	35	105
4	70	280
5	126	630
6	210	1260
7	330	2310
8	495	3960
9	715	6435
10	1001	10010

FIGURA E.3: Cálculos obtenidos con valores de $P = 10$ y $Q = 5$

Con la descripción anterior se realizan los cálculos pertinentes para insertar una cadena $Q = 5$ en una cadena $P = 10$ con los valores que se muestran en la figura E.3. La primer columna representa el número de elementos que restan una vez insertado x hasta el final de la cadena P. Con dichos cálculos se genera la gráfica E.4 que representa la dispersión para cada una de las situaciones.

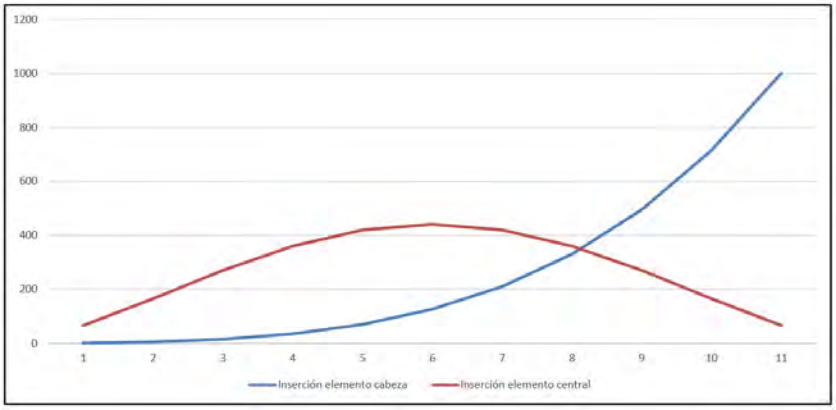


FIGURA E.4: Función dispersión de elementos para situaciones A y B

La gráfica de la figura E.4 es esclarecedora y confirma lo que la lógica parecía establecer al inicio del planteamiento del problema: una situación es mejor que la otra. Es verdad que el número de partes en que ambas dividen los elementos es idéntico, pero los tamaños de cada división (según muestra la figura E.3) son más homogéneos en la situación B que en la situación A.

E.3. Conclusiones

Se parte de una cuestión que al menos desde el punto de vista de la lógica, parecía conducir de forma inmediata a una clara respuesta sobre el elemento a seleccionar en una cadena que se pretende fusionar con otra: la elección del elemento medio parece ser siempre preferida frente al elemento situado en cualquier extremo.

Sin embargo el resultado no ha sido tan sencillo de obtener, más cuando la teoría de la información arroja el resultado de que para valores de p grandes (cadena formada por un elevado número de elementos) ambas situaciones son idénticas.

Aún así se busca otro punto de vista al problema: se analiza el resultado según el número de partes en que divide cada situación, así como el número de elementos que compone cada una. Bajo este nuevo prisma las situaciones son diametralmente opuestas. El número de divisiones es idéntico para ambas situaciones, sin embargo el tamaño de cada parte es mucho más homogéneo al seleccionar un elemento que se encuentra en una posición central de la cadena Q (siempre en términos estadísticos).

Al insertar x en la cadena P, las particiones r y s que se producen no son controlables, pero la dispersión generada en la situación A es muy superior a la obtenida bajo las premisas de la situación B.

Por tanto se puede asegurar que para fundir dos cadenas de elementos los mejores resultados se obtienen cuando el elemento seleccionado es insertado de forma binaria en la cadena final y, además, cuando la selección del mismo se produce por un punto medio de la cadena que contiene los elementos a insertar (la situación B descrita).

El único inconveniente que presentan estos resultados es que la programación de la situación B es ligeramente más compleja que la situación A. Es posible que bajo determinadas circunstancias externas aplicadas sobre un problema concreto no se vean compensadas la inversión y el esfuerzo que supone realizar dicha programación: antes de tomar cualquier decisión, la mejor opción es realizar un análisis exhaustivo de la casuística de cada problema en que pretenda aplicarse.

Apéndice F

Ordenación por *Selección*

El algoritmo de ordenación por *Selección*[18], por su sencillez de comprensión y codificación, se utiliza de forma habitual como primer ejemplo de algoritmo para ordenar un conjunto de elementos dado, generalmente almacenados en un array o vector.

Permite que el lector comprenda de una forma clara y concisa en qué consiste el proceso de ordenar elementos utilizando un algoritmo sencillo, así como cuantificar las medidas utilizadas para saber si un algoritmo de ordenación es mejor que otro: el número de comparaciones que ha de realizar entre los diferentes elementos del conjunto y el número de intercambios que deben realizarse para obtener la relación de orden total buscada.

En el algoritmo de ordenación por *Selección,* el método utilizado consiste en buscar en cada pasada del vector de elementos el menor de todos ellos y colocarlo en la posición que le corresponde. De tal manera que, en la primera pasada del vector se busca el menor elemento del conjunto. Una vez encontrado se intercambia con el elemento que ocupa la primera posición del vector. En la segunda pasada se busca el menor elemento entre todos los que quedan (exceptuando el primero) y, una vez encontrado, se coloca en la segunda posición. El proceso se repite hasta que se tengan colocados todos los elementos en el lugar que les corresponden, cumpliendo que el conjunto de elementos de entrada dado se encuentra en su salida de forma ordenada.

Los pasos sucesivos a dar por el algoritmo son:

1.- Seleccionar el elemento menor del vector de n elementos.

2.- Intercambiar dicho elemento con el primero.

3.- Repetir las 2 operaciones anteriores con los $n-1$ elementos restantes, colocando el segundo menor en la segunda posición, el tercero menor en la tercera posición, y así, sucesivamente.

El algoritmo de *Selección*, expresado en pseudocódigo, es el siguiente:

```
Entrada: Conjunto de n elementos.
Salida: Conjunto de n elementos cumpliendo una relación de orden total.
1: Para i := 1 hasta n−1 hacer
2:         minimo := i;
3:         Para j := i + 1 hasta n hacer
4:                 Si lista [j] < lista[minimo] entonces
5:                         minimo := j;
6:                 fin Si;
7:         fin Para;
8:         intercambiar (lista[i], lista[minimo]);
9: fin Para;
```

ALGORITMO 5: Pseudocódigo algoritmo de ordenación por *Selección*

La acción de *intercambiar* entre sí 2 elementos es una acción compuesta que contiene las siguientes acciones, utilizando para ello una variable auxiliar *Aux*:

```
Entrada: 2 elementos a intercambiar.
Salida: Valores intercambiados de los elementos pasados.
1: Aux := A;
2: A := B;
3: B := Aux;
```

ALGORITMO 6: Procedimiento para intercambiar elementos

El proceso de búsqueda del menor elemento para cada ocasión, produce que el elemento a colocar sea comparado con el resto de elementos. Así, si se dispone de un conjunto de entrada formado por n elementos, en la primera pasada el primer elemento es comparado con los $n-1$ elementos restantes. Colocado el menor elemento se busca el siguiente, que es comparado con los $n-2$ elementos restantes y así, sucesivamente.

El tiempo de ejecución para este algoritmo es de $O(n^2)$. A pesar de obtener muy malos tiempos de ejecución, su simplicidad de codificación hacen que sea apropiada su utilización cuando se dispone de una cantidad de elementos a ordenar muy reducida y no se quiere perder excesivo tiempo en codificar el algoritmo.

Apéndice G

Ordenación por intercambio o *Burbuja*

El algoritmo de clasificación de intercambio o *Burbuja*[18], es otro algoritmo muy sencillo en implementación y comprensión, que se basa en el principio de comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que se disponga de todos los elementos ordenados.

Su funcionamiento es muy simple: cada elemento se compara con el elemento adyacente. Si ambos elementos se encuentran en orden, no se hace nada. En caso contrario se intercambian sus posiciones. El proceso finaliza cuando todos los elementos se encuentran en orden y al comparar cualquier elemento con su adyacente no se produce ningún intercambio. Este algoritmo debe su nombre a la forma en la que suben por la lista los elementos al ser intercambiados. Simulan ser "pequeñas burbujas" que ascienden hasta la posición final que han de ocupar en el resultado.

A continuación se muestra el proceso de ordenación con un pequeño ejemplo. Supongamos que se desea clasificar en orden ascendente el vector (o lista de elementos) que se encuentra en la figura G.1.

Elem[1]	Elem[2]	Elem[3]	Elem[4]	Elem[5]	Elem[6]	Elem[7]	Elem[8]
50	15	56	14	35	1	12	9

FIGURA G.1: Lista de elementos desordenada

Los pasos a realizar por el algoritmo son los siguientes:

1.- Comparar Elemento[1] y Elemento[2]: si están en orden se mantienen como están. En caso contrario se intercambian entre sí.

2.- A continuación se comparan los elementos 2 y 3. De nuevo se intercambian en caso de ser necesario.

3.- El proceso continúa hasta que cada elemento del vector ha sido comparado con sus elementos adyacentes y se han realizado todos los intercambios necesarios.

El método expresado en pseudocódigo, para un primer diseño, es el siguiente:

Entrada: Conjunto de n elementos.
Salida: Conjunto de n elementos cumpliendo una relación de orden total.

```
1: Para j := 1 hasta n − 1 hacer
2:         Para i := 1 hasta n − 1 hacer
3:                 Si (Elem[i] > Elem[i + 1]) entonces
4:                         Intercambiar (Elem[i], Elem[i + 1])
5:                 fin Si;
6:         fin Para;
7: fin Para;
```

ALGORITMO 7: Pseudocódigo algoritmo de intercambio o *Burbuja*

La acción de intercambiar entre sí los valores de 2 elementos es idéntica a la ya utilizada en el algoritmo 6 del Apéndice F.

De esta forma, el elemento ganador sube posición a posición hacia el final de la lista de forma análoga a las burbujas de aire en un depósito de agua. Tras realizar un recorrido completo por todo el vector, el elemento ganador ha subido en la lista y ocupa la última posición. En la segunda pasada al vector, el segundo elemento ganador ocupa la penúltima posición del mismo y así, sucesivamente.

Si sobre un vector que contiene n *elementos* se efectúa la operación $n-1$ veces, se obtiene como resultado los elementos del vector inicial ordenados. Cada elemento para ocupar su posición, es comparado con los $n-1$ elementos restantes, lo que implica realizar $n-1$ comparaciones para cada uno. Por lo tanto, como máximo un elemento (concretamente el primero) es intercambiado en $n-1$ ocasiones. La ordenación total exige como máximo:

$$(n-1)(n-1) = (n-1)^2$$

Se puede realizar una mejora inmediata en la velocidad de ejecución del algoritmo. Es fácil observar que en el primer recorrido del vector (para $i = 1$) el elemento ganador es movido a la última posición del mismo $Elem[n]$. Con

esta nueva situación de los elementos, en la siguiente pasada al vector no es necesario realizar la comparación de los elementos del vector que ocupan las posiciones $n-1$ y n puesto que ya se sabe que el elemento ganador ocupa la posición n, por lo que la respuesta de esa comparación es conocida de antemano y por tanto redundante. En otras palabras, el límite superior del bucle "para" puede ser $n-2$. Tras finalizar cada pasada del algoritmo se decrementa en una unidad el límite superior de dicho bucle.

El algoritmo puede sufrir un pequeño refinamiento más mediante la utilización de una bandera, indicador o centinela, de forma que se detecte la presencia o ausencia de una determinada condición. En este caso la condición que interesa detectar de forma rápida es si en una pasada el conjunto de elementos ya se encuentra ordenado. La solución es muy sencilla: si durante una de las "pasadas" a los elementos del vector no se produce ningún intercambio de elementos, implica que los elementos del mismo están en orden y por lo tanto finaliza la ejecución del algoritmo devolviendo el resultado, aún cuando no se hayan realizado las $n-1$ "pasadas" al vector predeterminadas en el bucle *para* más externo.

El pseudocódigo que se muestra a continuación recoge las dos mejoras propuestas para el algoritmo:

Entrada: Conjunto de n elementos.
Salida: Conjunto de n elementos cumpliendo una relación de orden total.

```
1:  Finalizar := Falso
2:  Mientras ((j ⩽ n − 1) y (no Finalizar) hacer
3:          Finalizar := Verdadero
4:          Para i := 1 hasta n − j  hacer
5:                  Si (Elem[i] > Elem[i + 1]) entonces
6:                          Intercambiar (Elem[i], Elem[i + 1])
7:                          Finalizar := Falso
8:                  fin Si;
9:          fin Para;
10:         j := j + 1
11: fin Mientras;
```

Algoritmo 8: Pseudocódigo algoritmo de intercambio o *Burbuja* mejorado

Apéndice H

Ordenación por *Inserción*

El algoritmo de ordenación por *Inserción[19]*, por ser la forma habitual de colocar las cartas por parte de los jugadores, es conocido también con el nombre de *método de la baraja.* Este método consiste en insertar un elemento en una colección de elementos parcialmente ordenada obteniendo un nuevo conjunto de elementos también ordenado. Repitiendo el proceso con todos los nuevos elementos que aparecen, el resultado final es un conjunto ordenado de elementos en el que parcialmente se cumple la propiedad de orden para los elementos insertados.

La figura H.1 muestra un ejemplo de una lista de elementos desordenada.

Elem[1]	Elem[2]	Elem[3]	Elem[4]	Elem[5]	Elem[6]	Elem[7]	Elem[8]
5	14	24	39	43	65	84	45

FIGURA H.1: Lista de elementos desordenada

Como se observa, el conjunto presentado se encuentra parcialmente ordenado excepto por el elemento contenido en la octava posición *(Elem[8])*. Si se quiere insertar dicho elemento cuyo valor es 45, debe ser situado en una posición en la que tenga a su izquierda el elemento con valor 43 y a su derecha el elemento con valor 65. Esto supone tener que desplazar a la derecha todos aquellos elementos cuyo valor es superior a 45, es decir, el elemento con valor 45 debe "saltar" sobre los elementos con valores 65 y 84, debiendo ser estos desplazados una posición a la derecha. En la figura H.2 se muestra de forma gráfica el lugar de dicha inserción.

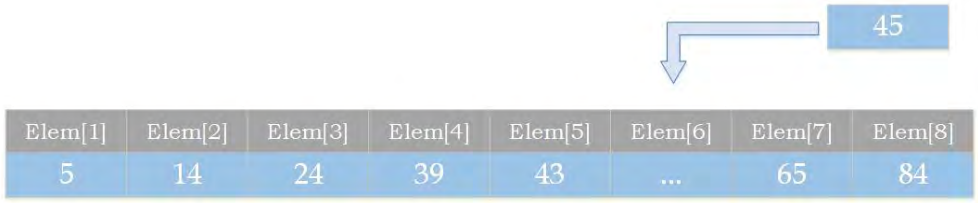


FIGURA H.2: Insertando un elemento en una lista parcialmente ordenada

El método de ordenación, por tanto, se basa en realizar comparaciones del elemento a insertar con los que componen el conjunto de elementos ya ordenado y realizar los desplazamientos necesarios de estos hasta que el nuevo elemento ocupe la posición que le corresponde. El algoritmo de clasificación de un vector A que contenga n elementos se realiza recorriendo todo el vector, realizando la inserción de cada elemento en el lugar que le corresponde desplazando el resto de elementos hacia posiciones superiores. Se comienza por un conjunto que dispone de un único elemento considerando que se encuentra ordenado. El proceso de comparación se realiza desde el segundo elemento del conjunto al enésimo. Un pequeño pseudocódigo que describe el proceso es el siguiente:

Entrada: Conjunto de n elementos.
Salida: Conjunto de n elementos cumpliendo una relación de orden total.
1: **Para** $i := 2$ *hasta* n **hacer**
2: *Insertar* $A[i]$ *en el lugar adecuado*
3: *Entre* $A[1]$ .. $A[i-1]$
4: **fin Para**;

ALGORITMO 9: Pseudocódigo algoritmo de ordenación por *Inserción*

La acción *Insertar* se realiza tantas veces como elementos se disponga menos uno. Esta acción se realiza de forma más sencilla con la inclusión de un valor centinela o bandera.

H.0.1. *Inserción binaria*

El algoritmo de inserción directa es mejorado de forma inmediata cambiando la forma de búsqueda del lugar que ocupa el elemento a insertar. Si en lugar de realizar una búsqueda secuencial de la posición que corresponde al elemento se utiliza el método de búsqueda binaria, se encuentra, por norma general, más rápidamente el lugar en el que ha de producirse la inserción

del nuevo elemento. El algoritmo de *Inserción binaria* es el que incluye este pequeño cambio.

H.0.2. Número de comparaciones

El cálculo del número de comparaciones $F(n)$ que realiza el algoritmo de *Inserción* es un cálculo sencillo.

Supongamos que se dispone de un vector que contiene n elementos para ordenar y nos situamos en el elemento que ocupa la posición $1 < x \leqslant n$. En esta situación, los $x-1$ elementos anteriores se encuentran ordenados ascendentemente según la clave de ordenación predeterminada.

Si la clave del nuevo elemento a insertar es mayor que todas las ya ordenadas el algoritmo sólo ejecuta una comparación, y, directamente, el elemento se sitúa en la posición x que ocupa actualmente. Si la clave es inferior a todas las restantes, el algoritmo ejecuta $x-1$ comparaciones hasta encontrar la posición del elemento que, en este caso, es la primera. El número medio de comparaciones es de $x/2$.

Si cuando comienza la ejecución se dispone de todos los elementos del vector en orden, el número de comparaciones que se realizan es mínimo e igual a $(n-1)$.

Si por contra, al comenzar la ejecución se encuentran todos los elementos en orden inverso a la solución buscada, se tiene que realizar un número de comparaciones máximas que es igual a

$$\frac{n(n-1)}{2}$$

Como $(n-1)+(n-2)+...+3+2+1 = \dfrac{n(n-1)}{2}$ es una progresión aritmética, esto da como resultado el siguiente valor para el número de comparaciones medias:

$$\frac{(n-1)+(n-1)\frac{n}{2}}{2} = \frac{n^2+n-2}{4}$$

Apéndice I

Algoritmo de ordenación *Shell*

El algoritmo de ordenación *Shell[19]* realiza una mejora inmediata al método de inserción directa cuando el número de elementos a ordenar es grande. El método utilizado recibe el nombre "Shell" en honor a su inventor, Donald Shell, aunque también se denomina *Inserción con incrementos decrecientes.*

En el método de clasificación por inserción descrito en el Apéndice H, cada elemento es comparado con los elementos contiguos de su izquierda de forma secuencial hasta que se encuentra la posición que ha de ocupar el nuevo elemento. Si el elemento que se desea insertar es más pequeño que todos los que se encuentran a su izquierda, hay que ejecutar un número máximo de comparaciones para obtener la posición definitiva del elemento, que en este caso es la primera de todas.

Shell modificó los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño, con lo que se obtiene una clasificación de los elementos más rápida. El método se basa en fijar el tamaño de los saltos con un valor constante, siendo dicho valor de salto superior a una posición.

Supongamos que se dispone de la siguiente lista de elementos (figura I.1), parcialmente ordenados:

Elem[1]	Elem[2]	Elem[3]	Elem[4]	Elem[5]	Elem[6]
4	12	16	24	36	3

FIGURA I.1: Insertando un elemento en lista parcialmente ordenada

Si se utiliza como método de ordenación el algoritmo de inserción directa, los saltos se realizan de posición en posición o, lo que es lo mismo, con

un valor de *salto* = 1. Por lo tanto para este caso concreto son necesarias 5 comparaciones. Si se utilizase el algoritmo de *Shell* con un valor de *salto* = 2, se realizan 3 comparaciones.

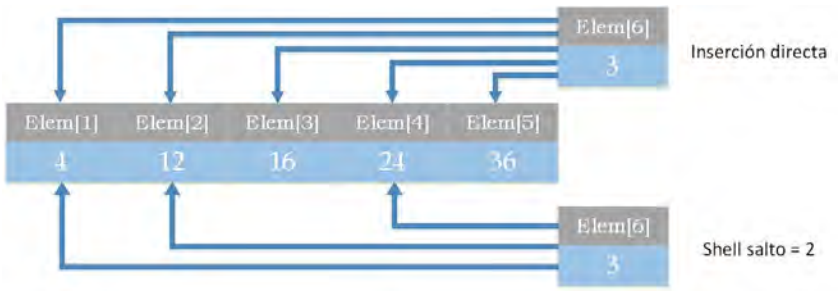


FIGURA I.2: Comparación inserción directa vs *Shell*

Si el número de elementos a ordenar es n el método toma como valor de salto $\frac{n}{2}$. Posteriormente se reduce el valor del *salto* mediante incrementos decrecientes hasta que llegue a tomar un valor mínimo igual a la unidad.

A diferencia del algoritmo de ordenación por inserción, este algoritmo intercambia elementos distantes entre sí. Este es el motivo por el que puede deshacer más de una inversión en cada intercambio, hecho que se aprovecha de una manera muy inteligente para incrementar notablemente la velocidad de ejecución.

El algoritmo en pseudocódigo queda expresado de la siguiente forma:

Entrada: Conjunto de n elementos.
Salida: Conjunto de n elementos cumpliendo una relación de orden total.

```
1:  Constante
2:          n = valor;
3:  Tipo
4:          array [1..n] de entero : lista;
5:  variables
6:          lista : l;
7:          entero : salto, inferior, superior;
8:
9:  Inicio
10: LlamaraLlenarListaElementos(l);
11: salto := n div 2;
12: Mientras (salto > 0) hacer
13:             inferior := 1;
14:             superior := inferior + salto;
15:             Mientras (Superior ⩽ n) hacer
16:                     Si (l[inferior] > l[superior]) entonces
17:                             Intercambiar (l[inferior], l[superior])
18:                     fin Si;
19:                     inferior := inferior + 1;
20:                     superior := superior + 1;
21:             fin Mientras;
22:             salto := salto div 2;
23: fin Mientras;
24: fin
```

Algoritmo 10: Pseudocódigo algoritmo ordenación *Shell*

Apéndice J

Algoritmo de ordenación *Quicksort*

El método de ordenación rápida[16] *Quicksort*, ordena o clasifica un conjunto de elementos almacenados habitualmente en un vector o lista de elementos (array). Se basa en el hecho de que es más sencillo y rápido ordenar 2 subconjuntos pequeños de elementos que un único conjunto que contenga a ambos.

Se denomina *método de ordenación rápida* porque, en general, puede ordenar un conjunto de elementos de forma más rápida que otros métodos explicados en los apéndices precedentes. Este método se debe a Hoare.

El algoritmo se basa en la estrategia típica de *divide y vencerás* (divide and conquer). El conjunto de elementos a clasificar almacenado en la estructura de datos que corresponda (por ejemplo en un vector) se divide en dos sublistas: una contiene todos los elementos menores o iguales que un determinado valor específico y, otra parte, todos los valores que sean mayores que dicho valor. El elemento elegido para partir el conjunto en dos es cualquier elemento arbitrario del vector y recibe el nombre de *pivote.*

El primer paso es, por tanto, elegir el valor que se toma para "partir" el conjunto de elementos en 2. De esta forma el conjunto original queda dividido en 3 partes. Si se considera que los elementos están almacenados en un *vector* V, este queda "partido" de la siguiente manera:

- *Subvector VI,* conteniendo todos los valores inferiores o iguales al "pivote".
- El "pivote" o elemento de separación.
- *Subvector VD,* conteniendo los valores superiores al "pivote".

Los *subvectores VI y VD* no están ordenados, excepto en el caso de que el número de elementos sea igual a 1.

Consideremos la lista de valores que se muestra en la figura J.1.

Elem[1]	Elem[2]	Elem[3]	Elem[4]	Elem[5]	Elem[6]	Elem[7]
18	11	27	13	9	4	16

FIGURA J.1: Lista de elementos a ordenar

La primera tarea que se debe realizar es la de elegir el pivote. Para la primera partición de este ejemplo se toma como "pivote" el elemento con valor 13, por ser el que está situado en el centro de la lista de valores.

El siguiente paso, es recorrer la lista de elementos desde el extremo izquierdo buscando un elemento con valor superior a 13, por lo que se encuentra el elemento con valor 18. A continuación se busca desde el extremo derecho del vector un valor menor que 13: se encuentra el elemento 4. Con ambos elementos la acción a realizar es intercambiarlos, obteniendo como resultado una nueva re-ordenación de los elementos de la lista. La figura J.2 muestra la nueva situación.

Elem[1]	Elem[2]	Elem[3]	Elem[4]	Elem[5]	Elem[6]	Elem[7]
4	11	27	13	9	18	16

FIGURA J.2: Elementos tras el primer intercambio

Se continua recorriendo el vector por la izquierda buscando un elemento con valor superior a 13, y se encuentra el elemento 27. En el recorrido desde la derecha se busca un valor menor que 13: se localiza el 9. Nuevamente con ambos valores localizados se repite la operación de intercambiar de posición a ambos, obteniendo la lista de elementos que se muestra en la figura J.3.

Elem[1]	Elem[2]	Elem[3]	Elem[4]	Elem[5]	Elem[6]	Elem[7]
4	11	9	13	27	18	16

FIGURA J.3: Elementos tras el segundo intercambio

Al realizar las mismas acciones una vez más, las exploraciones de los 2 extremos vienen juntas sin encontrar ningún futuro valor que esté "fuera de lugar". En este punto se sabe que todos los valores a la derecha del "pivote" son mayores que todos los situados a su izquierda. Por tanto se ha realizado una partición de la lista original que ha quedado dividida en 2 sublistas más pequeñas, representadas en la figura J.4.

FIGURA J.4: Elementos divididos por el pivote

Ninguna de las 2 sublistas se encuentra ordenada; sin embargo, repitiendo el mismo proceso utilizado para obtener la primera partición, se pueden ordenar las 2 particiones de forma independiente. El elemento pivote se encuentra en la posición que debe ocupar en la relación de orden final. Ordenando ambas sublistas mediante la misma estrategia y colocando en el centro el elemento "pivote", se obtiene la lista que cumple la relación de orden total (se representa el resultado final en la figura J.5).

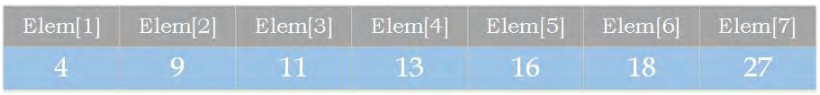


FIGURA J.5: Resultado final: elementos ordenados

La eficiencia del algoritmo depende fundamentalmente de la elección del elemento "pivote". En cada selección la mejor elección posible para el "pivote" es aquella que divide el conjunto de elementos en 2 subconjuntos de tamaño lo más similar posible. El peor de los casos es, lógicamente, la elección de un "pivote" en el que todos los elementos contenidos en un conjunto sean mayores o menores que este, porque uno de los subconjuntos se queda vacío.

El análisis general de la eficiencia de *Quicksort* es difícil. La mejor forma de ilustrar y calcular su complejidad es considerar el número de comparaciones realizadas suponiendo condiciones ideales. Supóngase que n (número de elementos de la lista) es una potencia de 2, $n = 2^k \quad (k = \log_2 n)$. Además, supongamos que el pivote es el elemento central de cada lista de modo que *Quicksort* divide la lista en 2 sublistas de tamaño similar.

En la primera exploración o recorrido se realizan $n-1$ comparaciones. El resultado de esta etapa crea 2 sublistas con tamaño aproximado $n/2$. En la siguiente fase, el proceso de cada sublista requiere aproximadamente de $n/2$ comparaciones. Las comparaciones totales de esta fase son $2\ n/2 = n$. La siguiente fase procesa cuatro sublistas que requieren un total de $4\ n/4 = n$ comparaciones, etc. Eventualmente el proceso de división termina después de k pasadas cuando la sublista resultante tiene un tamaño igual a 1. El número total de comparaciones es aproximadamente:

$$n + 2\,\frac{n}{2} + 4\,\frac{n}{4} + ... + n\,\frac{n}{n} = n + n + \ ... \ + n = n * k = n\ \log_2 n$$

El caso ideal examinado se produce cuando los elementos del conjunto se encuentran ordenados de forma ascendente. Para este caso concreto, el "pivote" es precisamente el elemento central en cada sublista. La ejecución del algoritmo sobre el conjunto ordenado de elementos de esta forma presenta una complejidad $O(n \log_2 n)$.

El escenario del peor caso para *Quicksort* se produce cuando el pivote cae en una sublista de un elemento y deja el resto de elementos para la otra sublista. Esto se produce cuando el pivote es siempre el elemento menor (o mayor) de una sublista. En el recorrido inicial hay m comparaciones y la sublista con elementos contiene $m-1$ elementos. En el siguiente recorrido, si sucede lo mismo, la mayor sublista requiere de $m-1$ comparaciones y produce una sublista que contendrá $m-2$ elementos, etc. El número total de comparaciones por tanto es:

$$m + (m-1) + (m-2) + \ ... \ + 2 \ = \ (m-1)(m-2)/2$$

Esto equivale a expresar que la complejidad es $O(n^2)$. Este es el principal inconveniente de utilizar el algoritmo *Quicksort*: su peor situación es catastrófica. Es verdad que en término medio la velocidad del algoritmo tiene como complejidad $O(n \log_2 n)$ siendo posiblemente uno de los algoritmos más rápidos, pero no es menos cierto que en el peor de los casos su tiempo de ejecución es desastroso.

Una primera aproximación al algoritmo de partición consistente en explorar desde cada extremo el conjunto de elementos, intercambiando los valores encontrados en caso de ser necesario, es el siguiente:

Entrada: Conjunto de n elementos.
Salida: Conjunto de n elementos cumpliendo una relación de orden total.

```
1: Algoritmo partición
2: Inicio
3:         Establecer x al valor de un elemento arbitrario de la lista
4: Mientras división no esté terminada hacer
5:             recorrer de izquierda a derecha para un valor ⩾ x
6:             recorrer de derecha a izquierda para un valor < x
7:             Si los valores localizados no están ordenados entonces
8:                     IntercambiarValores
9:             fin Si;
10: fin Mientras;
11: finAlgoritmo
```

Algoritmo 11: Pseudocódigo *Quicksort* versión inicial

Supongamos que la lista que se desea partir es $A[1],\ A[2],\ \ldots,\ A[n]$. Los índices que representan los extremos izquierdo y derecho de la lista son `L` y `R`. En el refinamiento del algoritmo se elige un valor arbitrario x suponiendo que el valor central de la lista es tan bueno como cualquier elemento arbitrario, y tan malo como cualquier otro. Los índices $i,\ j$ exploran desde los extremos el conjunto de elementos. Un posible refinamiento del anterior algoritmo que incluye un mayor número de detalles es el siguiente:

Entrada: Conjunto de n elementos.
Salida: Conjunto de n elementos cumpliendo una relación de orden total.

```
1:  Algoritmo Quicksort
2:  Variables
3:  entero : temporal, i, j, medio;
4:  i := inferior;
5:  j := superior;
6:
7:  Inicio
8:  Si (superior > inferior) entonces
9:          i:=inferior;
10:         j:=superior;
11:         medio:= A[(inferior + superior) div 2];
12:         Mientras (i < j) hacer
13:                 Mientras ((i < superior) y (A[ i ] < medio)) hacer
14:                         i:= i + 1;
15:                 fin Mientras;
16:                 Mientras ((j > inferior) y (A[ j ] > medio)) hacer
17:                         j:= j + 1;
18:                 fin Mientras;
19:                 Si (i ⩽ j) entonces
20:                         temporal:= A[i];
21:                         [i]:= A[j];
22:                         A[j]:= temporal;
23:                         i:= i + 1;
24:                         j:= j + 1;
25:                 fin Si;
26:         fin Mientras;
27: fin Si;
28: Si (inferior < j) entonces
29:         Quicksort(A, inferior, j);
30: fin Si;
31: Si (j < superior) entonces
32:         Quicksort(A, i, superior);
33: fin Si;
34: finAlgoritmo
```

Algoritmo 12: Pseudocódigo algoritmo *Quicksort* detallado

Apéndice K

Algoritmo de ordenación *Heap*

El algoritmo de ordenación *Heap[20]* es un algoritmo que se construye utilizando las propiedades de los montículos binarios. Se incluye entre la familia de algoritmos de ordenación como el de selección. Su orden de ejecución, como se verá posteriormente, es de $O(n \log n)$ siendo n el número de elementos que se van a tratar.

A modo de breve recordatorio, un montículo Max es un árbol binario completo cuyos elementos están ordenados del siguiente modo: para cada subárbol se cumple que el valor de la *raíz* es mayor que el de sus nodos *hijo.* Si por el contrario el montículo fuera *Min*, la raíz de cada subárbol debe tener un valor menor que el correspondiente a sus descendientes.

Recordemos que si bien un montículo se define como un árbol, para representar este se puede utilizar un array de datos, en el cual se accede a *padres* e *hijos* utilizando las siguientes transformaciones sobre sus índices: si el montículo se encuentra almacenado en el *Array A,* el nodo padre de $A[\,i\,]$ ocupa la posición $A[\,\frac{i}{2}]$ (truncado por abajo), el hijo izquierdo se encuentra en la posición $A[\,2 \times i\,]$, y el hijo derecho en $A[\,2 \times i + 1\,]$.

Al insertar o eliminar elementos de un montículo hay que comprobar que se sigue manteniendo la propiedad de orden del montículo. Lo que se hace habitualmente es construir rutinas de filtrado (que son ascendentes o descendentes) que toman un elemento del montículo (aquel elemento que viola la propiedad de orden) moviéndolo verticalmente por el árbol hasta encontrar aquella posición en la que se respete el orden entre los elementos del montículo.

Tanto la inserción como la eliminación (*eliminar_min* o *eliminar_max* según sea un montículo Min o Max respectivamente) de un elemento en un montículo se realizan en un tiempo $O(\,n \log n)$ en el peor caso (esto es debido al orden entre sus elementos y a la característica de árbol binario completo).

Para realizar un cálculo rápido de la eficiencia del algoritmo se contemplan 2 fases:

1.- Construcción inicial del montículo (heap): n operaciones de inserción con una complejidad del orden $O(\log n)$ sobre un árbol que ya se encuentra ordenado, por lo que la complejidad es del orden $O(n \log n)$.

2.- Extracción del mayor/menor elemento del montículo: Se realizan n operaciones de borrado cuya complejidad es del orden $O(\log n)$, sobre un árbol previamente ordenado $O(n \log n)$.

Dicha combinación produce como resultado una complejidad para el algoritmo de ordenación *Heap* de $O(n \log n)$.

Estrategia general del algoritmo: Existen 2 razones por las que la estrategia general ha de ser refinada. La primera es la utilización de un *TAD* auxiliar (montículo binario), lo cual puede implicar demasiado código para un simple algoritmo de ordenación. La segunda versa sobre una posible necesidad de memoria adicional para el montículo y para una posible copia del array.

Lo que se trata de conseguir es una máxima reducción del código minimizando lo más posible la abstracción a un montículo. Una vez que se han obtenido los elementos a tratar, estos no son copiados a un montículo (insertando cada elemento). Lo que se hace es, a cada elemento de la mitad superior del array, se le aplica un filtrado *descendente*, "bajando" cada elemento por el árbol binario hasta que se disponga de 2 *hijos* que cumplan con la condición de orden del montículo: esto es suficiente para hacer que el array cumpla con ser un montículo binario.

Nótese también que al ejecutar un *eliminar_max* se obtiene el primer elemento de la ordenación (o el último, según el criterio establecido) liberando un hueco. Se utiliza esta estrategia para evitar tener que hacer una copia del array en la que meter las eliminaciones sucesivas. Para ello se almacena la salida de *eliminar_max* después del último elemento del montículo. Esto hace que al ejecutar $n - 1$ veces *eliminar_max* se obtenga el array ordenado de menor a mayor.

Para realizar una reducción del código, se pretende lograr *insertar* y *eliminar_max* con una sola rutina de filtrado descendente. Ya se ha explicado cómo hacer que el array de datos mantenga el orden de los elementos del montículo: lo que se hace para ordenarlo usando sólo la rutina de filtrado descendente es ejecutar un *eliminar_max* intercambiando el primer elemento del array por el último del montículo, y filtrar el nuevo primer elemento hasta que se respete el orden *Max.* El pseudocódigo que se muestra a continuación pretende dar una visión más clara de las explicaciones realizadas:

```
1:  Algoritmo Heap
2:
3:  FiltradoDescendente (Dato *A, int i, int N)
4:  {Queremos que se respete el orden MAX del montículo}
5:      Dato tmp := A[i]
6:      int hijo := 2 * i;
7:  Si ((hijo < N) y (A[hijo + 1] > A[hijo])) entonces
8:          hijo := hijo + 1;
9:  fin Si
10: Mientras ((hijo ≤ N) y (tmp < A[hijo])) hacer
11:         Si ((hijo < N) y (A[hijo + 1] > A[hijo])) entonces
12:                 hijo := hijo + 1;
13:         fin Si
14:         A[i] := A[hijo];
15:         i := hijo;
16:         hijo := 2 * i;
17: fin Mientras
18:     A[i] := tmp;
19: FinFiltradoDescendente
20:
21: Intercambiar (Dato *A, int i, int j)
22:     Dato tmp := A[i];
23:     A[i] := A[j];
24:     A[j] := tmp;
25: FinIntercambiar
26:
27: Heap (Dato *A, int N)
28: int i;
29: {Se introducen los datos en el montículo}
30: Para (i := n/2; i ≥ 0; i := i − 1) hacer
31:         FiltradoDescendente (A, i, N);
32: fin Para
33: {Se sacan datos y se guardan al final para obtener el array ordenado}
34: Para (i := N − 1; i > 0; i := i − 1) hacer
35:         Intercambiar (A, 0, i);
36:         FiltradoDescendente (A, 0, i − 1);
37: fin Para
38: finAlgoritmo
```

Algoritmo 13: Pseudocódigo algoritmo *Heap*

A continuación se muestra un pequeño ejemplo explicativo sobre el funcionamiento del algoritmo. Lo primero que debe realizarse es la carga de los elementos que componen la estructura de árbol, de tal forma que una vez insertado cada nuevo elemento en la estructura se debe comprobar que se mantiene el criterio del montículo para todo el árbol. En el siguiente ejemplo se describe paso a paso un pequeño ejemplo con una estructura de árbol *Max,* lo que se traduce en que el valor de cada elemento de cada nodo debe ser siempre igual o superior al de sus hijos. Los valores y el orden inicial de los mismos se muestran en la figura K.1.

FIGURA K.1: Conjunto de elementos a ordenar

La primera acción que se ha de realizar es la de construir la estructura de árbol que disponga de la propiedad de montículo *Max.* De esta forma, cada vez que se inserta un elemento en la estructura de árbol se ha de comprobar si la propiedad se mantiene (en cuyo caso no ha de hacerse nada), o si por el contrario se ha perdido y debe realizarse un intercambio de elementos en el árbol. Una vez tratados los elementos anteriores siendo tomados de izquierda a derecha, la estructura de árbol construida es la que se muestra en la figura K.2.

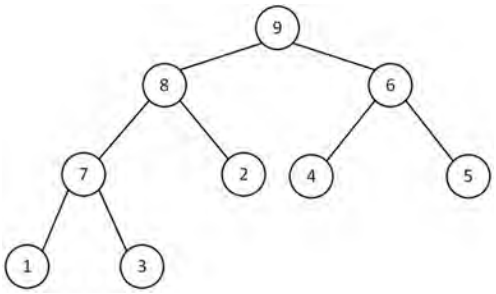


FIGURA K.2: Árbol montículo *Max* inicial

Como se aprecia, al mantener la estructura definida se tiene en el nodo raíz el elemento de mayor valor de todos los insertados. De esta forma dicho elemento es sacado de la estructura. ¿Cuál es el siguiente paso?. Colocar en la raíz del árbol el elemento cuya posición se encuentre lo más profundo y a la derecha posible en la anterior estructura. En esta ocasión dicho elemento

es el que tiene valor 3. La figura K.3, muestra de forma gráfica la situación descrita.

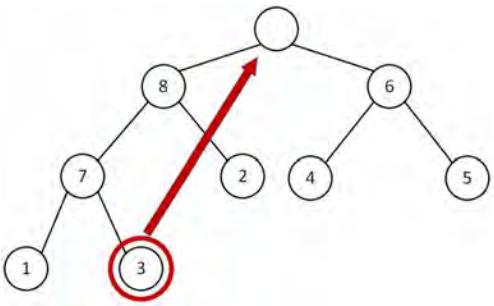


FIGURA K.3: El elemento más profundo y a la derecha es enviado a la raíz

Una vez colocado dicho elemento en el nodo raíz se procede con la tarea de mantenimiento que permite reorganizar nuevamente los elementos del árbol para que este siga conservando la propiedad del montículo. Para ello al insertar el elemento con valor 3, se comprueba que su primer hijo con valor 8 es mayor que él, por lo que se procede a realizar el intercambio de ambos elementos obteniendo como nodo raíz el de valor 8. Se continua descendiendo por el árbol y nuevamente el elemento con valor 3 es comparado con su primer hijo, resultando de la comparación el elemento 3 como menor, por lo que ambos nodos intercambian sus valores 3 y 7 respectivamente como se muestra en la figura K.4.

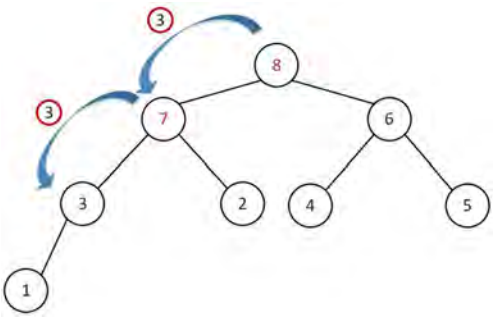


FIGURA K.4: Intercambia elemento 3 para mantener el invariante

En la siguiente comparación todos los hijos del nodo 3 son menores que él. Finalizan los intercambios y el árbol vuelve a disfrutar de la propiedad del montículo *Max,* lo que permite finalizar las tareas rutinarias de mantenimiento

y afirmar que el siguiente elemento en la relación de orden es el que se encuentra en la raíz del árbol, en este caso el de valor 8.

Dicho elemento es extraído de la estructura de árbol y pasa a ocupar su lugar el elemento que se encuentra en la posición más profunda y a la derecha del árbol, en este caso el que tiene el valor 1, situándose en la raíz del árbol donde nuevamente se procede a realizar una reorganización de los elementos que mantenga la propiedad de montículo *Max.* Siendo este un proceso repetitivo, las figuras K.5 y K.6 muestran los diferentes nodos con sus valores de manera que queden detallados todos los pasos de la ejecución del algoritmo.

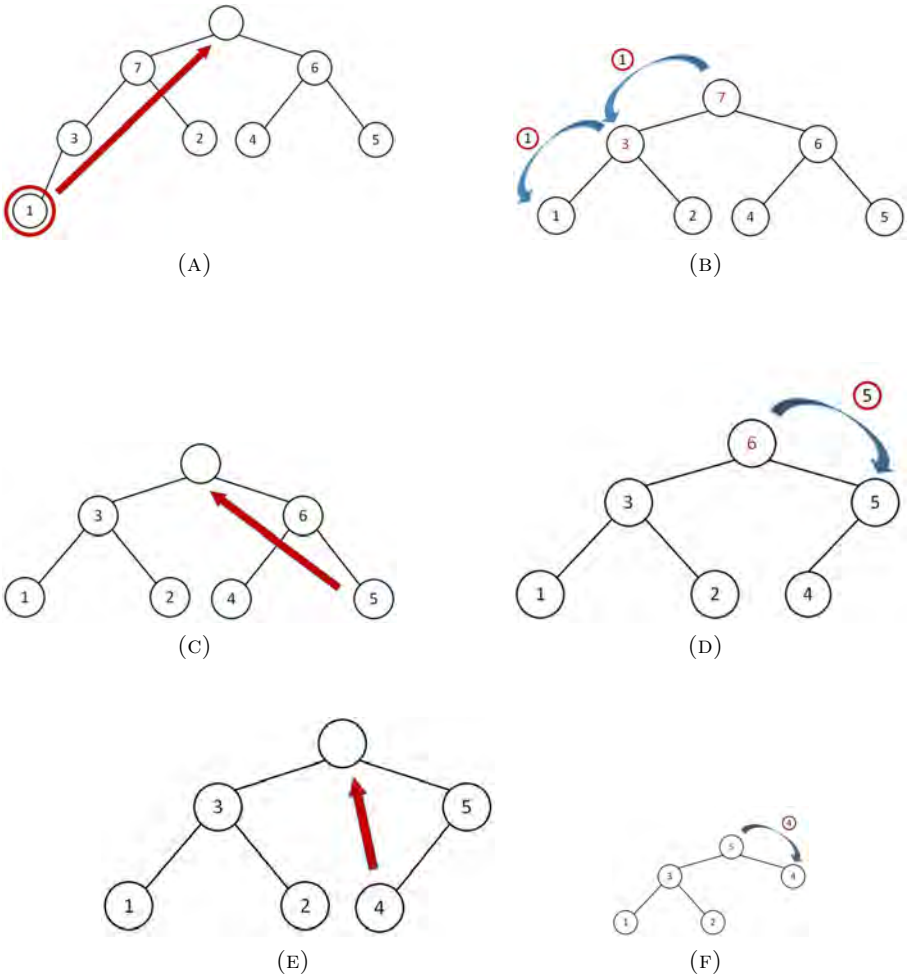


FIGURA K.5: Ejemplo de ordenación con algoritmo *Heap*

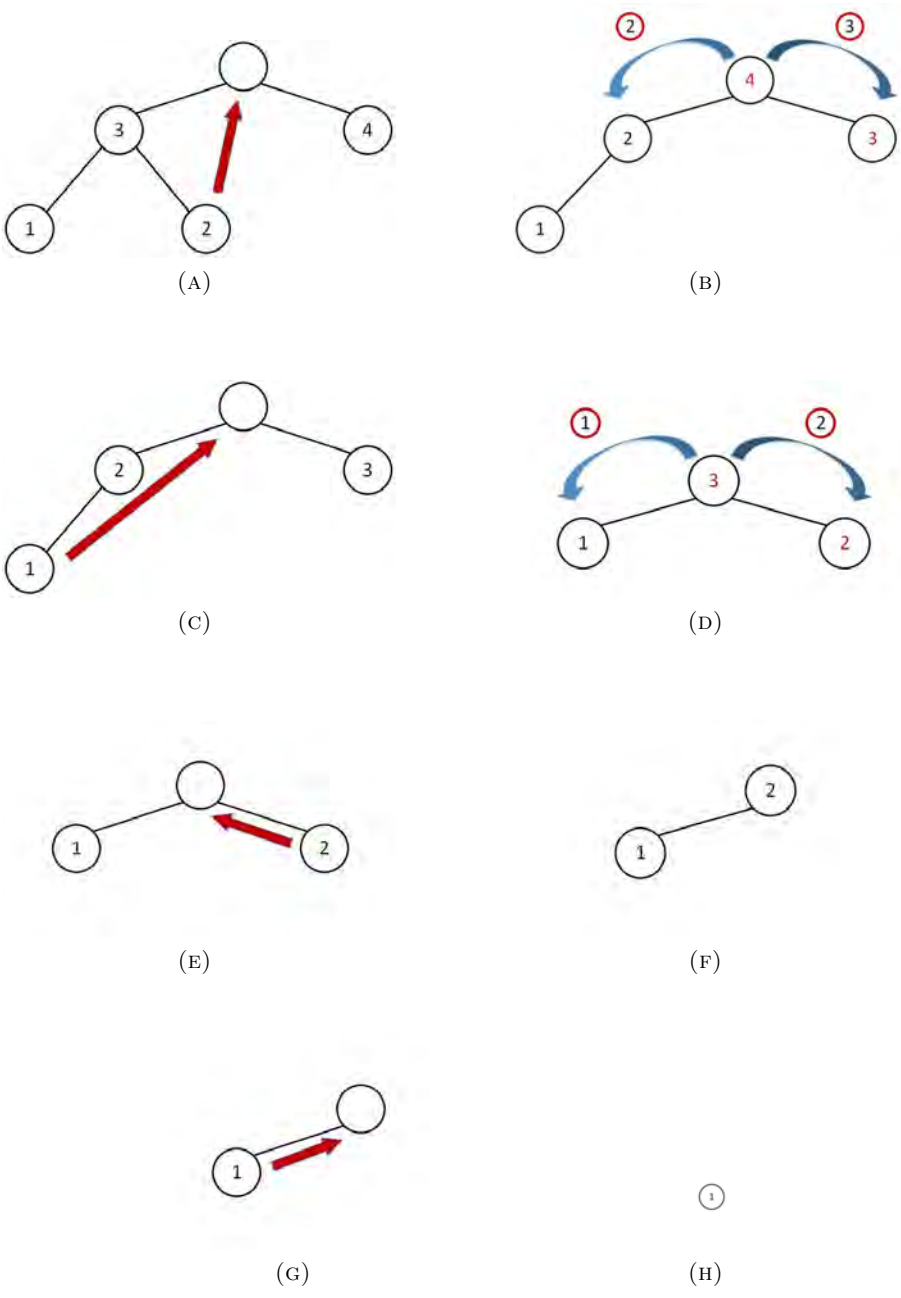


FIGURA K.6: Continuación ejemplo de ordenación con algoritmo *Heap*

Apéndice L

Algoritmo de ordenación *Merge*

El algoritmo de ordenación *Merge* fue desarrollado en 1945 por John Von Neumann. Es un algoritmo recursivo con un número de comparaciones mínimo cuyo tiempo de ejecución promedio es $O(\, n \log n)$. La desventaja del algoritmo es que ha de trabajar con un array auxiliar lo que produce 2 consecuencias negativas: utilización de memoria extra e incremento en el tiempo de ejecución por el trabajo añadido para realizar copias entre arrays (aunque es un trabajo con un tiempo de ejecución lineal).

Conceptualmente la ordenación por *Fusión* opera de la siguiente manera:

1.- Si la longitud de la lista es 0 o 1, la lista está ordenada. En caso contrario:

2.- Se divide la lista desordenada en 2 sublistas con aproximadamente la mitad de elementos cada una.

3.- Ordenar cada sublista de forma recursiva, aplicando la ordenación por *Fusión*.

4.- Mezclar las 2 sublistas en una sola lista ordenada.

La ordenación por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

a) Una lista pequeña necesita menos pasos para ser ordenada que una lista grande.

b) Son necesarios menos pasos para fusionar dos sublistas que ya estén en orden que 2 sublistas que se encuentren completamente desordenadas. Sólo es necesario entrelazar cada sublista una vez que ambas se encuentran ordenadas.

```
1: Algoritmo MergeSort (array A[x..y])
2: Si (y − x > 1) entonces
3:         Array A1 := MergeSort( A[ x .. (int ( (x + y)/2 ))])
4:         Array A2 := MergeSort( A[ int ( 1 + ( (x + y)/2 )) .. y])
5:         Devolver Merge ( A1, A2)
6: Si no
7:         returnA
8: fin Si
9: finMergeSort
10:
11: Función Merge(array A1[0..n], array A2[0..m])
12: entero z:=0;
13: entero t:=0;
14: {Suponiendo que n y m son las posiciones del array}
15: array R[0 .. (n + m + 2)];
16:
17: Mientras ((z ≤ n) o (t ≤ m)) hacer
18:         Si ((z ≤ n) y (A1[z] ≤ A2[t])) entonces
19:                 R[z + t]:= A1[z];
20:                 z:= z+1;
21:         Si no
22:                 Si ((t ≤ m) y (A1[z] > A2[t])) entonces
23:                         R[z+t]:= A2[t];
24:                         t:= t+1;
25:                 fin Si
26:         fin Si
27: fin Mientras
28:
29: DevolverR
30: finMerge
```

Algoritmo 14: Pseudocódigo algoritmo *Merge*

Apéndice M

Algoritmo de ordenación utilizando árbol *Binario*

El algoritmo de ordenación mediante la utilización de árbol *Binario*, se apoya en la construcción de un árbol binario de búsqueda para devolver los elementos en el orden que se desea. La ejecución del algoritmo consta de dos fases:

- Primera fase: Construcción del árbol binario de búsqueda que contiene en cada nodo un elemento del conjunto a ordenar.
- Segunda fase: Recuperación de resultados. Recorrido en inorden del árbol binario construido.

  El árbol binario de búsqueda se define de la siguiente manera:

- O bien es una estructura vacía,
- O bien es un nodo con a lo sumo dos subárboles (izquierdo y derecho), disjuntos, que cumplen con las siguientes características:
  - Todas las claves del subárbol izquierdo del nodo son menores que la clave del nodo.
  - Todas las claves del subárbol derecho del nodo son mayores que la clave del nodo.
  - Ambos subárboles (izquierdo y derecho) son árboles binarios de búsqueda.

La forma de proceder en la primera fase de creación del árbol binario de búsqueda es la siguiente:

1.- Se toma un elemento del conjunto a tratar. Si no existe nodo raíz con el que compararlo se crea el nodo, pasando a ocupar dicho elemento esa posición y se toma el siguiente elemento.

2.- Si existe nodo con el que compararlo se realiza dicha comparación. Si de la misma resulta que el elemento es mayor que el nodo, es enviado a su subárbol derecho. En caso contrario, se envía al subárbol izquierdo.

3.- Se repite proceso de comparación del punto 2 hasta que este encuentre el hueco en el árbol en que ha de ser insertado y se selecciona el siguiente elemento.

4.- El proceso de creación finaliza cuando el conjunto de elementos a tratar sea igual a $\emptyset$.

Insertar elementos en un árbol binario de búsqueda presenta una complejidad de $O(\log n)$. Por lo tanto, insertar n elementos presenta una complejidad de $O(n \log n)$.

Finalizada la construcción del árbol binario de búsqueda, basta con realizar un recorrido en inorden del mismo para recuperar el conjunto completo de elementos en el orden deseado (en este caso de menor a mayor).

Recorrer los elementos del árbol presenta una complejidad de $O(n)$. Por lo tanto puede decirse que en general el rendimiento de este algoritmo es bueno. Su eficiencia depende del orden de tratamiento de los elementos y de la posición final que ocupen en la estructura de datos: es mejor cuanto más equilibrado sea el árbol que se construye, siendo el peor caso cuando todos los elementos se van siempre por la misma rama para cada iteración, ya que la estructura de datos construida será un árbol binario de búsqueda con aspecto de lista lineal.

M.1. Especificación del *TAD* árbol binario

A continuación se detalla una posible especificación del tipo abstracto de datos árbol *Binario.*

- *TAD* - Árbol *Binario.*
- Valores - árboles binarios.
- Operaciones:
  - *CrearArbolBinario(e, h_1, h_2)* $\rightarrow$ *árbol Binario* - crea un nuevo árbol binario cuya raíz almacena el dato e y tiene como hijos h_1 y h_2, que pueden ser árboles vacíos.

- *EliminarArbolBinario(A)* - elimina el árbol A y sus descendientes.
- *EsVacio(A)* $\rightarrow$ *Boolean* - cierto si el árbol es vacío.
- *Datos(A)* $\rightarrow$ *elemento* - devuelve la información guardada en la raíz del árbol. Tiene como precondición que el árbol no sea vacío.
- *Izquierdo(A)* $\rightarrow$ *árbol Binario* - devuelve el subárbol binario izquierdo de A, que puede ser vacío.
- *Derecho(A)* $\rightarrow$ *árbol binario* - devuelve el subárbol binario derecho de A, que puede ser vacío.
- *EsHoja(A)* $\rightarrow$ *Boolean* - devuelve cierto si el nodo raíz de A no tiene descendientes.

M.2. Implementación

La implementación que se propone está basada en memoria dinámica. Cuando se trata de plasmar algoritmos que trabajan con árboles, resulta de gran utilidad la posibilidad de "devolver" un árbol como resultado. Esto es fácil de conseguir definiendo el tipo *árbol binario* como un puntero a una estructura de datos que representa el nodo raíz del árbol.

```
1:  Estructura de datos TAD árbol Binario
2:
3:  Constante
4:      ArbolBinarioVacio = Null;
5:
6:  Tipos
7:  {El árbol binario es en sí mismo un puntero a registro}
8:      ArbolBinario = ^TRegistroArbolBinario;
9:
10: {Datos almacenados en cada nodo del árbol}
11:     TElemento = Cadena
12:
13: {Árbol binario}
14:     TRegistroArbolBinario = registro
15:     Datos : TElemento;
16:     Hijos : array [1..2] de ArbolBinario;
17: FinTipos
18:
19:
```

Algoritmo 15: Estructura de datos *TAD* árbol binario

M.2.1. Liberar memoria ocupada por árbol binario

Muchas de las operaciones son implementadas mediante la utilización de recursividad. La propia estructura de datos *árbol* tiene un marcado carácter recursivo.

La operación que se encarga de liberar la memoria asociada a un árbol binario se implementa mediante técnicas recursivas. Si el árbol a eliminar A está vacío no hay que hacer nada. Sino, se eliminan sus subárboles y por último se libera la memoria asociada al nodo raíz de A. Para eliminar sus subárboles se utiliza una recursividad directa sobre la propia operación. El parámetro se pasa por referencia, lo que hace posible que desde el ámbito en que se encuentra el "cliente" el árbol pase a ser un *árbol vacío* cuando concluye la operación.

Entrada: Estructura de datos árbol binario.
Salida: Libera la memoria ocupada por la estructura de datos pasada.

```
1: Procedimiento EliminarArbolBinario (var A: ArbolBinario);
2: Inicio
3: Si (A <> ArbolBinarioVacio) entonces
4:         EliminarArbolBinario(Aˆ.Hijos[1]);
5:         EliminarArbolBinario(Aˆ.Hijos[2]);
6:         LiberarMemoria(A); A:=Null;
7: fin Si
8: fin
```

Algoritmo 16: Procedimiento para liberar la memoria de árbol binario

M.2.2. Recorrer un árbol binario

Para recorrer cualquier árbol (sea o no binario) existen múltiples formas de realizar la tarea. El recorrido *Inorden* se encuentra entre las más utilizadas. Además, es el método que utiliza el meta-algoritmo descendente (capítulo 4) para recuperar los elementos una vez ordenados en el árbol de procesos.

La forma de realizar las visitas a los nodos del árbol, sigue el orden predefinido de primero el subárbol izquierdo, después la raíz, y en último lugar el subárbol derecho. A continuación se muestra el pseudocódigo recursivo que permite realizar esta tarea.

```
Entrada: Estructura de datos árbol binario.
Salida: Visita en Inorden los nodos de un árbol binario.
1: Procedimiento Inorden (var A: ArbolBinario);
2: Inicio
3: Si (A <> ArbolBinarioVacio) entonces
4:         Inorden(HijoIzquierdo(A));
5:         VisitarNodo(A);
6:         Inorden(HijoDerecho(A));
7: fin Si
8: FinProcedimiento
```

ALGORITMO 17: Recorrido de árbol binario en *inorden*

M.3. Árbol binario de búsqueda

Un árbol binario de búsqueda es un árbol binario en el que cada nodo almacena un valor (*clave*), y que además cumple que para todo nodo las claves en su subárbol izquierdo son menores que su clave, y las de su subárbol derecho son mayores. Los árboles binarios de búsqueda no son únicos para los datos que contienen. Su organización depende del orden en que se insertan las claves o, lo que es lo mismo, del orden de llegada de los elementos.

M.3.1. Inserción de un elemento en árbol binario de búsqueda

Insertar un nuevo elemento en un árbol binario de búsqueda realiza la inserción siempre como una hoja del árbol. El valor o clave debe ser almacenado en el lugar que le corresponda, de forma que una vez realizada la inserción se mantenga la propiedad de árbol binario de búsqueda. A continuación se muestra una posible implementación mediante la utilización de técnicas recursivas.

```
Entrada: Árbol binario de búsqueda y el elemento a insertar.
Salida: Árbol binario de búsqueda con el elemento insertado.
 1: Procedimiento InsertarABB (var A: ABB; e : TElemento);
 2: Inicio
 3: Si (A <> ABBVacio) entonces
 4:         {Insertar en el subárbol que corresponda}
 5:         Segun Comparar(c, DatosABB(A)) hacer
 6:             '<': InsertarABB(A^.Hijos[1], e);
 7:             '>': InsertarABB(A^.Hijos[2], e);
 8:         FinSegun
 9: Si no
10:         {Se alcanza una hoja, A es vacío, insertar aquí}
11:         A:= CrearABB(c, ABBVacio, ABBVacio);
12: fin Si
13: FinProcedimiento
```

ALGORITMO 18: Insertar elemento en árbol binario de búsqueda

M.3.2. Buscar un elemento en un árbol binario de búsqueda

La búsqueda aprovecha la propiedad que define el árbol binario de búsqueda para elegir el camino a seguir a la hora de localizar un dato en el árbol. Por término medio se logra una operación mucho más eficiente que una búsqueda secuencial.

```
Entrada: Árbol binario de búsqueda y el elemento a encontrar.
Salida: Verdadero o Falso en función del resultado de la búsqueda.
 1: Procedimiento BuscarElABB (var A: ABB; e : TElem):Booleano;
 2: Inicio
 3: Si (A <> ArbolBinarioVacio) entonces
 4:         {Si no está en A, buscar en el subárbol adecuado}
 5:         Segun Comparar(c, DatosABB(A)) hacer
 6:           '<': BuscarElABB:= BuscarElABB(HijoIzqdo(A), c);
 7:           '>': BuscarElABB:= BuscarEl(HijoDrcho(A), c);
 8:           '=': BuscarElABB:= Verdadero;
 9:         FinSegun
10: Si no
11:         {Se alcanza una hoja sin encontrar el elemento}
12:         BucarElABB:= Falso;
13: fin Si
14: FinProcedimiento
```

ALGORITMO 19: Buscar un elemento en un *Árbol binario de búsqueda*

Apéndice N

Algoritmo de *Torneo*

El algoritmo clásico de *Torneo* se ha descrito y desarrollado de muy diversas formas a lo largo de la historia de la informática. Por este motivo se hace una descripción de las características generales comunes que se encuentran en cualquier implementación del mismo.

La comprensión de su funcionamiento es simple ya que se asemeja a la forma de disputar un torneo, por ejemplo de tenis. En un torneo de tenis los jugadores disputan el mismo enfrentándose de dos en dos hasta obtener un ganador final. Cuando se finaliza se conoce quien es el mejor tenista para ese torneo, pero se desconoce el orden final para el resto de jugadores (de hecho el otro jugador finalista no necesariamente es el segundo mejor jugador del torneo, puede serlo cualquiera de los jugadores que perdieron con el vencedor).

El algoritmo de ordenación *Torneo* realiza su ejecución de forma análoga: en este caso los jugadores se corresponden con el conjunto de elementos a ordenar, cada partido de tenis se corresponde con una comparación entre dos elementos del conjunto de entrada, y al finalizar el torneo se conoce el menor elemento del conjunto pero se ignora la relación de orden final entre todos los demás. Por este motivo, si se quiere ordenar n elementos mediante este algoritmo es necesario repetir el proceso de partidos (comparaciones) $n-1$ veces, conociendo un ganador en cada iteración, el cual es eliminado de los elementos a ordenar para la siguiente iteración puesto que ya ocupa el lugar en el orden que le corresponde.

El proceso de ordenación realizado por el algoritmo se puede decir que consta de dos fases diferenciadas:

- Una primera fase en la que se separa el conjunto de entrada formado por los n elementos hasta obtener todos los conjuntos disjuntos formados por un único elemento cada uno (la unión de todos esos subconjuntos da como resultado el conjunto de entrada que contiene los n elementos).

- Una segunda fase en la que los elementos compiten para "ganar" el torneo.

En la primera fase, puede ser visto como si se dispone de un árbol binario y todos los elementos del conjunto de entrada son distribuidos únicamente en las hojas de este. Dicho árbol binario dispone de n niveles.

Tras la primera comparación, en el nivel $n-1$ se dispone del elemento menor de cada par de elementos de las hojas. Se comparan los elementos del nivel $n-1$ para conformar con los ganadores el nivel $n-2$, y así sucesivamente hasta llegar a la raíz del árbol que dispone en cada pasada del menor elemento.

El ganador, si la estructura de datos lo permite, es eliminado, y si no lo permite es sustituido por un elemento con el mayor valor posible para que no interfiera en el resto de comparaciones.

El número total de comparaciones que realiza el algoritmo es de $(n-1)\log_2 n$, lo cual es aproximadamente proporcional a $n\log_2 n$. No se practican intercambios en el algoritmo, pero es necesario recorrer la estructura de datos para marcar (o eliminar si es posible) todos los nodos que han contenido al elemento ganador. Su mayor inconveniente es la necesidad de una gran cantidad de memoria para almacenar no sólo los n elementos a ordenar, sino toda la estructura de datos necesaria para realizar la tarea encomendada.

Bibliografía

[1] D.E. Knuth. *The art of computer programming, vol. 3: Sorting and searching*. Addison-Wesley, 1973.

[2] E.W.Dijsktra. *A Discipline of Programming*. Prentice Hall, 1976.

[3] Peter Amey. Logic versus magic in critical systems. *6th Ada-Europe International Conference, Leuven, Belgium, may 14-18, 2001*, 2001.

[4] H. Lorin. *Sorting and sort systems*. Addison-Wesley, 1975.

[5] Alan Burns and Andy Wellings. *Sistemas de tiempo real y lenguajes de programación, 3rd Edición*. Addison Wesley, 2003.

[6] Baudet G. and Stevenson D. Optimal sorting algorithms for parallel computers. page 0, 1975.

[7] Dina Bitton, David J. Dewitt, David K. Hsiao, and Jaishankar Menon. A taxonomy of parallel sorting. *ACM Computing Surveys (CSUR)*.

[8] Barbara Liskov. *Abstrtaction and specification in program development*. MIT Press, 1987.

[9] Bertrans Meyer. *Construcción de software orientado a objetos, 2nd Edition*. Prentice Hall, 2002.

[10] Peter Amey. Closing the loop - the influence of code analysis on design. *7th Ada-Europe International Conference, Vienna, Austria, June 2002*, 2002.

[11] Klaas Sikkel. *Parsing schemata: A framework for Specification and Analysis of Parsing Algorithms*. Springer, 1997.

[12] Klaas Sikkel. Parsing schemata and correctness of parsing algorithms. URL http://wwwhome.cs.utwente.nl/~sikkel/papers/pdf/amilp95.pdf.

[13] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson International Edition, 2006.

[14] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math*, 13: 354–356, 1969.

[15] Huberto Comon, Max Dauchet, RMI Gilleron, Florent Jazquemard, Denis Lugiez, Sophie Tison, and Marc Tomasi. *Tree automata techniques and applications*. 2007. release October, 12th 2007.

[16] C.A.R. Hoare. *Essays in computer science*, chapter Chapter 2: Quicksort, pages 19–30. Prentice Hall, 1989.

[17] C.A.R. Hoare. *Essays in computer science*, chapter Chapter 5: Find, pages 59–74. Prentice Hall, 1989.

[18] Luis Joyanes Aguilar, Luis Rodríguez Buena, and Matilde Fernández Azuela. *Fundamentos de programación*. McGraw-Hill, 2008.

[19] Luis Joyanes Aguilar and Ignacio Zahonero. *Estructura de datos, algoritmos, abstracción y objetos, 3ra Edición*. McGraw-Hill, 1999.

[20] Luis Joyanes Aguilar. *Algoritmos y estructuras de datos una perspectiva en C*. McGraw-Hill, 2004.

[21] Christian Lenganer, Charles Consel, and Martin Odersky. Domain specific program generation. *International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, 2003.

[22] John Barnes. *High integrity software. The Spark approach to safety and security*. Addison-Wesley, 2003.

[23] D. Hilbert and S. Cohn-Vossen. *Geometry and the imagination*. Chelsea Publishing House, 1952.

[24] David A. Patterson and John L. Hennessy. *Organización y diseño de computadores. La interfaz hardware/software, 2nd Edición*. Editorial Reverte, 2011.

[25] James Rumbaugh, Michael Bluha, William Premerlani, Frederick Eddy, and William Lorensen. *Modelo y diseño orientado a objetos*. Prentice-Hall, 1991.

[26] A.J. Cole. *Macroprocesors*. Cambridge University Press, 2nd Edition, 1981.

[27] Alekseyev V.E. On certain algorithms for sorting with minimun memory. 1969.

[28] D.E. Knuth. *The art of computer Programming, Volume 1: Fundamental Algorithms, 3rd Edition.* Addison-Wesley, 1997.

[29] D.E. Knuth. *Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1997.

[30] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design, 2nd Edition.* Springer, 2012.

[31] Kenneth C. Louden. *Construcción de compiladores. Principios y práctica.* Thomson, 2005.

[32] M.H. Alsuaiyel. *Design techniques and Analysis.* World Scientific, 1999.

[33] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, translation and compiling.* Prentice Hall, 1973.

[34] G. Brassard and P. Bratley. *Fundamentos de algoritmia.* Prentice Hall, First Edition, 1997.

[35] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers.- Principles, Techniques & Tools, 2nd Edition.* Addison Wesley, 2007.

[36] Anany Levitin. *Introduction to the design and Analysis of algorithms, 2nd Edition.* Pearson International, 2007.

[37] R.C.T. Lee, S.S. Tseng, R.C. Chang, and Y.T. Tsai. *Introducción al diseño y análisis de algoritmos. Un enfoque estratégico.* McGraw-Hill, 2007.

[38] George Coulouris, Jean Dollimore, and Tim Kindberg. *Sistemas distribuídos. Conceptos y diseño.* Addison Wesley, 2007.

[39] John L. Hennessy and David A. Patterson. *Computer Architectura. A quantitative approach, 4th Edición.* McGraw-Hill, 2006.

[40] Udi Hanber. *Introduction to algorithms. A creative approach.* Addison Wesley, 1989.

Índice alfabético

Abstracción, 27, 28
    Meta-algoritmo, 31
    Software, 29
Algoritmo
    Definición, 5
Algoritmos clásicos
    Binario, 219
    Burbuja, 143, 191
    Características generales, 6
    Clasificación, 7
    Eficiencia, 155
    Heap, 209
    Inserción, 195
    Investigación, 32
    Merge, 217
    QuickSort, 203
    Selección, 189
    Shell, 199
    Torneo, 225
Asimetrías, 138
    Forma de trabajo, 138
    Inserción Elementos, 138

Burbuja
    Análisis gráfico, 145
    Comparaciones, 154
    Intercambios, 154
    Invertida, 153
    Paralelismo, 152
    Prioridades, 153
    Variaciones, 155
    Ventajas, 154
    Versión clásica, 144
    Versión mejorada, 144
    Versión optimizada, 147

Cadena
    Inserción elemento, 167
Cita
    C.A.R. Hoare, 5, 99
    Edsger W. Dijkstra, 135
    Hilbert, 117
    Nikolay Lobachevsky, 71
    Zen, 25
Concordancias, 140
    Análisis, 140
    Conclusiones, 143
    Situaciones, 141
    Soluciones, 142

Dualidad
    Meta-algoritmos, 135
    Problema, 137
    Proceso, 135
    Quicksort y binario, 96, 137
    Torneo vs Fusión, 114

Eficiencia
    Algoritmos clásicos, 155
    Binario, 159
    Meta-algoritmo ascendente, 161
    Meta-algoritmo descendente, 159
    Paralelismo, 157
    Quicksort, 158
    Torneo y Fusión, 160
Ejemplo
    Concordancias, 140

Elementos
  Primero con rapidez, 161
  Productor lento, 162
Entropía, 174
Esquemas
  Acciones, 134
  Definición, 117
  Idea, 118
  Jerarquía, 134
Esquemas de árboles, 127
  Burbuja, 152
  Conclusiones, 132
Esquemas de cadena única, 133
Esquemas de cadenas, 133
  Burbuja, 152
Esquemas generales, 120
  Burbuja, 125

Grafo, 51
  Ejemplo, 51

Inserción
  Binaria, 169
    Cabeza, 181
    Central, 181
  Binaria vs secuencial, 169
  Cabeza, 174
  Cabeza vs central, 173
  Central, 176
  Elemento, 167
  Elemento en cadena, 167
  Secuencial, 170
Intercambiar, 190

Matriz de incidencia, 54
  Ejemplo, 60
  Propiedades, 63
Meta-algoritmo
  Ascendente, 99
  Consideraciones, 47
  Definición, 25
  Descendente, 71
  Fundamentos, 37
Meta-algoritmo ascendente
  Derivaciones, 112
  Fusión, 113
  Primera fase, 101
  Representación, 101
  Segunda fase, 103
  Torneo, 114
  Versión formal, 110
  Versión indeterminaciones, 111
  Versión informal, 109
  Versión intuitiva, 100
Meta-algoritmo descendente
  Binario, 95
  Burbuja, 151
  Corrección, 86
  Definiciones básicas, 73
  Inicios, 72
  Primera fase, 76
  Quicksort, 89
  Representación, 76
  Segunda fase, 79
  Versión formal, 80
  Versión informal, 85
  Versión intuitiva, 74

Optimalidad
  Merge, 179
Optimización
  Binario, 96
  Fusión, 115
  Meta-algoritmo ascendente, 111
  Quicksort, 97
  Torneo, 115
Ordenación
  Ángulos, 34
  Abstracción, 1
  Análisis, 34
  Binario, 219
  Convenciones, 39
  En bloque, 19
  Estrategias, 47

Externa, 9
Interna, 8
Introducción, 1
Invertir, 45
Mejoras, 35
Optimalidad, 43
Pregunta, 46
Repeticiones, 41
Serie y paralelo, 10
Ordenación paralelo, 10
Burbuja, 15
En serie, 14
Fusión, 17
Hardware, 22
Ordenación en bloque, 19
Software, 24

Para-algoritmo, 38
Procedimiento
Ascendente, 137
Descendente, 136

Representaciones, 49
Grafo, 51
Matriz de incidencia, 54

Strassen, 44

Tesis
Conclusiones, 163
Desarrollos futuros, 164
Tipos de datos
Definición, 30
Ventajas, 30
Transitividad, 65
Binario, 66
Burbuja, 66