

# Análisis de Algoritmos.

Segundo Curso, Grado en Ingeniería Informática  
Escuela Politécnica Superior  
Universidad Autónoma de Madrid

---

Carlos Aguirre  
José Dorronsoro  
Pablo Varona



# Capítulo 1

## Introducción al análisis de algoritmos

### 1.1. Medida de eficacia de algoritmos

¿Qué analizar en un algoritmo? Tenemos diferentes opciones.

- Corrección
- Uso de memoria
- Rendimiento en ejecución: cuánto va a tardar

En este curso nos vamos a centrar en la tercera opción, es decir, en estudiar el tiempo de ejecución de un algoritmo. Para llevar a cabo esta tarea podemos considerar diferentes opciones:

**Opción 1.** Medir la duración de la ejecución con un reloj según los siguientes pasos:

(1) escoger un pseudocódigo  $\Rightarrow$  (2) programa  $\Rightarrow$  (3) ejecutable  $\Rightarrow$  (3) seleccionar entradas y  $\Rightarrow$  (4) ejecutar sobre las entradas midiendo tiempos con un reloj

Esta aproximación se demuestra poco práctica ya que tiene muchas dependencias

- Depende de la pericia del programador.
- Depende de la calidad del compilador.
- Depende del tipo de máquina y, sobre todo, de los datos sobre los que se ejecute.

Principal pega: el tiempo empleado sobre unas entradas no dice nada sobre lo que se tardará sobre otras.

**Opción 2.** Análisis de tiempos abstractos de ejecución sobre pseudocódigo. **Es la que usaremos en el curso.**

## 1.2. Tiempo abstracto de ejecución

Para medir los tiempos abstractos de ejecución de un algoritmo necesitamos definir la unidad de tiempo abstracto. La unidad de tiempo que vamos a considerar será la “sentencia elemental”, esto es, una línea simple de pseudocódigo que no contenga ninguna llamada a otra función. (Por supuesto, tendremos que tener cuidado con estas sentencias “no elementales”.)

Definiremos **tae** o **tiempo abstracto de ejecución** como el número de unidades de tiempo abstracto ejecutadas, es decir el número de sentencias elementales ejecutadas para unas entradas dadas.

Más formalmente, para un algoritmo  $A$  y una entrada  $I$  definimos  $t_A(I)$  como el número de sentencias elementales de  $A$  que se ejecutan sobre la entrada  $I$ .

La base del análisis está por tanto en el pseudocódigo.

### Ejemplo 1. Ordenación por selección

```
SelectSort(tabla T, ind P, ind U)
```

```
(1) i=P;
(2) mientras i < U :
    | m= min(T,i,U):      // NO es sentencia elemental; hay que hacer algo
    (2) | swap(T[i],T[m]); // tampoco lo es ... pero la tomamos como tal!
    | i++;
```

```
ind min(tabla T, ind P, ind U)
```

```
(1) m=P;
(2) para i de P+1 a U:    // la tomamos como sentencia simple
    | si T[i]<T[m]:
    (2) | m=i;
(1) devolver m;
```

Las instrucciones marcadas con (1) son instrucciones que se ejecutan una sola vez, las instrucciones marcadas con (2) se ejecutan repetidas veces. Un ejemplo de funcionamiento del algoritmo sobre la tabla

”4 3 2 1”

Seguimos la evolución de los índices  $i$  y  $m$ :

$i=1$   $m=4$  swap( $T[1],T[4]$ ) 1 3 2 4

i=2 m=3 swap(T[2],T[3]) 1 2 3 4  
i=3 m=3 swap(T[3],T[3]) 1 2 3 4

Vamos a calcular primero  $t_{min}(T, P, U)$  tiempo abstracto de ejecución de la función  $min$ , es decir, cuantas ejecuciones de sentencias elementales hace  $min$  para la tabla  $T$  entre  $P$  y  $U$ . Hay que observar que la condición  $T[m] > T[i]$  no siempre es cierta y por tanto la asignación  $m=i$  no siempre se ejecuta. Para superar esto, nos ponemos en el caso de “más trabajo”, suponiendo que siempre se cumple la condición y acotamos el número real de ejecuciones mediante un  $\geq$ . Por lo tanto, esto nos lleva a

$$t_{min}(T, P, U) \leq 2 + 3 \times \text{Número de iteraciones del bucle.}$$

donde contamos 3 sentencias elementales a lo sumo dentro del bucle de  $min$ . Claramente el bucle se ejecuta  $U - P$  veces, lo que nos lleva a

$$t_{min}(T, P, U) \leq 2 + 3(U - P).$$

Por lo tanto, tomando el caso particular  $P = 1, U = N$  nos da

$$t_{min}(T, P, U) \leq 2 + 3(N - 1).$$

Con esto ya podemos empezar a estimar  $t_{ss}(T, P, U)$ . Como mezcla sentencias básicas con la que contiene la llamada a  $min$ , que ejecuta un número variable de sentencias básicas en cada iteración, llegamos a:

$$\begin{aligned} t_{ss}(T, P, U) &= 1 + \sum_{i=P}^{U-1} (4 + t_{min}(T, i, U)) + 1 \\ &\leq 2 + 4(U - P) + \sum_{i=P}^{U-1} (2 + 3(U - i)) \\ &= 2 + 4(N - 1) + 2(N - 1) + 3 \sum_{i=P}^{U-1} (U - i) \\ &= 6N - 4 + 3 \sum_{j=1}^{N-1} j = 6N - 4 + 3 \frac{N(N - 1)}{2} \\ &= \frac{3}{2}N^2 + \frac{9}{2}N - 4. \end{aligned}$$

## Ejemplo 2. Búsqueda binaria

```

ind BBin(tabla T, ind P, ind U, clave k)
(2)      mientras P <= U :
          | m= (P+U)/2;
(2)      | si T[M]==k :
          |   devolver M;
          | else si k < T[M] :
          |   U=M-1;
          | else :
          |   P=M+1
(1)      devover err;

```

El número de sentencias elementales que se ejecutan en cada iteración del bucle es mayor o igual que cinco: una debida a la comprobación de la condición del bucle más, al menos cuatro, instrucciones dentro del bucle (como antes, no todas se ejecutan en cada iteración, por lo que de nuevo acotamos con un  $\geq$ ). Con esto se tiene

$$t_{BBin}(T, P, U, k) \leq 2 + 5 * n. \text{ de iteraciones.}$$

Por lo tanto es necesario estimar el número de iteraciones. Para ello estimamos el tamaño de la subtabla después de cada iteración.

Tras cada iteración:

iteración	1	2	3	.....
tamaño de la subtabla	N/2	N/4	N/8	.....

Supongamos que el número de elementos  $N$  de la tabla  $T$  es  $2^{l-1} < N \leq 2^l$  para algún  $l \in \mathbb{N}$ . Por lo tanto,  $l - 1 < \log_2 N \leq l$ , esto es,

$$l = \lceil \log_2(N) \rceil$$

donde el operador  $\lceil \cdot \rceil$  es el operador techo (a modo de ejemplo,  $\lceil 2,7 \rceil = 3$ ) Entonces, si el algoritmo llega a la iteración  $l$ , tras la iteración se seguiría buscando en una tabla de tamaño  $< N/2^l \leq 1$ , lo que no es posible. Por lo tanto, no hay más de  $\lceil \log_2(N) \rceil$  iteraciones y se tiene que

$$T_{BBin}(T, P, U, k) \leq 2 + 5 \lceil \log_2 N \rceil$$

**Observación** El trabajo máximo se produce agotando todas las iteraciones ( $P > U$ ), lo cual ocurre cuando la clave a buscar  $k$  no está en  $T$ .

Los ejemplos anteriores nos permiten establecer las siguientes observaciones:

1. Parece que para cualquier algoritmo  $A$ , a sus entradas  $I$  se les puede asignar un **tamaño de entrada**  $\tau(I)$  y se puede encontrar una cierta función  $f_A$  tal que

$$tae_A(I) \leq f_A(\tau(I)) \quad (1.1)$$

Hemos visto:

A	I	$\tau$	$f_A$
SelectSort	(T,P,U)	U-P+1	$\frac{3}{2}N^2 + \dots$
BBin	(T,P,U,k)	U-P+1	$5\lceil \log_2 N \rceil + 2$

2. El  $tae$  nos permite **generalizar los tiempos abstractos de ejecución** para nuevas entradas y además estimar tiempos reales de ejecución, como podemos ver en el siguiente ejemplo.

Consideremos **SelectSort** con una entrada  $I$  tal que  $\tau(I) = N$  y otra entrada  $I'$  tal que  $\tau(I') = 2N$ . Se tiene entonces que

$$tae_{SSort}(I') = \frac{3}{2}(2N)^2 + \dots = 4\left(\frac{3}{2}N^2 + \dots\right) \approx 4tae_{SSort}(I).$$

En general si  $I'$  tiene un tamaño  $k$  veces mayor que el tamaño de  $I$  (i.e  $\tau(I') = kN$ )

$$tae_{SSort}(I') = k^2 tae_{SSort}(I) \quad (1.2)$$

Supongamos ahora que un programa que implemente *SelectSort* tarda 1 segundo al ordenar una tabla de  $N = 1000$  elementos. Entonces

Tamaño	1000	2000	10000	100000
Tiempo real de ejecución	1 seg	4 seg	100 seg	10000 seg

3. En  $tae_{SSort}(N) = \frac{3}{2}N^2 + 6N - 6$  el término que más importa para valores grandes de  $N$  es el correspondiente a  $N^2$ . En el caso de  $tae_{BBin}(N) = 2 + 5\lceil \log_2 N \rceil$  el término de mayor peso es  $\lceil \log_2 N \rceil$ .
4. El  $tae$  tal como lo hemos definido sigue siendo algo artificioso ya que su valor puede depender de cómo se escriba el pseudocódigo.

Por ejemplo los códigos básicamente iguales de abajo dan lugar a  $taes$  distintos.

Código 1	Código 2
D=A+B+C;	E=A+B;
	F=C+E;
	D=F;
tae 1	tae 3

Por tanto, hay que hacer algo a este respecto.

5. En *SelectSort* el término  $N^2$  y en *BBin* el término  $\lceil \log_2 N \rceil$  vienen de ejecutar el bucle mas interno. Por esta razón vamos a cambiar la unidad de medida que usamos a la hora de calcular el *tae*. En lugar de contar el número de sentencias básicas que se ejecutan lo que vamos a considerar es el número de veces que se ejecuta una cierta **operación básica**.

### 1.3. Operación básica de un algoritmo

La **operación básica** (OB) de un algoritmo  $A$  es una sentencia elemental de  $A$  que además cumple las siguientes condiciones:

1. Debe aparecer en el bucle más interno del algoritmo. Debe ser la instrucción que más veces se ejecuta.
2. Debe ser una operación representativa del algoritmo.

**Ejemplo 1.** En el caso de *SelectSort* la instrucción de comparación de claves (cdc)

```
si T[i]<T[m]
```

cumple las condiciones 1 y 2 al estar en el bucle mas interno y tratarse de una comparación de claves. La instrucción

```
m=P;
```

no siempre se ejecuta, y la instrucción

```
i++;
```

es una instrucción general de muchos algoritmos.

**Ejemplo 2.** En el caso de *BBin* también tomaremos la cdc como operación básica la instrucción



```

si T[M]==K
...
else si K < T[M]
...
else si

```

A partir de ahora usaremos en vez de  $t_{abstractoj}$  la nueva medida  $n_A(I)$ , que es el número de veces que el algoritmo  $A$  ejecuta su operación básica (OB) sobre la entrada  $I$ . Veamos tres ejemplos, dos ya conocidos y uno nuevo.

### Ejemplo 1: Búsqueda binaria

Para calcular  $n_{BBin}(T, P, U, k)$  analizamos los bucles del algoritmo vemos que la operación básica siempre se ejecuta una vez para cada iteración del bucle. Por tanto se tiene:

$$n_{BBin} \leq 1 * \text{numero de ejecuciones del bucle} = \lceil \log_2 N \rceil \quad (1.3)$$

### Ejemplo 2: Ordenación por selección

Calculamos  $n_{SSort}(T, P, U)$  usando como OB la cdc. Sólo hay cdc en  $min$  y dentro de la función  $min$  el número de veces que se ejecuta la operación básica depende de los valores de entrada. El coste total del algoritmo será, tomando  $P = 1$  y  $U = N$ .

$$n_{SSort}(T, 1, N) = \sum_{i=1}^{N-1} n_{min}(T, i, N) = \sum_{i=1}^{N-1} N - i = \frac{N^2}{2} - \frac{N}{2} \quad (1.4)$$

### Ejemplo 3: Método de la burbuja

Para el método de la burbuja para ordenación veremos dos versiones.

#### Versión 1.

```

BurbujaSort_v1(Tabla T, ind P, ind U)
para i de U a P+1
  para j de P a i-1
    si T[j]>T[j+1] <---- Operación básica
      swap(T[j], T[j+1]);

```

La evolución del algoritmo sobre la tabla  $T = [4 \ 3 \ 2 \ 1]$  con  $P = 1$  y  $U = 4$  se puede ver en la tabla 1.3.

Como ya establecimos,  $n_{BSort}(T, P, U)$  será el número de comparaciones de clave (operación básica) que realice el algoritmo para ordenar la tabla. Por tanto:

$$n_{BSort}(T, P, U) = \sum_{i=P+1}^U n_{BSort}(T, P, i) \quad (1.5)$$

$i$	$j$	Operaciones	Tabla
4	1	$T[1] > T[2] ?$ Si $\text{swap}(T[1], T[2])$	3, 4, 2, 1 ↕
4	2	$T[2] > T[3] ?$ Si $\text{swap}(T[2], T[3])$	3, 2, 4, 1 ↕
4	3	$T[3] > T[4] ?$ Si $\text{swap}(T[3], T[4])$	3, 2, 1, 4 ↕
3	1	$T[1] > T[2] ?$ Si $\text{swap}(T[1], T[2])$	2, 3, 1, 4 ↕
3	2	$T[2] > T[3] ?$ Si $\text{swap}(T[2], T[3])$	2, 1, 3, 4 ↕
2	1	$T[1] > T[2] ?$ Si $\text{swap}(T[1], T[2])$	2, 1, 3, 4 ↕

Burbuja

donde  $n_{BSort}(T, P, i)$  es el número de veces que se ejecuta la operación básica para cada iteración (valor fijo de  $i$ ) del bucle externo. Tomando  $P = 1$  y  $U = N$  se tiene  $n_{BSort}(T, P, i) = i - 1$  y

$$n_{BSort}(T, 1, N) = \sum_{i=2}^N (i - 1) = \sum_{i=1}^{N-1} i = \frac{N^2}{2} - \frac{N}{2} \quad (1.6)$$

Podemos realizar las siguientes observaciones:

- Esta versión de  $BSort$  es bastante mala. Si ordenamos la tabla ya ordenada  $T = [1, 2, \dots, N]$  se tiene  $n_{BSort}(T, 1, N) = \frac{N^2}{2} - \frac{N}{2}$ , es decir, tardaremos lo mismo en ordenar una tabla ya ordenada que otra tabla desordenada.
- Una solución a lo anterior es cambiar el pseudocódigo añadiendo un detector de "swaps".

Dicho detector será un flag que se pondrá inicialmente al valor 0 para cada iteración de la variable  $i$  y que se cambiará al valor 1 si se produce algún swap. Si durante una iteración completa de la variable  $j$  para un valor fijo

de  $i$  no se produce ningún swap significa que la tabla ya está ordenada y por tanto no es necesario realizar mas comparaciones.

### Versión 2.

```
BurbujaSort_v2(Tabla T, ind P, ind U)
  Flag=1;
  i=U;
  mientras (Flag==1 && i>=P+1) :
    Flag=0;
    para j de P a i-1 :
      si T[j]>T[j+1] :
        swap(T[j],T[j+1]);
        Flag=1;
    i--;
```

Ahora se tiene que

$$n_{BSort}([1, 2, 3, \dots, N], 1, N) = N - 1,$$

lo cual es mucho mejor que el pseudocódigo anterior.

Sin embargo, el número de veces que se ejecuta la OB depende de la tabla que se ordene y, en general, sólo podemos decir que en la iteración  $i$  se cumple  $n_{BSort}(T, P, i) \leq i - 1$ , por lo que seguimos teniendo que

$$n_{BSort}([N, N - 1, \dots, 3, 2, 1], 1, N) \leq \frac{N^2}{2} - \frac{N}{2}.$$

## 1.4. Comparación de algoritmos

A la vista de lo anterior nos gustaría tener una manera de comparar algoritmos. Por supuesto tendremos que poner algunas condiciones para poder compararlos.

1. Que resuelvan el mismo problema (ordenación, búsqueda, etc).
2. Que lo resuelvan usando la misma operación básica.

Por tanto lo que vamos a considerar es  $F$ , familias de algoritmos que cumplan las condiciones 1 y 2. Por ejemplo

$F = \{\text{Algoritmos de ordenación por comparación de clave}\}.$

El método que seguiremos para comparar algoritmos será el siguiente:

1. Considerar una familia  $F$  de algoritmos que cumplan las condiciones 1 y 2.
2. Para todo algoritmo  $A \in F$  encontrar  $f_A$  t.q.

$$n_A(I) \leq f_A(\tau(I)) \quad (1.7)$$

3. Diremos que  $A_1$  es mejor que  $A_2$  si y solo si  $f_{A_1}$  “es menor” que  $f_{A_2}$ .

En general la comparación de rendimiento de algoritmos sólo es relevante para entradas grandes. Por ello compararemos las funciones  $f_{A_1}(N)$  y  $f_{A_2}(N)$  para valores grandes de  $N$  (es decir cuando  $N \rightarrow \infty$ ).

## 1.5. Comparación asintótica de funciones

En esta sección vamos a definir los principales operadores de comparación asintótica de funciones. En general, y salvo que se indique lo contrario, todas las funciones que vamos a considerar serán positivas.

### 1.5.1. $f = o(g)$

Dadas dos funciones positivas,  $f$  y  $g$  diremos que  $f = o(g)$  si

$$\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 0 \quad (1.8)$$

**Ejemplo.** Sean  $f(N) = N^k$  y  $g(N) = N^{k+\epsilon}$  con  $\epsilon > 0$ . Se tiene que  $f = o(g)$  ya que  $\lim_{N \rightarrow \infty} \frac{N^k}{N^{k+\epsilon}} = 0$

**Ejemplo.** Sean  $f(N) = \ln N$  y  $g = N^\epsilon$  con  $\epsilon > 0$ . Se tiene que  $f = o(g)$ , el cálculo del límite se puede realizar fácilmente aplicando la regla de L'Hopital.

Este operador correspondería (por supuesto de una forma muy genérica) al operador matemático “ $<$ ”. Es decir, en este caso consideramos que la función  $f$  es “claramente más pequeña” que la función  $g$ .

**1.5.2.**  $f = O(g)$ 

Decimos que  $f = O(g)$  si existen  $N_0$  y  $C$  tales que

$$f(N) \leq Cg(N) \quad \forall N \geq N_0. \quad (1.9)$$

**Ejemplo.** Sean  $f(N) = N^2$  y  $g(N) = N^2 + \sqrt{N}$ . Se tiene que

i  $f = O(g)$  ya que  $\forall N, f(N) \leq g(N)$

ii  $g = O(f)$  ya que  $\forall N, g(N) \leq 2f(N)$ ; es decir, si tomamos  $N_0 = 1$  y  $C = 2$ , se puede ver fácilmente que  $N^2 + \sqrt{N} \leq 2N^2$  porque  $\sqrt{N} \leq N^2$ .

**Observación**  $f = o(g) \Rightarrow f = O(g)$ , pues si  $\frac{f(N)}{g(N)} \rightarrow 0$  entonces, por la definición de límite debe haber un  $N_0$  tal que para todo  $N \geq N_0$  se tiene

$$\frac{f(N)}{g(N)} \leq 1 \Leftrightarrow f(N) \leq g(N).$$

Por tanto, basta tomar el mismo  $N_0$  que el del límite y  $C = 1$ .

El recíproco en general es falso, es decir  $f = O(g) \not\Rightarrow f = o(g)$ . Para ver lo anterior basta tomar  $f(N) = N^2$  y  $g(N) = N^2 + \sqrt{N}$ . Ya hemos visto que  $f = O(g)$ ; sin embargo se tiene  $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = \frac{f(N)}{g(N)} = \frac{N^2}{N^2 + \sqrt{N}} = 1 \neq 0$ .

Es importante además recalcar sobre la observación anterior que las **las afirmaciones positivas se demuestran y en las afirmaciones negativas se encuentra un contraejemplo**. Aplicaremos esta observación repetidas veces.

La notación  $O$  nos permite además eliminar algunas constantes y términos de menor peso en las comparaciones. En particular:

1. **Las constantes multiplicativas no importan** ya que si  $f = O(g) \Rightarrow f = O(kg)$  para cualquier valor de  $k$ . Por lo tanto siempre nos quedaremos con la expresión mas simple y significativa. Es decir, si hemos encontrado que  $f = O(3N^2)$  diremos que  $f = O(N^2)$  o por ejemplo si tenemos que  $f = O(e^{\pi^3} N^2)$ , de nuevo escribiremos simplemente  $f = O(N^2)$ .
2. Además si  $f = O(g)$  y  $h = o(g)$  entonces  $f = O(g + h)$ , es decir, **los términos menos significativos no importan**. Por tanto, en el caso de la suma de varias funciones nos quedaremos con el término mas significativo y no diremos  $f = O(N^2 + N)$  sino  $f = O(N^2)$ .

La notación asintótica  $f = O(g)$  corresponde (señalando siempre que de forma general y dentro del marco de las comparaciones asintóticas) a la comparación matemática  $f \leq g$ . Hay que señalar que puede ocurrir perfectamente que con la notación matemática se verifique que  $f(x) \geq g(x) \forall x$  y sin embargo se verifique  $f = O(g)$ .

**1.5.3.**  $f = \Omega(g)$ 

Decimos que  $f = \Omega(g)$  si y sólo si  $g = O(f)$ .

Esta notación correspondería a la comparación matemática  $f \geq g$ .

**1.5.4.**  $f = \Theta(g)$ 

Decimos que  $f = \Theta(g)$  si y solo si se verifica:  $f = O(g)$  y  $f = \Omega(g)$ .

También se puede aplicar cualquiera de las condiciones equivalentes, es decir  $f = \Theta(g)$  si  $f = O(g)$  y  $g = O(f)$  o si  $g = \Omega(f)$  y  $g = O(f)$  o si  $g = \Omega(f)$  y  $f = \Omega(g)$ . Esta notación corresponde al igual matemático ( $f = g$ ). De nuevo señalar que puede ocurrir perfectamente que  $f(x) \neq g(x) \forall x$  y sin embargo  $f = \Theta(g)$ .

Es fácil demostrar, aplicando la definición anterior, la siguiente propiedad:

$$f = \Theta(g) \Leftrightarrow g = \Theta(f) \quad (1.10)$$

La siguiente propiedad nos sera muy útil para la comparación  $\Theta$  de dos funciones:

Si  $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = L$ , con  $0 < L < \infty$ , entonces  $f = \Theta(g)$ .

La demostración es la siguiente:

$$\begin{aligned} \frac{f}{g} - L \rightarrow 0 &\Rightarrow \exists N_0 \text{ t.q. } \forall N \geq N_0, \left| \frac{f}{g} - L \right| \leq \frac{L}{2} \\ &\Rightarrow -\frac{L}{2} \leq \frac{f}{g} - L \leq \frac{L}{2} \Rightarrow \frac{L}{2} \leq \frac{f}{g} \leq \frac{3L}{2} \end{aligned}$$

con lo cual se tiene:

$$\begin{aligned} f(N) &\leq \frac{3L}{2}g(N) \Rightarrow f = O(g) \\ g(N) &\leq \frac{2}{L}f(N) \Rightarrow g = O(f) \end{aligned}$$

y por tanto  $f = \Theta(g)$ .

Con esta propiedad es fácil ver que  $N^2 + N = \Theta(N^2)$  ya que

$$\lim_{N \rightarrow \infty} \frac{N^2 + N}{N^2} = 1 \neq 0, \infty$$

1.5.5.  $f \sim g$ 

Diremos que  $f$  es asintóticamente equivalente a  $g$ , esto es,  $f \sim g$ , si

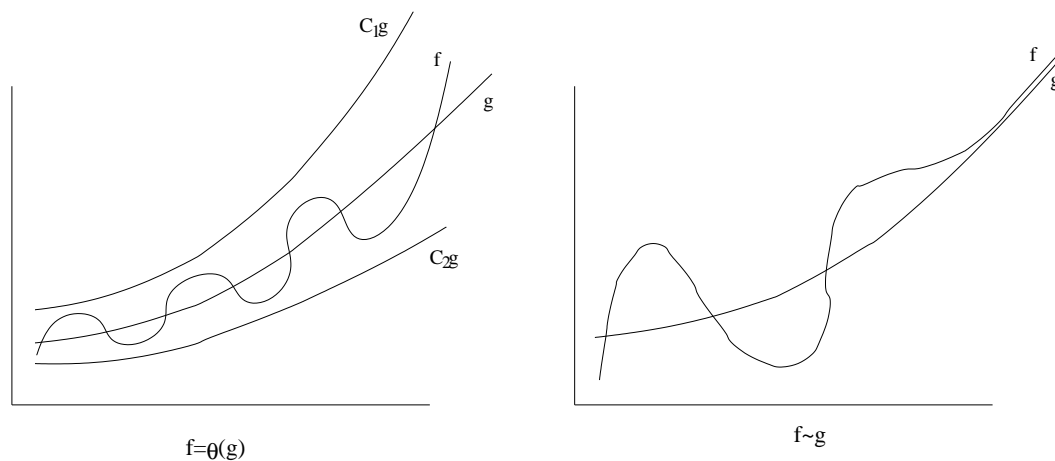
$$\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1. \quad (1.11)$$

Nótese que  $f \sim g$  es un caso particular de  $f = \Theta(g)$  en el que la constante del límite vale exactamente 1. Esta observación nos permite afirmar:

$$f \sim g \Rightarrow f = \Theta(g) \quad (1.12)$$

ya que si  $f \sim g$  entonces  $\lim_{N \rightarrow \infty} \frac{f(N)}{g(N)} = 1 = L$  con  $0 < L < \infty$ . Se puede considerar, por tanto, que la comparación  $\sim$  es mas precisa que la comparación  $\Theta$ .

En la figura inferior se puede ver la diferencia entre ambas notaciones.

1.5.6.  $f = g + O(h)$ 

Si  $h = o(g)$  diremos que  $f = g + O(h)$  si

$$|f - g| = O(h) \quad (1.13)$$

**Ejemplo.**

Sea  $f(N) = N^2 + \sqrt{N} + \ln N$  entonces si  $g(N) = N^2$  se tiene que:

- $f \sim g$ . Basta con dividir el numerador y el denominador por el término de mayor grado, en este caso  $N^2$ .
- $f = g + O(\sqrt{N})$ , ya que  $|f - g| = \sqrt{N} + \ln(N)$ , sin embargo el término de mayor peso en  $\sqrt{N} + \ln(N)$  es  $\sqrt{N}$ .

En general se verifica que

$$f = g + O(h) \Rightarrow f \sim h \quad (1.14)$$

La demostración es la siguiente: si  $|f - g| = O(h)$  existen  $C$  y  $N_0$  tales que

$$0 \leq \left| \frac{f(N)}{g(N)} - 1 \right| = \frac{|f(N) - g(N)|}{g(N)} \leq \frac{C \cdot h(N)}{g(N)}.$$

Como  $\frac{h}{g} \rightarrow 0$ , entonces  $\frac{C \cdot h}{g} \rightarrow 0$  y, por tanto

$$\left| \frac{f}{g} - 1 \right| \rightarrow 0 \Rightarrow \frac{f}{g} \rightarrow 1.$$

Se tiene, por tanto, la siguiente cadena de comparaciones de precisión decreciente:

$$f = g + O(h) \Rightarrow f \sim g \Rightarrow f = \Theta(g) \Rightarrow f = O(g) \quad (1.15)$$

En general, intentaremos ser lo más precisos posible. Por ejemplo, en el caso de los polinomios  $P(N) = a_k N^k + a_{k-1} N^{k-1} + \dots + a_0$ , con  $a_k \neq 0$  podríamos establecer las siguientes comparaciones:

- $P(N) = O(N^k)$
- $P(N) = \Theta(N^k)$ , ya que  $\lim_{N \rightarrow \infty} \frac{P(N)}{N^k} = a_k \neq 0$ .
- $P(N) \sim a_k N^k$ , ya que  $\lim_{N \rightarrow \infty} \frac{P(N)}{a_k N^k} = 1$ .
- $P(N) = a_k N^k + O(N^{k-1})$ , siendo ésta la expresión más precisa.

## 1.6. Crecimiento de funciones

En esta sección vamos a obtener algunas fórmulas para el crecimiento de funciones expresadas como sumas. El caso más simple es el de la suma de los términos de la progresión aritmética elemental

$$f(N) = S_N = \sum_{i=1}^N i = \frac{N(N+1)}{2} = \frac{N^2}{2} + O(N) \quad (1.16)$$

Una expresión más general de la fórmula anterior corresponde a la suma de los términos de una **progresión aritmética** general:



$$\begin{aligned}
f(N) = S_N &= \sum_{i=1}^N (ai + b) = a \sum_{i=1}^N i + bN \\
&= \frac{a}{2}N^2 + \frac{a}{2}N + bN \\
&= \frac{a}{2}N^2 + O(N)
\end{aligned}$$

Estudiamos la suma de los términos de una **progresión geométrica** de razón  $x \neq 1$ :

$$S_N = \sum_{i=1}^N x^i = \frac{x \cdot x^N - x}{x - 1} = \frac{x^{N+1} - x}{x - 1} \quad (1.17)$$

Nótese que la fórmula no tiene sentido si  $x = 1$ , pero entonces  $S_N = N$ . En la expresión anterior podemos distinguir dos casos:

- Si  $x > 1$ . Se tiene  $S_N = \Theta(x^N)$
- Si  $x < 1$ . Cambiando de signo el numerador y el denominador se tiene:

$$S_N = \frac{x - x^{N+1}}{1 - x} \xrightarrow{N \rightarrow \infty} \frac{x}{1 - x} \quad (1.18)$$

Una vez obtenida la fórmula anterior, se pueden estimar alguna sumas derivadas, como por ejemplo la suma  $S_N = \sum_{i=1}^N ix^i$ . Se pueden calcular mediante la siguiente expresión:

$$S_N = \sum_{i=1}^N ix^i = x \sum_{i=1}^N ix^{i-1} = x \frac{d}{dx} \left( \sum_{i=1}^N x^i \right) = x \frac{d}{dx} \left( \frac{x^{N+1} - x}{x - 1} \right) = \Theta(Nx^N) \quad (1.19)$$

Para sumas derivadas con potencias de  $i$  se aplica el siguiente método similar, considerando ahora derivadas segundas

$$S_N = \sum_{i=1}^N i^2 x^i = x^2 \sum_{i=1}^N i(i-1)x^{i-2} + \sum_{i=1}^N ix^i = x^2 \frac{d^2}{dx^2} (\dots) = \Theta(N^2 x^N)$$

En general se tiene:

$$S_N = \sum_{i=1}^N i^k x^i = \Theta(N^k x^N) \quad (1.20)$$

La siguiente fórmula se puede demostrar por inducción

$$S_N = \sum_{i=1}^N i^2 = \frac{N(N+1)(2N+1)}{6} = \frac{N^3}{3} + O(N^2) \quad (1.21)$$

Hay fórmulas similares para sumas de otras potencias. Sin embargo, si sólo estamos interesados en el crecimiento de estas sumas, se puede aplicar un método general.

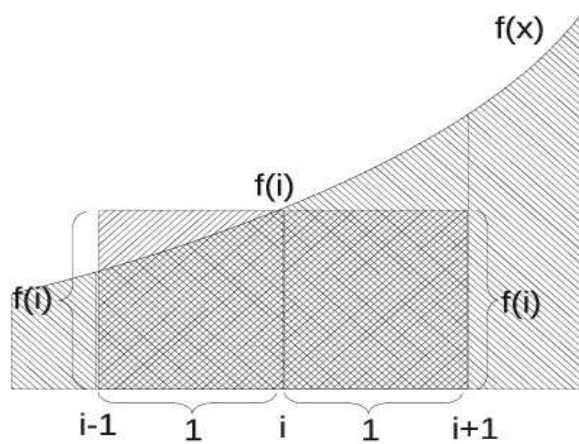
## 1.7. Estimación de sumas mediante integrales

El método de estimación de sumas por integrales nos permitirá aproximar el valor de ciertas sumas de las cuales no tengamos una expresión cerrada.

Sea  $f$  función creciente en el intervalo  $[i-1, i+1]$ , se tiene:

$$\int_{i-1}^i f(x)dx \leq f(i) \leq \int_i^{i+1} f(x)dx \quad (1.22)$$

En el siguiente gráfico se puede ver la anterior acotación del valor de  $f(i)$



Si  $f$  es **creciente** en el intervalo  $[0, N]$  podemos sumar la expresión (1.26) para los valores  $i = 1, 2, \dots, N$ , con lo que se obtiene

$$\sum_{i=1}^N \int_{i-1}^i f(x)dx \leq \sum_{i=1}^N f(i) \leq \sum_{i=1}^N \int_i^{i+1} f(x)dx$$

Finalmente, usando las propiedades de la suma de integrales obtenemos:

$$\int_0^N f(x)dx \leq S_N = \sum_{i=1}^N f(i) \leq \int_1^{N+1} f(x)dx \quad (1.23)$$

**Ejemplo 1:** Calcular  $S_N = \sum_{i=1}^N i^2$ . Aplicando la fórmula (1.27) a la función  $f(x) = x^2$  se tiene

$$\begin{aligned} \int_0^N x^2 dx &\leq S_N \leq \int_1^{N+1} x^2 dx \\ \Rightarrow \left[ \frac{X^3}{3} \right]_0^N &\leq S_N \leq \left[ \frac{X^3}{3} \right]_1^{N+1} \\ \Rightarrow \frac{N^3}{3} &\leq S_N \leq \frac{(N+1)^3}{3} - \frac{1}{3} \\ \Rightarrow S_N &\sim \frac{N^3}{3}. \end{aligned}$$

ya que dividiendo los tres términos de la desigualdad por  $\frac{N^3}{3}$  se tiene:

$$1 \leq \frac{S_N}{\frac{N^3}{3}} \leq \left( \frac{N+1}{N} \right)^3 - \frac{1}{N^3} \xrightarrow{N \rightarrow \infty} 1$$

y por tanto

$$\frac{S_N}{\frac{N^3}{3}} \xrightarrow{N \rightarrow \infty} 1 \Rightarrow S_N \sim \frac{N^3}{3}$$

También se puede demostrar que  $S_N = \frac{N^3}{3} + O(N^2)$ . Para ello, en lugar de dividir, restamos  $\frac{N^3}{3}$  en los tres términos de la desigualdad, con lo que se obtiene:

$$0 \leq S_N - \frac{N^3}{3} \leq \left( \frac{N+1}{N} \right)^3 - \frac{1}{N^3} - \frac{N^3}{3} = \frac{N^3}{3} + N^2 + \dots - \frac{N^3}{3} = N^2 + O(N)$$

y como  $0 \leq S_N - \frac{N^3}{3}$  se tiene que  $S_N - \frac{N^3}{3} = \left| S_N - \frac{N^3}{3} \right|$  y por tanto:

$$\left| S_N - \frac{N^3}{3} \right| \leq N^2 + O(N) \Rightarrow S_N = \frac{N^3}{3} + O(N^2)$$

En general, usando el método de aproximación de sumas por integrales se puede demostrar:

$$\sum_{i=1}^N i^k \sim \frac{N^{k+1}}{k+1} \quad (1.24)$$

$$\sum_{i=1}^N i^k = \frac{N^{k+1}}{k+1} + O(N^k) \quad (1.25)$$

**Ejemplo 2:** Vamos a estimar  $S_N = \sum_{i=1}^N \log i = \log N!$ .

En este caso  $\log x$  es de nuevo una función creciente y por tanto podemos aplicar de nuevo la fórmula (1.27), teniendo en cuenta que  $\lim_{x \rightarrow \infty} x \log x = 0$  se tiene:

$$\begin{aligned} \int_0^N \log x \, dx &\leq S_N \leq \int_1^{N+1} \log x \, dx \\ \Rightarrow [x \log x - x]_0^N &\leq S_N \leq [x \log x - x]_1^{N+1} \\ \Rightarrow N \log N - N &\leq S_N \leq (N+1) \log(N+1) - (N+1) + 1 \end{aligned}$$

dividiendo ahora los tres términos de la desigualdad por  $N \log N$  obtenemos:

$$1 - \frac{1}{\log N} \leq \frac{S_N}{N \log N} \leq \frac{(N+1) \log(N+1)}{N \log N} - \frac{N}{N \log N},$$

dado que (fácilmente por L'Hopital)

$$\lim_{N \rightarrow \infty} \frac{(N+1) \log(N+1)}{N \log N} - \frac{N}{N \log N} = \lim_{N \rightarrow \infty} \left( 1 - \frac{1}{\log N} \right) = 1,$$

y se llega a

$$\lim_{N \rightarrow \infty} \frac{S_N}{N \log N} = 1 \Rightarrow S_N \sim N \log N \quad (1.26)$$

Con algo más de trabajo se puede demostrar que  $S_N = N \log N + O(N)$ .

Dado que  $\log N! \sim N \log N$  podríamos preguntarnos si

$$e^{\log N!} = N! \sim e^{N \log N}.$$

La respuesta es no. De hecho, en general si

$$f = \Theta(g) \not\Rightarrow e^f = O(e^g)$$

Al ser lo anterior una negación vamos a dar un contraejemplo.

Sea  $f = 2N^2$  y  $g = N^2$ , por tanto  $e^f = e^{2N^2} = (e^{N^2})^2 = (e^g)^2$  y por tanto se tiene que  $e^g = o(e^f)$  y por tanto  $e^f \neq O(e^g)$ .

También se puede demostrar que  $f \sim g \not\Rightarrow e^f \sim e^g$ . Por ejemplo, sea  $f = N^2 + N$  y  $g = N^2$  sin embargo

$$\lim_{N \rightarrow \infty} \frac{e^{N^2+N}}{e^{N^2}} = \lim_{N \rightarrow \infty} \frac{e^{N^2} e^N}{e^{N^2}} = \lim_{N \rightarrow \infty} e^N = \infty \neq 1$$

Para la estimación de  $N!$  existe la **fórmula de Stirling**:

$$N! \sim \sqrt{2\pi N} \left(\frac{N}{e}\right)^N \quad (1.27)$$

**Ejemplo 3** vamos a estimar el crecimiento del  $N$ -simo número armónico, es decir  $H_N = \sum_{i=1}^N \frac{1}{i}$ .

La función  $f(x) = \frac{1}{x}$  es una función **decreciente**, por lo tanto la fórmula (1.27) no es válida. Afortunadamente usando un argumento similar se puede demostrar que si  $f(x)$  es una función **decreciente** en el intervalo  $[0, N+1]$  se verifica:

$$\int_0^N f(x)dx \geq S_N = \sum_{i=1}^N f(i) \geq \int_1^{N+1} f(x)dx \quad (1.28)$$

Por tanto, para estimar el crecimiento de  $H_N$  podemos escribir

$$\int_0^N \frac{1}{x} dx \geq H_N \geq \int_1^{N+1} \frac{1}{x} dx \Rightarrow \log x|_0^N \geq H_N \geq \log x|_1^{N+1}$$

El **problema** con la acotación anterior es que

$$\lim_{x \rightarrow 0} \log x = -\infty$$

y por tanto se tiene la inútil desigualdad  $\infty \geq H_N$  y  $H_N$  no queda acotado superiormente por una función de  $N$ . Si observamos la integral vemos que este valor infinito viene producido por el valor 0 del límite inferior de la primera integral. Una posible solución para evitar ese valor sería **quitar de alguna forma el primer término del sumatorio**. Por tanto convertimos  $H_N = \sum_{i=1}^N \frac{1}{i}$  en  $H_N = 1 + \sum_{i=2}^N \frac{1}{i}$ . Con lo que se obtiene:

$$\begin{aligned} \int_1^N \frac{1}{x} dx &\geq \sum_{i=2}^N \frac{1}{i} \geq \int_2^{N+1} \frac{1}{x} dx \\ \Rightarrow 1 + \int_1^N \frac{1}{x} dx &\geq 1 + \underbrace{\sum_{i=2}^N \frac{1}{x}}_{H_N} \geq \int_1^{N+1} \frac{1}{x} dx \end{aligned}$$

Con este cambio obtenemos:

$$1 + \log N \geq H_N \geq \log(N+1)$$

y dividiendo los tres términos por  $\log N$  y tomando límites se tiene que:

$$H_N \sim \log N \quad (1.29)$$

Se puede demostrar también que  $\lim_{N \rightarrow \infty} (H_N - \log N) = \gamma$  constante y por tanto

$$H_N = \log N + O(1).$$

El número  $\gamma$  es la **constante de Euler** y su valor es aproximadamente  $\gamma = 0,577\dots$  (En el año 2006 se alcanzó el record de cifras calculadas de la constante de Euler con un total de 116 millones de cifras decimales exactas).

## 1.8. Complejidad de Algoritmos

Hasta ahora hemos calculado el tiempo abstracto de ejecución de algunos algoritmos calculando el número de veces que se ejecutaba la operación básica del algoritmo. También hemos comparado algoritmos mediante la comparación asintótica de las funciones que acotaban el número de operaciones básicas como función del tamaño de la entrada  $n_A(I) \leq f(\tau(I))$ .

Sin embargo, en el esquema anterior hay un cierto desequilibrio, ya que el trabajo (número de operaciones básicas) que realiza un algoritmo depende de cada entrada en particular. Sin embargo las funciones  $f$  solamente dependen del tamaño de la entrada  $\tau(I)$ . Como ejemplo de este desequilibrio retomemos el ejemplo de la ordenación mediante burbuja BSort.

Si  $A = BSort$  se tiene

- $n_{BSort}(T, P, U) \leq \frac{N^2}{2} - \frac{N}{2}$  que es una cantidad elevada.
- $n_{BSort}([1, 2, \dots, N], P, U) = N - 1$  que es un valor mucho menor que el anterior.

En vista de lo anterior ¿qué función  $f$  deberíamos usar para comparar BSort con otros algoritmos? ¿ $N^2$  o  $N - 1$ ? Está claro que tenemos que controlar la dependencia del rendimiento de alguna manera independiente de las entradas individuales que se consideren.

Tenemos que precisar la función que vamos a comparar, para ello definimos el **Espacio de Entradas de un Algoritmo**.

Dado un Algoritmo  $A$  y un tamaño  $N$  definimos el espacio de entradas ( $E_A(N)$ ) de tamaño  $N$  para el algoritmo  $A$  como:

$$E_A(N) = \{I \text{ entradas de } A : \tau(I) = N\} \quad (1.30)$$

Vamos a calcular algunos espacios de entrada.

**Ejemplo 1:**  $E_{BSort}(N)$ . En principio podríamos establecer

$$E_{BSort}(N) = \{(T, P, U) : U - P + 1 = N\}$$

Es decir, cualquier tabla  $T$  (por grande que sea), más dos números enteros cualesquiera tales que  $U - P + 1 = N$ .

Este espacio de entradas es demasiado general (y demasiado grande) por lo que convendría precisar un espacio de entradas  $E_A(N)$  mucho mas manejable. Para ello, tenemos que observar que lo que importa en  $T$ , en general, es que la tabla esté mas o menos desordenada, no los valores iniciales y finales  $P$  y  $U$ , por tanto una primera acotación simple sería imponer  $P = 1$  y  $U = N$ .

También podemos observar que no son importantes los números que estén en la tabla  $T$ , sino el desorden relativo entre ellos. Es decir, por ejemplo, BSort realizará las mismas comparaciones al ordenar la tabla  $T = [3, 1, 2]$  que al ordenar la tabla  $T' = [25, 3, 12]$ . Con esta observación podemos establecer que  $1 \leq T[i] \leq N$ .

Con estas observaciones podemos establecer que

$$E_{BSort}(N) = \text{Permutaciones de } N \text{ Elementos} = \Sigma_N \quad (1.31)$$

Este espacio es mucho mas manejable que el que teníamos inicialmente. Su tamaño es  $|E_{BSort}(N)| = |\Sigma_N| = N!$ .

**Ejemplo 2:**  $E_{BBin}(N)$ .

En el caso de la búsqueda binaria podemos establecer como primera aproximación el siguiente espacio de entradas:

$$E_{BBin}(N) = \{(T, P, U, k) : k \text{ cualquiera, } U - P + 1 = N, T \text{ ordenada}\}$$

De nuevo estamos ante un espacio inmanejable por ser demasiado general. La primera acotación simple puede ser de nuevo  $P = 1, U = N$ . A la hora de buscar una clave  $k$  que esté en la tabla, da igual cuales sean los elementos  $T[i]$  de la tabla (por supuesto, han de estar ordenados); por tanto, podemos, al igual que en el caso de la burbuja, asumir que la tabla  $T$  verifica  $1 \leq T[i] \leq N$ . Sin embargo al estar la tabla  $T$  ordenada podemos tomar  $T = [1, 2, \dots, N]$ . También podemos observar que BBin realiza siempre el mismo trabajo para cualquier clave que no esté en la tabla por lo que a efectos de trabajo, basta considerar una clave genérica  $\hat{k} \neq [1, 2, 3, \dots, N]$ . Con estas observaciones podemos establecer el siguiente espacio de entradas para BBin:

$$E_{BBin}(N) = \{1, 2, 3, \dots, N, \hat{k}\} \quad (1.32)$$

Por tanto, el espacio de entradas esta formado por las  $N + 1$  posibles claves que podemos pasarle al algoritmo, es decir, los  $N$  elementos de la tabla  $T$ , que corresponden a las búsquedas con éxito y un elemento que no está en  $T$  y que por tanto corresponde a las búsquedas con fracaso. En este caso, por tanto, el tamaño del espacio de entradas es  $|E_{BBin}(N)| = N + 1$ .

## 1.9. Casos Peor, Mejor y Medio

Una vez definido el espacio de entradas de un algoritmo, vamos a definir lo que entendemos por casos peor, mejor y medio del algoritmo. Dado un algoritmo  $A$  con espacio de entradas de tamaño  $N$ ,  $E_A(N)$ , definimos:

- **Caso Peor**

$$W_A(N) = \max\{n_A(I) : I \in E_A(N)\} \quad (1.33)$$

- **Caso Mejor**

$$B_A(N) = \min\{n_A(I) : I \in E_A(N)\} \quad (1.34)$$

- **Caso Medio**

$$A_A(N) = \sum_{I \in E_A(N)} n_A(I)p(I) \quad (1.35)$$

donde  $p(I)$  es la probabilidad con la que aparece la entrada  $I$ .

Como  $B_A(N) \leq n_A(I) \leq W_A(N)$  para todo  $I \in E_A(N)$  se verifica que

$$B_A(N)p(I) \leq n_A(I)p(I) \leq W_A(N)p(I) \forall I \in E_A(N)$$

y por tanto

$$\sum_{I \in E_A(N)} B_A(N)p(I) \leq \sum_{I \in E_A(N)} n_A(I)p(I) \leq \sum_{I \in E_A(N)} W_A(N)p(I)$$

Como  $\sum_{I \in E_A(N)} p(I) = 1$  se tiene:

$$B_A(N) \leq A_A(N) \leq W_A(N) \quad (1.36)$$

Los casos mas simples (aunque no siempre) de calcular son  $W_A(N)$  y  $B_A(N)$  para los cuales daremos un método de general estimación. Los casos mas interesantes suelen ser  $W_A(N)$  y  $A_A(N)$ .



## 1.10. Cálculo de los casos peor y mejor

El cálculo del caso mejor de un algoritmo para un tamaño de entrada  $N$  se puede realizar en general en dos pasos:

- Paso 1: Encontrar  $f_A$  tal que si  $\tau(I) = N$

$$n_A(I) \leq f_A(N)$$

A veces, en lugar de pedir la desigualdad estricta pediremos  $n_A(I) = O(f_A(N))$ .

- Paso 2: Encontrar una entrada  $\tilde{I}$  tal que

$$n_A(\tilde{I}) \geq f_A(N)$$

A partir de estos dos pasos obtenemos las siguientes desigualdades:

- Por 1:  $W_A(N) \leq f_A(N)$  o bien  $W_A(N) = O(f_A(N))$
- Por 2:  $W_A(N) \geq f_A(N)$  o bien  $W_A(N) = \Omega(f_A(N))$ ,

y por tanto, juntando 1 y 2 obtenemos  $W_A(N) = f_A(N)$  o bien  $W_A(N) = \Theta(f_A(N))$

El caso mejor se calcula de una forma similar.

- Paso 1: Encontrar  $f_A$  tal que si  $\tau(I) = N$

$$n_A(I) \geq f_A(N)$$

- Paso 2: Encontrar una entrada  $\tilde{I}$  tal que

$$n_A(\tilde{I}) \leq f_A(N)$$

**Ejemplo 1:**  $W_{BSort}(N)$ .

Vamos a calcular  $W_{BSort}(N)$ , caso peor de la burbuja para tablas de  $N$  elementos. Para el paso 1 ya vimos que  $n_{BSort}(\sigma) \leq \frac{N^2}{2} - \frac{N}{2} \forall \sigma \in \Sigma_N$  y por tanto podemos decir

$$W_{BSort}(N) \leq \frac{N^2}{2} - \frac{N}{2} = \frac{N^2}{2} + O(N)$$

Para el paso 2 tomamos  $\tilde{\sigma} = [N, N-1, N-2, \dots, 2, 1]$ . Es fácil ver que  $n_{BSort}(\tilde{\sigma}) = \frac{N^2}{2} - \frac{N}{2}$  con lo que se tiene:

$$W_{BSort}(N) \geq \frac{N^2}{2} - \frac{N}{2}$$

Considerando ahora 1 y 2 obtenemos

$$W_{BSort}(N) = \frac{N^2}{2} - \frac{N}{2} = \frac{N^2}{2} + O(N)$$

Se puede demostrar (se deja como ejercicio) que  $B_{BSort}(N) = N - 1$ .

En general el caso medio es más complicado ya que el resultado depende de la distribución de probabilidad  $p(I)$ . Una posible simplificación es suponer que todas las entradas son equiprobables, en este caso se tiene:

$$p_A(I) = \frac{1}{|E_A(N)|} \quad (1.37)$$

En este caso, para el algoritmo BSort se tendría:

$$p(\sigma) = \frac{1}{N!} \forall \sigma \in E_{BSort}(N)$$

En el caso de la búsqueda binaria:

$$p(k) = \frac{1}{N+1} \forall k \in E_{BBin}(N)$$

En algunas ocasiones es relativamente fácil calcular  $A_A(N)$ , por ejemplo, en el caso de la búsqueda lineal.

**Ejemplo 2:**  $A_{BLin}(N)$ .

El pseudocódigo de la búsqueda lineal es el siguiente:

```
ind BLin(tabla T, ind P, ind U, clave k)
  para i de P a U
    si k==T[i]
      devolver i;
  devolver error;
```

La operación básica es la comparación de clave **si**  $k == T[i]$  dentro del bucle. Para un tamaño de entrada  $N$  podríamos proponer como espacio de entradas, tomando  $P = 1$ ,  $U = N$ :

$$E_{BLin} = \{\sigma \in \Sigma_N, 1, N, k \text{ cualquiera}\}$$

donde  $\Sigma_N$  es el conjunto de permutaciones de  $N$  elementos.

De nuevo este espacio es muy grande y podemos reducirlo. Si se trata de una búsqueda con éxito, el trabajo para buscar una clave que esté en la tabla

dependerá exclusivamente de la posición que ocupe la clave a buscar. Si la clave no está en la tabla, el número de cdc's es  $N$  pues se recorre la tabla completa. Por tanto, podemos de nuevo considerar como espacio de entradas:

$$E_{BLin}(N) = \{1, 2, 3, \dots, N, \hat{k}\} \quad (1.38)$$

con  $\hat{k}$  una clave genérica tal que  $\hat{k} \neq [1, 2, 3, \dots, N]$ . Además con este espacio tenemos dos posibles situaciones.

- La clave no está en la tabla. En este caso se tiene  $n_{BLin} = N$ .
- La clave está en la posición  $i$  de la tabla (es decir,  $T[i] == k$ ). En este caso se tiene  $n_{BLin}(i) = i$ .

Si asumimos equiprobabilidad de las entradas se tiene:

$$p(k == i, 1 \leq i \leq N) = p(k \neq [1, 2, 3, \dots, N]) = \frac{1}{N+1},$$

Con estas observaciones es fácil ver  $W_{BLin}(N) = N$  y  $B_{BLin}(N) = 1$ . Para calcular el caso medio recurrimos a la definición

$$\begin{aligned} A_{BLin}(N) &= \underbrace{\sum_{i=1}^N n_{BLin}(i) \frac{1}{N+1}}_{\text{búsqueda con éxito}} + \underbrace{N \frac{1}{N+1}}_{\text{búsqueda sin éxito}} \\ &= \frac{1}{N+1} \sum_{i=1}^N i + \frac{N}{N+1} \\ &= \frac{1}{N+1} \frac{N(N+1)}{2} + \underbrace{\frac{N}{N+1}}_{O(1)} = \frac{N}{2} + O(1) \end{aligned}$$

**Ejemplo 3:**  $A_{BLin}^e(N)$ .

Vamos a calcular el caso medio de la búsqueda lineal con éxito suponiendo ahora que:

$$p(k == i) = \frac{1}{C_N} \frac{\log i}{i}$$

donde  $C_N$  es una constante de normalización necesaria para que  $p(k == i)$  verifique la condición de distribución de probabilidad; esto es,

$$\sum_{i=1}^N p(k == i) = 1 \Rightarrow \sum_{i=1}^N \frac{1}{C_N} \frac{\log i}{i} = 1 \Rightarrow C_N = \sum_{i=1}^N \frac{\log i}{i}.$$

Como en este caso se trata de búsquedas con éxito, es decir, la clave  $k$  a buscar siempre está en la tabla, se tiene que  $p(k \neq [1, 2, 3, \dots, N]) = 0$ .

Aplicamos ahora la definición del caso medio:

$$\begin{aligned} A_N &= A_{BLin}^e(N) = \sum_{i=1}^N n_{BLin}(i)p(k == i) = \sum_{i=1}^N i \frac{1}{C_N} \frac{\log i}{i} = \\ &= \sum_{i=1}^N \frac{1}{C_N} \log i = \frac{1}{C_N} \sum_{i=1}^N \log i \\ &= \frac{S_N}{C_N}. \end{aligned}$$

donde:

$$S_N = \sum_{i=1}^N \log i$$

El valor de  $S_N$  ya lo calculamos previamente en (1.30), por lo tanto se tiene:

$$S_N \sim N \log N$$

Ahora vamos a estimar el valor de  $C_N$ ; para ello usaremos el método de aproximación por integrales.

La función  $f(x) = \frac{\log x}{x}$  es una función **decreciente** y por tanto tendremos que usar la fórmula (1.32).

$$\begin{aligned} \int_0^N \frac{\log x}{x} dx &\geq C_N \geq \int_1^{N+1} \frac{\log x}{x} dx \\ \Rightarrow \left[ \frac{1}{2} (\log x)^2 \right]_0^N &\geq C_N \geq \left[ \frac{1}{2} (\log x)^2 \right]_1^{N+1} \end{aligned}$$

Observamos, al igual que ocurrió en la estimación de la serie armónica, que la suma no queda bien acotada ya que  $\log 0 = -\infty$ . El problema de nuevo aparece por el límite inferior de la primera integral. La solución, al igual que en el caso de la serie armónica, pasa por sacar fuera del sumatorio el primer término:

$$C_N = \sum_{i=1}^N \frac{\log i}{i} = \underbrace{\frac{\log 1}{1}}_0 + \sum_{i=2}^N \frac{\log i}{i} = \sum_{i=2}^N \frac{\log i}{i}$$

con lo que se ahora se tiene:

$$\begin{aligned}
& \int_1^N \frac{\log x}{x} dx \geq C_N \geq \int_2^{N+1} \frac{\log x}{x} dx \\
\Rightarrow & \left[ \frac{1}{2}(\log x)^2 \right]_1^N \geq C_N \geq \left[ \frac{1}{2}(\log x)^2 \right]_2^{N+1} \\
\Rightarrow & \frac{1}{2}(\log N)^2 \geq C_N \geq \frac{1}{2}(\log N + 1)^2 - \frac{1}{2}(\log 2)^2
\end{aligned}$$

Dividiendo ahora los tres términos de la desigualdad por  $\frac{1}{2}(\log N)^2$  se tiene:

$$1 \geq \frac{C_N}{\frac{1}{2}(\log N)^2} \geq \frac{\frac{1}{2}(\log N + 1)^2}{\frac{1}{2}(\log N)^2} - \frac{\frac{1}{2}(\log 2)^2}{\frac{1}{2}(\log N)^2} \xrightarrow{N \rightarrow \infty} 1$$

Por lo tanto

$$\lim_{N \rightarrow \infty} \frac{C_N}{\frac{1}{2}(\log N)^2} = 1 \Rightarrow C_N \sim \frac{1}{2}(\log N)^2$$

Tenemos hasta ahora una igualdad  $A_N = \frac{S_N}{C_N}$  y dos igualdades asintóticas  $C_N \sim \frac{1}{2}(\log N)^2$  y  $S_N \sim N \log N$ .

Podemos suponer entonces que

$$A_N \sim \frac{N \log N}{\frac{1}{2}(\log N)^2} = \frac{2N}{\log N}$$

Para ver que lo anterior es cierto tenemos que comprobar que

$$\lim_{N \rightarrow \infty} \frac{A_N}{\left( \frac{2N}{\log N} \right)} = 1$$

Para ello reescribimos el límite anterior de la siguiente manera:

$$\lim_{N \rightarrow \infty} \frac{A_N}{\left( \frac{2N}{\log N} \right)} = \lim_{N \rightarrow \infty} \frac{\left( \frac{S_N}{C_N} \right)}{\left( \frac{N \log N}{\frac{1}{2}(\log N)^2} \right)} = \lim_{N \rightarrow \infty} \left( \frac{S_N}{N \log N} \right) \frac{1}{\left( \frac{C_N}{\frac{1}{2}(\log N)^2} \right)} = 1$$

y por tanto:

$$A_N \sim \frac{2N}{\log N}$$



# Capítulo 2

## Métodos de ordenación

### 2.1. Ordenación por inserción

La idea principales del funcionamiento método de ordenación de una tabla  $T$  con índices entre  $P$  y  $U$  por inserción es realizar de manera iterada entre  $i = P + 1$  y  $U$  lo siguiente:

1. Suponer al inicio de la iteración  $i$ -ésima que la subtabla  $T[P], T[P+1], \dots, T[i-1]$  contiene elementos ordenados entre si, es decir

$$T[P] < T[P + 1] < \dots < T[i - 1].$$

2. En la iteración  $i$ , el elemento  $T[i]$  se coloca en la posición correspondiente de la subtabla  $T[P], T[P + 1], \dots, T[i - 1]$ .

#### Ejemplo

$P = 1, U = 4, T = [1\ 4\ 3\ 2]$ . En la iteración  $i = 3$ , la tabla pasa a  $1\ 4\ 3\ 2 \longrightarrow 1\ 3\ 4\ 2$  y luego pasaríamos a insertar para  $i = 4$ .

El pseudocódigo del método de inserción es el siguiente:

```
InsertSort(Tabla T, ind P, ind U)
para i de P+1 a U:
  A=T[i]; // variable auxiliar para evitar swaps
  j=i-1;
  mientras (j >= P && T[j]>A):
    T[j+1]=T[j];
    j--;
  T[j+1]=A;
```

La operación básica del algoritmo es la comparación de clave  $T[j] > A$  situada dentro del bucle. Para el análisis suponemos  $P = 1$ ,  $U = N$  y  $T = \sigma$ . El trabajo del bucle interno `mientras (j >= P && T[j]>A)` es muy dependiente del estado de la tabla, con lo cual es conveniente escribir

$$n_{IS}(\sigma) = \sum_{i=2}^N n_{IS}(\sigma, i) \quad (2.1)$$

donde  $n_{IS}(\sigma, i)$  es el número de operaciones básicas que InsertSort realiza sobre una tabla  $\sigma$  en la iteración  $i$ -ésima.

Observando el pseudocódigo podemos acotar:

$$1 \leq n_{IS}(\sigma, i) \leq i - 1 \Rightarrow \sum_{i=2}^N 1 \leq \sum_{i=2}^N n_{IS}(\sigma, i) \leq \sum_{i=2}^N (i - 1) = \sum_{i=1}^{N-1} i$$

y por tanto

$$N - 1 \leq n_{IS}(\sigma) \leq \frac{N^2}{2} - \frac{N}{2} \quad (2.2)$$

Para la estimación de  $W_{IS}(N)$  acabamos de encontrar una función  $f(N) = \frac{N^2}{2} - \frac{N}{2}$  tal que

$$n_{IS}(\sigma) \leq \frac{N^2}{2} - \frac{N}{2} \forall \sigma \in E_{IS}(N)$$

y por tanto

$$W_{IS}(N) \leq \frac{N^2}{2} - \frac{N}{2}.$$

Por otro lado si consideramos

$$\tilde{\sigma} = [N, N - 1, N - 2, \dots, 3, 2, 1],$$

es fácil comprobar que  $n_{IS}(\tilde{\sigma}, i) = i - 1$  y por tanto

$$n_{IS}(\tilde{\sigma}) = \sum_{i=2}^N (i - 1) = \frac{N^2}{2} - \frac{N}{2}.$$

Por tanto se tiene (ver sección 1.10):

$$W_{IS}(N) = \frac{N^2}{2} - \frac{N}{2}. \quad (2.3)$$



El caso mejor  $B_{IS}(N)$  se estima de forma similar, tomando en este caso la permutación  $\tilde{\sigma} = [1, 2, 3, \dots, N-1, N]$ , con lo que se obtiene (se dejan los detalles como ejercicio):

$$B_{IS}(N) = N - 1. \quad (2.4)$$

Para la estimación del caso medio empleamos la definición, asumimos equiprobabilidad y usamos de nuevo la cantidad intermedia  $n_{IS}(\sigma, i)$ :

$$\begin{aligned} A_{IS}(N) &= \sum_{\sigma \in \Sigma_N} n_{IS}(\sigma) p(\sigma) = \frac{1}{N!} \sum_{\sigma \in \Sigma_N} \sum_{i=2}^N n_{IS}(\sigma, i) \\ &= \sum_{i=2}^N \frac{1}{N!} \sum_{\sigma \in \Sigma_N} n_{IS}(\sigma, i) = \sum_{i=2}^N A_{IS}(\sigma, i) \end{aligned} \quad (2.5)$$

donde  $A_{IS}(\sigma, i)$  es el trabajo medio que realiza el algoritmo InsertSort en la iteración  $i$ .

Para estimar  $A_{IS}(\sigma, i)$  sea  $\sigma(i)$  el elemento que está en la posición  $i$  al inicio de la  $i$ -ésima iteración. Por el funcionamiento del algoritmo InsertSort se tiene en ese momento que

$$\sigma(1) < \sigma(2) < \dots < \sigma(i-1).$$

En la tabla 2.1 se puede ver cuantas comparaciones de clave necesita el elemento  $\sigma(i)$  para ser insertado en su posición correcta en función de la posición final en la que quede.

Posición Final	CDC perdidas ( $\sigma(i) < \sigma(j)$ )	CDC ganadas ( $\sigma(i) > \sigma(j)$ )	Total CDC
i	0	1 ( $\sigma(i) > \sigma(i-1)$ )	1
i-1	1 ( $\sigma(i) < \sigma(i-1)$ )	1 ( $\sigma(i) > \sigma(i-2)$ )	2
i-2	2 ( $\sigma(i) < \sigma(i-1), \sigma(i) < \sigma(i-2)$ )	1 ( $\sigma(i) > \sigma(i-3)$ )	3
...			
3	i-2 ( $\sigma(i) < \sigma(i-1), \dots, \sigma(i) < \sigma(3)$ )	1 ( $\sigma(i) > \sigma(2)$ )	i-2
2	i-2 ( $\sigma(i) < \sigma(i-1), \dots, \sigma(i) < \sigma(2)$ )	1 ( $\sigma(i) > \sigma(1)$ )	i-1
1	i-1 ( $\sigma(i) < \sigma(i-1), \dots, \sigma(i) < \sigma(1)$ )	0	i-1

Cuadro 2.1: Posibles CDCs en la iteración  $i$ .

Por tanto podemos calcular

$$A_{IS}(\sigma, i) = 1p(1) + 2p(i-1) + \dots + (i-1)p(2) + (i-1)p(1)$$

donde  $p(j)$  es la probabilidad de que  $\sigma(i)$  termine en la posición  $j$  ( $j \leq i$ ) tras la iteración  $i$ . Asumiendo de nuevo equiprobabilidad se tiene que  $p(j) \simeq 1/i$  al haber un total de  $i$  posibles posiciones  $(1, 2, 3, \dots, i-1, i)$  donde puede situarse el elemento  $\sigma(i)$ . Con lo cual se tiene

$$A_{IS}(\sigma, i) \simeq \frac{1}{i} \sum_{j=1}^{i-1} i + \frac{i-1}{i} = \frac{i-1}{2} + \frac{i-1}{i}$$

Sustituyendo en (2.5) se tiene

$$A_{IS}(\sigma, i) = \sum_{i=2}^N \left( \frac{i-1}{2} + \frac{i-1}{i} \right) = \sum_{j=1}^{N-1} \frac{i}{2} + \underbrace{\sum_{j=1}^{N-1} \frac{j}{j+1}}_{O(1)} = \frac{N^2}{4} + O(N) \quad (2.6)$$

En el caso de InsertSort se tiene  $A_{IS}(N) \simeq \frac{1}{2}W_{IS}(N)$ ; por lo tanto InsertSort es un método bastante malo.

## 2.2. Métodos Shell\*

La idea de los métodos Shell consiste en realizar una ordenación iterada en subtablas de diferentes tamaños con el fin de trabajar al principio con tablas quizá poco ordenadas pero muy pequeñas y al final con tablas muy grandes pero ya muy ordenadas. La forma de hacerlo es iterar una variante de la Inserción con “incrementos” según el pseudocódigo de la rutina siguiente:

```

ISInc(tabla T, ind P, ind U, inc k)
  i=P+k;
  mientras i <= U:
    A=T[i];
    j=i-k;
    mientras j >= P && T[j] > A :
      T[j+k]=T[j];
      j-=k;
    T[k+k]=A;
    i+=k

```

La rutina ISInc no es mas que un InsertSort igual que el visto en la sección anterior con la única diferencia que la ordenación se realiza solo en los elementos separados exactamente por  $k$  posiciones.

En el pseudocódigo general, la tabla **Inc** es una tabla que contiene los diferentes incrementos a utilizar y el argumento **nInc** indica el número de incrementos que hay en **Inc**. El método de Shell itera **ISInc** con incrementos decrecientes, con  $\text{Inc}[\text{nInc}] = 1$ ; esto es, al final se aplica la Inserción estándar pero sobre una tabla ya bastante ordenada.

```
ShellS(tabla T, ind P, ind U, tabla Inc, int nInc)
  para i de 1 a nInc:
    para j de 1 a Inc[i]:
      IsInc(T, P+j-1,U,Inc[i]);
```

El análisis de ShellSort es complicado y depende esencialmente de los incrementos de **Inc**.

**Ejemplo** Si  $\text{Inc} = \{2^k, 2^{k-1}, \dots, 4, 2, 1\}$  se puede demostrar que el algoritmo tiene un coste cuadrático en el caso peor, lo cual es malo. La razón de este mal comportamiento es que vuelve a realizar muchas veces comparaciones que ya se han realizado.

Por ejemplo si consideramos la tabla  $T = [1, 2, 3, 4, 5, 6, 7, 8, 9]$ . Para  $\text{Inc}[1]=4$  se realizan 4 llamadas a **ISInc**

1	5	9	j=1
2	6		j=2
3	7		j=3
4	8		j=4

Para  $\text{Inc}[2]=2$  se realizan 2 llamadas a **ISInc**:

1	3	5	7	9	j=1
2	4	6	8		j=2

Vemos que los números en cada subtabla ya habían coincidido en varias ocasiones en subtablas anteriores.

**Ejemplo** Si  $\text{Inc} = \{2^k - 1, 2^{k-1} - 1, \dots, 7, 3, 1\}$  se puede demostrar que en este caso el rendimiento mejora a  $O(N^{\frac{3}{2}})$  lo cual es algo mejor que la inserción. Con otros incrementos se pueden obtener rendimientos mejores, por ejemplo  $O(N^{\frac{4}{3}})$ .

Una cuestión que surge a raíz de lo anterior es ¿hasta cuánto puedo mejorar ShellSort?, o incluso más generalmente, ¿hasta cuánto puedo mejorar los algoritmos de ordenación? Es decir, ¿existe alguna función  $f$  tal que para todo algoritmo de ordenación  $A$

$$h_A(I) = \Omega(f(N)) \tag{2.7}$$

con  $N = \tau(I)$ ? Un buen candidato podría ser  $f(N) = N$  ya que es razonable pensar que cualquier algoritmo de ordenación deberá comparar cada elemento de la tabla al menos una vez. ¿Existe algún algoritmo que alcance esa cota para sus casos peor y medio?

### 2.3. Cotas inferiores para algoritmos locales de ordenación

Hasta ahora hemos visto que el trabajo que realiza un algoritmo en una tabla depende del desorden que haya en la tabla. En las permutaciones de  $N$  elementos el desorden se mide en función del número de **inversiones** que haya en la permutación.

**Definición** Dada  $\sigma \in \Sigma_N$  decimos que dos índices  $i < j$  están en **inversión** si  $\sigma(i) > \sigma(j)$ .

**Ejemplo** Dado  $\sigma = (3, 2, 1, 5, 4)$  las inversiones son  $(1, 2), (1, 3), (2, 3), (4, 5)$ , donde  $(i, j)$  indica que los índices  $i, j$  están en inversión. En el caso de  $\sigma$  hay un total de 4 inversiones.

Denotamos como  $inv(\sigma)$  el número de inversiones que hay en  $\sigma$ .

Podemos calcular las inversiones de algunas permutaciones de  $N$  elementos:

$$inv([1, 2, 3, \dots, N]) = 0 \quad (2.8)$$

$$inv([N, N-1, \dots, 2, 1]) = (N-1) + (N-2) + \dots + 2 + 1 = \frac{N^2}{2} - \frac{N}{2} \quad (2.9)$$

Podemos observar que no puede haber otra permutación con mayor desorden que (2.9) ya que  $\forall \sigma \in \Sigma_N$

$$\begin{aligned} inv([\sigma(1), \sigma(2), \dots, \sigma(N-1), \sigma(N)]) &= \\ inv(\sigma(1)) + inv(\sigma(2)) + \dots + inv(\sigma(N-1)) + \underbrace{inv(\sigma(N))}_0 &\leq \\ (N-1) + (N-2) + \dots + 2 + 1 = \frac{N^2}{2} - \frac{N}{2} & \end{aligned}$$

por lo que  $inv(\sigma) \leq inv([N, N-1, \dots, 2, 1]) = \frac{N^2}{2} - \frac{N}{2}$ .

**Definición:** Un algoritmo de ordenación por comparación de clave diremos que es **local** si por cada comparación de clave se deshace **a lo sumo** una inversión.

InsertSort, BubbleSort son algoritmos locales; SelectSort técnicamente no lo es, pero “moralmente” sí.

Como consecuencia de la definición, si un algoritmo  $A$  es local, el número mínimo de comparaciones de clave que deberá realizar para ordenar una entrada  $\sigma$  serán, al menos, tantas como inversiones tenga  $\sigma$ . Es decir

$$n_A(\sigma) \geq \text{inv}(\sigma) \quad (2.10)$$

Como resultado se tiene nuestro primer ejemplo de cota inferior: si un algoritmo  $A$  es local se verifica:

$$W_A(N) \geq \frac{N^2}{2} - \frac{N}{2} \quad (2.11)$$

ya que

$$W_A(N) \geq n_A([N, N-1, \dots, 2, 1]) \geq \text{inv}([N, N-1, \dots, 2, 1]) = \frac{N^2}{2} - \frac{N}{2}.$$

Por tanto InsertSort y BubbleSort son óptimos para el caso peor entre los algoritmos locales.

Para estimar cotas inferiores para el caso medio de algoritmos locales necesitamos la siguiente definición.

Si  $\sigma \in \Sigma_N$  definimos la permutación traspuesta  $\sigma^t$  como

$$\sigma^t(i) = \sigma(N - i + 1) \quad (2.12)$$

Por ejemplo si  $\sigma = [3, 2, 1, 5, 4]$ , entonces  $\sigma^t = [4, 5, 1, 2, 3]$ .

La permutación traspuesta verifica

$$(\sigma^t)^t = \sigma \quad (2.13)$$

ya que

$$(\sigma^t)^t(i) = \sigma^t(\underbrace{N - i + 1}_j) = \sigma(N - j + 1) = \sigma(N - (N - i + 1) + 1) = \sigma(i).$$

El número de inversiones de una permutación y las de su traspuesta están relacionados. Veamos unos ejemplos.

Para  $N = 5$  elegimos

$$\sigma = (3 \ 2 \ 1 \ 4 \ 5) \longrightarrow \text{inv}(\sigma) = 3,$$

luego

$$\sigma^t = (5 \ 4 \ 1 \ 3 \ 2) \longrightarrow \text{inv}(\sigma^t) = 7.$$

Vemos que  $\text{inv}(\sigma) + \text{inv}(\sigma^t) = 10 = \frac{5^2}{2} - \frac{5}{2}$ .

Si ahora elegimos

$$\sigma = (1\ 2\ 3\ 4\ 5) \longrightarrow \text{inv}(\sigma) = 0,$$

mientras que

$$\sigma^t = (5\ 4\ 3\ 2\ 1) \longrightarrow \text{inv}(\sigma^t) = 10.$$

Vemos que también  $\text{inv}(\sigma) + \text{inv}(\sigma^t) = 10$ .

En general para toda  $\sigma \in \Sigma_N$  se tiene:

$$\text{inv}(\sigma) + \text{inv}(\sigma^t) = \frac{N^2}{2} - \frac{N}{2} \quad (2.14)$$

La demostración es la siguiente: dados dos índices  $i, j$  ( $i < j$ ) en  $\sigma$ , una y solo una de las dos afirmaciones siguientes es cierta:

- o  $(i, j)$  están en inversión en  $\sigma$ .
- o  $(N - j - 1, N - i - 1)$  están en inversión en  $\sigma^t$ .

Supongamos que  $i, j$  no están en inversión en  $\sigma$ , es decir  $\sigma(i) < \sigma(j)$ . Entonces

$$\sigma^t(N - j + 1) > \sigma^t(N - i + 1)$$

ya que

$$\sigma^t(N - j + 1) = \sigma(j) > \sigma(i) = \sigma^t(N - i + 1).$$

Sin embargo, como  $i < j$  se tiene que  $N - i + 1 > N - j + 1$ . Con lo cual  $(N - j - 1, N - i - 1)$  están en inversión en  $\sigma^t$ .

Se deja como ejercicio demostrar que si  $(i, j)$  están en inversión en  $\sigma$  entonces  $(N - j - 1, N - i - 1)$  no están en inversión en  $\sigma^t$ .

Se puede observar por tanto que

$$\text{inv}(\sigma) + \text{inv}(\sigma^t) = \text{núm de parejas}\{(i, j), 1 \leq i < j \leq N\} = \frac{N(N-1)}{2}$$

ya que escribiendo  $np$  en vez de número de parejas se tiene

$$\begin{aligned} np\{(i, j), 1 \leq i < j \leq N\} &= np\{(1, j), 1 < j \leq N\} + np\{(2, j), 2 < j \leq N\} + \dots + \\ &\quad np\{(N-1, j), N-1 < j \leq N\} \\ &= (N-1) + (N-2) + \dots + 2 + 1. \end{aligned}$$

Con las observaciones anteriores se puede demostrar que si  $A$  es algoritmo local

$$A_A(N) \geq \frac{N^2}{4} + O(N). \quad (2.15)$$

ya que si  $A$  es local se tiene:

$$A_A(N) = \frac{1}{N!} \sum_{\sigma \in \Sigma_N} n_A(\sigma) \geq \frac{1}{N!} \sum_{\sigma \in \Sigma_N} inv(\sigma)$$

Ahora sumamos de manera agrupada el número de inversiones en parejas de una permutación y su transpuesta, es decir

$$\begin{aligned} \frac{1}{N!} \sum_{\sigma \in \Sigma_N} inv(\sigma) &= \frac{1}{N!} \sum_{\sigma, \sigma^t \in \Sigma_N} \underbrace{inv(\sigma) + inv(\sigma^t)}_{\frac{N(N-1)}{2}} = \frac{1}{N!} \frac{N(N-1)}{2} \sum_{\sigma, \sigma^t \in \Sigma_N} 1 \\ &= \frac{1}{N!} \frac{N(N-1)}{2} \frac{N!}{2} = \frac{N^2}{4} - \frac{N}{4}, \end{aligned}$$

y por tanto  $A_A(N) \geq \frac{N^2}{4} + O(N)$ . Por tanto se tiene que el método de inserción es **óptimo** dentro de los algoritmos locales. Sin embargo sigue sin ser un buen método en general. Hemos visto que cualquier algoritmo local tiene un coste medio muy elevado y por tanto son algoritmos con un rendimiento muy bajo que hay que mejorar mediante otro tipo de algoritmos.

## 2.4. Métodos divide y vencerás

La idea de los métodos divide y vencerás se basa en aplicar de manera recursiva los siguientes tres pasos:

1. **Partir**  $T$  en  $T_1$  y  $T_2$ .
2. **Ordenar**  $T_1$  y  $T_2$  de forma recursiva.
3. **Combinar**  $T_1$  y  $T_2$  (ya ordenados) en una ordenación de  $T$ .

Un pseudocódigo general de un algoritmo divide y vencerás suele tener la forma:

```
DyVSort(tabla T)
  si dim(T) <= dimMin :
    directSort(T); // por un alg local.
```

```

else :
    Partir(T,T1,T2);
    DyVSort(T1);
    DyVSort(T2);
    Combinar(T1,T2,T);

```

### 2.4.1. Mergesort

Una primera opción sería implementar un **Partir** sencillo y un **Combinar** complicado. Es el caso del algoritmo **Mergesort**.

```

status MergeSort(tabla T, ind P, ind U)
    si P>U :
        devolver ERROR;
    si P==U :
        devolver OK;
    else :
        M=(P+U)/2;
        MergeSort(T,P,M);
        MergeSort(T,M+1,U);
    devolver Combinar(T,P,M,U);

```

El pseudocódigo de Combinar es largo pero también bastante sencillo:

```

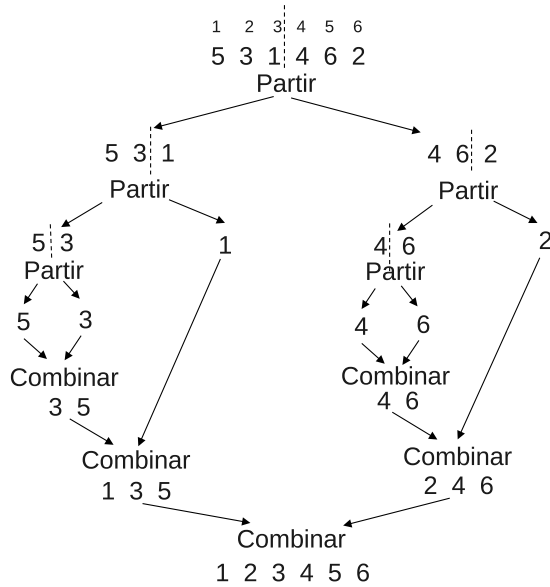
status Combinar(Tabla T,ind P,ind U,ind M)
    T'=TablaAux(P,U);
    Si T'==NULL : devolver Error;
    i=P;j=M+1;k=P;
    mientras i<=M && j<=U :
        si T[i]<T[j] : T'[k]=T[i];i++;
        else :      T'[k]=T[j];j++;
        k++;
    si i>M :
        mientras j<=U :
            T'[k]=T[j]; j++; k++;
        else si j>U :
            mientras i<=M :
                T'[k]=T[i]; i++; k++;
    Copiar(T',T,P,U);
    Free(T');
    devolver Ok;

```

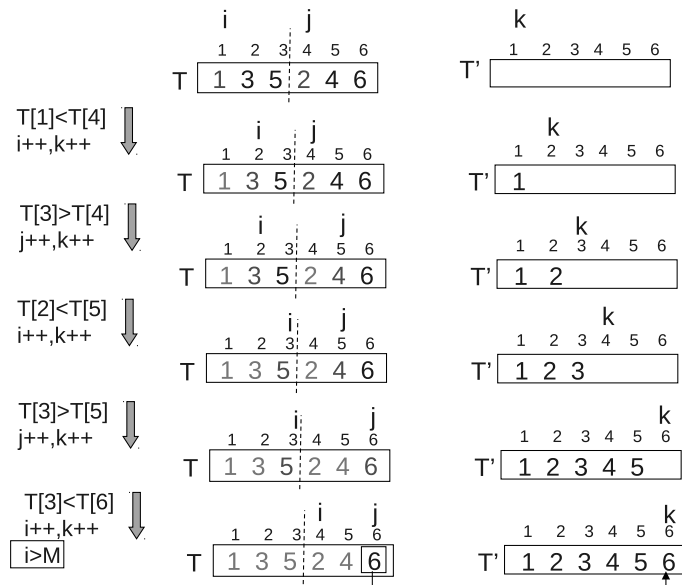


Mergesort requiere memoria auxiliar y por eso tiene status de retorno debido a los problemas que puedan aparecer en la reserva de memoria.

En la siguiente figura se puede ver la evolución de Mergesort sobre la tabla  $T = [5\ 3\ 1\ 4\ 6\ 2]$ .



El funcionamiento de **Combinar** para las subtablas  $T1 = [1\ 3\ 5]$  y  $T2 = [2\ 4\ 6]$  es el siguiente:



Finalmente copiaríamos  $T'$  en  $T$  y liberaríamos  $T'$ .

### 2.4.2. Cálculo del rendimiento de Mergesort

Al ser recursivo, encontrar la operación básica de Mergesort es un poco más complicado que antes. Antes de proceder al cálculo del rendimiento de Mergesort podemos hacer las siguientes observaciones:

1. Mergesort es un algoritmo que consta de un partir simple que divide una tabla  $\sigma$  en dos mitades  $\sigma_i, \sigma_d$ , dos llamadas recursivas y finalmente una llamada a la rutina **Combinar**, por tanto se tiene:

$$n_{MS}(\sigma) = n_{MS}(\sigma_i) + n_{MS}(\sigma_d) + n_{Combinar}(\sigma_i, \sigma_d, \sigma) \quad (2.16)$$

2. Lo anterior indica que la operación básica de Mergesort se encuentra en el bucle más interno de la rutina **Combinar**

```
si T[i]<T[j]
```

3. En el caso de Mergesort el tamaño de cada una de las subtablas  $\sigma_i, \sigma_d$  es fácil de calcular

$$\text{tamaño}(\sigma_i) = \left\lfloor \frac{N}{2} \right\rfloor \quad (2.17)$$

$$\text{tamaño}(\sigma_d) = \left\lceil \frac{N}{2} \right\rceil \quad (2.18)$$

4. El numero de comparaciones de clave que realiza **Combinar** se puede acotar inferior y superiormente:

$$\left\lfloor \frac{N}{2} \right\rfloor \leq n_{Combinar}(\sigma_i, \sigma_d, \sigma) \leq N - 1 \quad (2.19)$$

La primera desigualdad se da en el caso en el que todos los elementos de  $\sigma_i$  son más pequeños que el primer elemento de  $\sigma_d$ . La segunda desigualdad se da en el caso en el que los elementos de ambas tablas se encuentran alternados, ganando de forma alterativa las comparaciones un elemento de cada tabla (por ejemplo,  $\sigma_i = [1\ 3\ 5], \sigma_d = [2\ 4\ 6]$ ).

5. Por (2.16) y (2.19) se tiene:

$$W_{MS}(N) \leq W_{MS} \left( \left\lceil \frac{N}{2} \right\rceil \right) + W_{MS} \left( \left\lfloor \frac{N}{2} \right\rfloor \right) + N - 1 \quad (2.20)$$

Además es obvio que  $W_{MS}(1) = 0$ . A la expresión anterior se le denomina desigualdad recurrente.

### 2.4.3. Estimación del crecimiento de soluciones de desigualdades recurrentes

La forma general de una desigualdad recurrente es la siguiente:

$$\begin{aligned} T(N) &\leq T(N_1) + \dots + T(N_k) + f(N), & N_i < N \\ T(1) &= C \end{aligned} \quad (2.21)$$

donde  $C$  es una constante y  $f(N)$  es una función conocida.

En el caso de Mergesort se tiene:

$$W_{MS}(1) = 0 \quad (2.22)$$

ya que cuando la tabla  $\sigma$  tiene un único elemento no se realiza ninguna comparación de clave.

Como método general, las ecuaciones recurrentes se resuelven frecuentemente en dos pasos:

1. Se da una estimación del crecimiento de  $T$  para ciertos valores de  $N$ ; generalmente son valores sencillos que simplifican la recurrencia y facilitan su resolución, frecuentemente “desplegando” la recurrencia.
2. Se verifica la estimación anterior para cualquier valor de  $N$  mediante inducción.

En el caso de Mergesort se tiene:

$$W_{MS}(N) \leq W_{MS} \left( \left\lceil \frac{N}{2} \right\rceil \right) + W_{MS} \left( \left\lfloor \frac{N}{2} \right\rfloor \right) + N - 1$$

Vemos que sería conveniente poder **eliminar los suelos y los techos**  $\lceil \cdot \rceil$  y  $\lfloor \cdot \rfloor$ . Para ello vamos a estimar primero la función  $W_{MS}(N)$  para un caso particular. Si tomamos

$$N = 2^k \quad (2.23)$$

podemos quitar los techos y los suelos y por tanto la ecuación recurrente (2.20) se convierte en la **más sencilla** desigualdad

$$W_{MS}(N) \leq 2W_{MS}\left(\frac{N}{2}\right) + N - 1 \quad (2.24)$$

Ahora podemos **desplegar la recurrencia**. Por (2.24) se tiene que

$$W_{MS}\left(\frac{N}{2}\right) \leq 2W_{MS}\left(\frac{N}{2^2}\right) + \frac{N}{2} - 1$$

y sustituyendo en (2.24) se tiene:

$$\begin{aligned} W_{MS}(N) &\leq N - 1 + 2\left(2W_{MS}\left(\frac{N}{2^2}\right) + \frac{N}{2} - 1\right) = 2N - 1 - 2 + 2^2W_{MS}\left(\frac{N}{2^2}\right) \\ &\leq 2N - 1 - 2 + 2^2\left(\frac{N}{2^2} - 1 + 2W_{MS}\left(\frac{N}{2^3}\right)\right) \\ &\leq \dots \\ &\leq kN - (1 + 2 + 2^2 + 2^3 + \dots + 2^{k-1}) + 2^kW_{MS}\left(\frac{N}{2^k}\right) \end{aligned}$$

Por (2.23) se tiene que  $W_{MS}\left(\frac{N}{2^k}\right) = W_{MS}(1) = 0$  y por tanto:

$$W_{MS}(N) \leq kN - \sum_{i=0}^{k-1} 2^i = kN - (2^k - 1) = N \log N - N + 1,$$

esto es,

$$W_{MS}(N) = N \log N + O(N) \quad (2.25)$$

Para la demostración del caso general aplicamos inducción:

Para  $N = 1$  se tiene

$$W(1) = 0 \leq 1 \log 1 + O(1)$$

Para el caso general tenemos:

$$\begin{aligned}
W_{MS}(N) &\leq N - 1W_{MS}\left(\left\lceil\frac{N}{2}\right\rceil\right) + W_{MS}\left(\left\lfloor\frac{N}{2}\right\rfloor\right) \\
&\stackrel{\text{por inducción}}{\leq} N - 1 + \left\lceil\frac{N}{2}\right\rceil \log\left\lceil\frac{N}{2}\right\rceil + O\left(\left\lceil\frac{N}{2}\right\rceil\right) + \left\lfloor\frac{N}{2}\right\rfloor \log\left\lfloor\frac{N}{2}\right\rfloor + O\left(\left\lfloor\frac{N}{2}\right\rfloor\right) \\
&\stackrel{\log\lfloor N/2\rfloor \leq \log\lceil N/2\rceil}{\leq} N - 1 + \left(\left\lceil\frac{N}{2}\right\rceil + \left\lfloor\frac{N}{2}\right\rfloor\right) \log\left\lceil\frac{N}{2}\right\rceil + O\left(\left\lceil\frac{N}{2}\right\rceil + \left\lfloor\frac{N}{2}\right\rfloor\right) \\
&= N - 1 + N \log\left\lceil\frac{N}{2}\right\rceil + O(N) \\
&= N - 1 + N \log\left\lceil\frac{N}{2}\right\rceil + O(N) + \log\frac{N}{2} - \log\frac{N}{2} \\
&= N - 1 + N \log N - N + N \log\frac{\lceil N/2\rceil}{N/2} + O(N) \\
&\stackrel{\lfloor (N/2)\rfloor \leq (N+1)/2}{\leq} N \log N + N \log\frac{N+1}{2} + O(N) \\
&= N \log N + \underbrace{\log\left(1 + \frac{1}{N}\right)^N}_{\simeq e=O(1)} + O(N) \\
&= N \log N + O(N)
\end{aligned}$$

Es decir, Mergesort tiene un peor coste ( $N \log N$ ) muy bueno en el sentido de que el aumento es mucho menor que en los algoritmos donde el caso peor crecía como  $N^2/2$ .

Para el caso mejor de Mergesort  $B_{MS}(N)$  volvemos a usar la identidad (2.16) y la acotación (2.19) con lo que se obtiene:

$$B_{MS}(N) \geq B_{MS}\left(\left\lceil\frac{N}{2}\right\rceil\right) + B_{MS}\left(\left\lfloor\frac{N}{2}\right\rfloor\right) + \left\lfloor\frac{N}{2}\right\rfloor \quad (2.26)$$

y además

$$B_{MS}(1) = 0 \quad (2.27)$$

Con lo que volvemos a obtener una ecuación recurrente. Al igual que en el caso peor tomamos primero un caso sencillo para  $N$ ,  $N = 2^k$  con lo que se tiene:

$$B_{MS}(N) \geq 2B_{MS}\left(\frac{N}{2}\right) + \frac{N}{2} \quad (2.28)$$

Si desplegamos la recurrencia obtenemos

$$\begin{aligned} B_{MS}(N) &\geq 2B_{MS}\left(\frac{N}{2}\right) + \frac{N}{2} \geq \frac{N}{2} + 2\left(2B_{MS}\left(\frac{N}{2^2}\right) + \frac{N}{2^2}\right) \\ &= \frac{N}{2} + \frac{N}{2} + 2^2 B_{MS}\left(\frac{N}{2^2}\right) \geq \dots \\ &\geq k\frac{N}{2} + 2^k B_{MS}\left(\frac{N}{2^k}\right) = \frac{N}{2} \log N \end{aligned}$$

ya que  $N = 2^k \Rightarrow k = \log N$  y  $B_{MS}\left(\frac{N}{2^k}\right) = B_{MS}(1) = 0$ .

Por tanto, si  $N = 2^k$  tenemos:

$$B_{MS}(N) \geq \frac{N}{2} \log N \quad (2.29)$$

y en general

$$B_{MS}(N) = \frac{N}{2} \log N + O(N). \quad (2.30)$$

El análisis preciso del coste medio es complicado pero por (1.40), (2.25) y (2.29) se tiene que:

$$\frac{N}{2} \log N \leq B_{MS}(N) \leq A_{MS}(N) \leq W_{MS}(N) \leq N \log N + O(N)$$

por lo que  $A_{MS}(N) = \Theta(N \log N)$ .

Mergesort tiene un caso medio muy bueno; sin embargo tiene algunos costes ocultos que perjudican su rendimiento:

- Necesita memoria auxiliar.
- Es recursivo. Esto es, tiene costes ocultos de gestión de la recursión que no estamos teniendo en cuenta pero que pesan mucho en el coste efectivo de ejecución.

#### 2.4.4. Quicksort

Otra opción distinta a la que toma Mergesort es hacer una rutina **Partir** complicada y una rutina **Combinar** más simple. Este es el caso del algoritmo **Quicksort**.

En Quicksort la rutina Partir elige una posición  $M$  de la tabla  $T$  de tal forma que el valor  $T[M]$  (pivote) esté aproximadamente en el medio de la tabla, luego divide la tabla  $T$  en dos subtablas no ordenadas  $T_i, T_d$  tales que:

$$\begin{aligned} T_i[j] &< T[M] & P \leq j < M \\ T_d[j] &> T[M] & M < j < U \end{aligned}$$

Después de una llamada a **Partir** la tabla  $T$  contiene los elementos colocados de la siguiente forma:

$$\underbrace{T_i}_{\text{elementos } < T[M]} \quad T[M] \quad \underbrace{T_d}_{\text{elementos } > T[M]}$$

donde las subtablas  $T_i$  y  $T_d$  no están ordenadas.

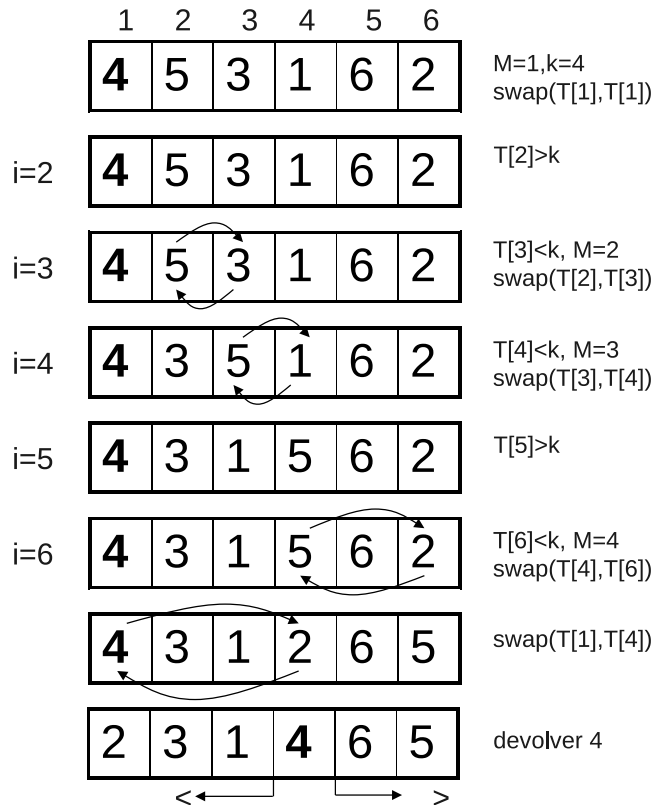
El pseudocódigo de Quicksort es el siguiente

```
status QS(tabla T, ind P, ind U)
  si P>U :
    devolver ERROR;
  si P==U :
    devolver OK;
  else :
    M=Partir(T,P,U);
    si P<M-1 :
      QS(T,P,M-1);
    si M+1 < U :
      QS(T,M+1,U);
  devolver OK;
```

Se observa que tras las dos llamadas recursivas las subtablas izquierda y derecha ya están ordenadas por lo que no hace falta combinarlas. Vemos ahora el pseudocódigo de la rutina Partir:

```
ind Partir(tabla T, ind P, ind U)
  M=Medio(T,P,U);
  k=T[M];
  swap(T[P],T[M]);
  M=P;
  para i de P+1 a U :
    si T[i]<k :
      M++;
      swap(T[i],T[M]);
  swap(T[P],T[M]);
  devolver M;
```





En la figura 2.4.4 se puede ver la evolución de **Partir** sobre la tabla  $T = [4, 5, 3, 1, 6, 2]$ .

Se observa que no hay posibilidad de errores en **Partir** ni errores internos en Quicksort. Si existe la posibilidad que Quicksort reciba argumentos erróneos. Quicksort es un método *in-place* que no necesita memoria auxiliar.

La función **Medio** devuelve la posición del pivote. Hay muchas formas de implementar medio; las más habituales son:

1. Devolver  $P$  (es la que usaremos a lo largo de este curso).
2. Devolver  $U$ .
3. Devolver  $M = \lfloor \frac{P+U}{2} \rfloor$ .
4. Devolver el valor que este en el medio de  $T[P], T[M], T[U]$ .

No es conveniente que en **Medio** haya un número de comparaciones de clave que dependa de  $N$  ya que estas comparaciones de clave afectarían al rendimiento de Quicksort.

### 2.4.5. Rendimiento de Quicksort

La operación básica de Quicksort es la comparación de claves que se encuentra en **Partir**

si  $T[i] < k$

El número de comparaciones de clave que realiza **Partir** sólo depende del tamaño de la entrada,

$$n_{Partir}(\sigma) = N - 1 \quad \forall \sigma \in \Sigma_N \quad (2.31)$$

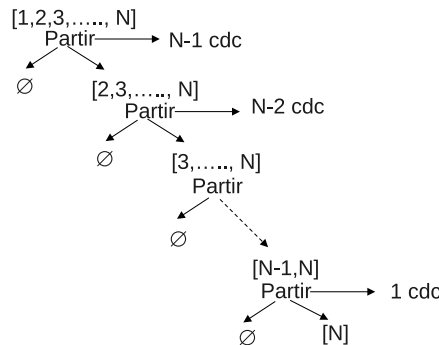
Por tanto podemos establecer la siguiente recurrencia para el número de operaciones básicas que realiza Quicksort

$$\begin{aligned} n_{QS}(\sigma) &= n_{Partir}(\sigma) + n_{QS}(\sigma_i) + n_{QS}(\sigma_d) \\ &= N - 1 + n_{QS}(\sigma_i) + n_{QS}(\sigma_d) \end{aligned} \quad (2.32)$$

Vamos a ver algunos casos particulares: Empezamos calculando  $n_{QS}([1, 2, 3, \dots, N])$ .

La primera llamada a **Partir** divide la tabla  $\sigma = [1, 2, 3, \dots, N]$  en dos subtablas,  $\sigma_i = [1]$  y  $\sigma_d = [2, 3, \dots, N]$  necesitando para ello  $N - 1$  comparaciones de clave. Además no hay recursión en  $\sigma_i$  y sólo en  $\sigma_d$ . La siguiente llamada a **Partir** divide la tabla  $\sigma = [2, \dots, N]$  en dos subtablas,  $\sigma_i = [2]$  y  $\sigma_d = [3, \dots, N]$  necesitando para ello  $N - 2$  comparaciones de clave.

Así iteraríamos sobre tablas un elemento mas pequeño cada vez. El proceso se puede ver en la siguiente figura:



Por tanto

$$n_{QS}([1, 2, 3, \dots, N]) = (N - 1) + (N - 2) + \dots + 2 + 1 = \frac{N^2}{2} - \frac{N}{2} \quad (2.33)$$

Por tanto, con esta elección del pivote, Quicksort da un rendimiento muy malo para una tabla  $\sigma$  ordenada. Una situación similar a la anterior se produce con la tabla  $\sigma = [N \ N - 1 \ \dots \ 3 \ 2 \ 1]$ .

### 2.4.6. Caso peor de Quicksort\*

Sea  $\sigma \in \Sigma_N$ . Supongamos que  $\sigma(1) = i$ , entonces:

- i.  $\sigma_i \in \Sigma_{i-1}$
- ii.  $\sigma_d \in \Sigma_{N-i}$  En realidad  $\sigma_d$  no es una permutación de  $N - i$  elementos ya que no contiene estrictamente números del 1 al  $N - i$  ( $\sigma_d(j) - i$  sí lo es); sin embargo si que se comporta como una permutación de  $N - i$  elementos en cuanto al número de comparaciones de clave que **Partir** va a realizar sobre esta tabla.

Por (2.32) tenemos:

$$n_{QS}(\sigma) = N - 1 + n_{QS}(\sigma_i) + n_{QS}(\sigma_d)$$

y entonces

$$\begin{aligned} n_{QS}(\sigma) &\leq N - 1 + W_{QS}(i - 1) + W_{QS}(N - i) \\ &\leq N - 1 + \max_{1 \leq i \leq N} \{W_{QS}(i - 1) + W_{QS}(N - i)\} \end{aligned} \quad (2.34)$$

Por lo tanto, se llega a la desigualdad recurrente para  $N$

$$W_{QS}(N) \leq N - 1 + \max_{1 \leq i \leq N} \{W_{QS}(i - 1) + W_{QS}(N - i)\} \quad (2.35)$$

$$W_{QS}(1) = 0 \quad (2.36)$$

Hemos visto que

$$n_{QS}([1, 2, 3, \dots, N]) = \frac{N^2}{2} - \frac{N}{2} \Rightarrow W_{QS}(N) \geq \frac{N^2}{2} - \frac{N}{2} \quad (2.37)$$

con lo cual, asumiendo que Quicksort no va a ser peor que los algoritmos locales que hemos visto, es razonable suponer que  $W_{QS}(N) = \frac{N^2}{2} - \frac{N}{2}$ . Vamos demostrar por inducción que la anterior suposición es cierta.

Por tanto suponemos que  $\forall N' < N$  se verifica  $W_{QS}(N') = \frac{N'^2}{2} - \frac{N'}{2}$ . Aplicando esto a (2.35) se tiene:

$$\begin{aligned}
W_{QS}(N) &\leq N - 1 + \max_{1 \leq i \leq N} \left\{ \frac{(i-1)^2}{2} - \frac{i-1}{2} + \frac{(N-i)^2}{2} - \frac{N-i}{2} \right\} \\
&= \max_{1 \leq i \leq N} \left\{ N - 1 + \frac{i^2}{2} - i + \frac{1}{2} - \frac{i}{2} + \frac{1}{2} + \frac{N^2}{2} - Ni + \frac{i^2}{2} - \frac{N}{2} + \frac{i}{2} \right\} \\
&= \max_{1 \leq i \leq N} \left\{ \frac{N^2}{2} - \frac{N}{2} + N + i^2 - i - Ni \right\} \\
&= \frac{N^2}{2} - \frac{N}{2} + \underbrace{\max_{1 \leq i \leq N} \left\{ \underbrace{(i-1)}_{\geq 0} \underbrace{(N-i)}_{\leq 0} \right\}}_{\leq 0} \\
&\leq \frac{N^2}{2} - \frac{N}{2}.
\end{aligned}$$

Por tanto se tiene:

$$W_{QS}(N) \leq \frac{N^2}{2} - \frac{N}{2} \quad (2.38)$$

que junto a (2.37) nos permite afirmar:

$$W_{QS}(N) = \frac{N^2}{2} - \frac{N}{2} \quad (2.39)$$

### 2.4.7. Caso medio de Quicksort

De nuevo vamos a aplicar el resultado obtenido en (2.33)

$$n_{QS}(\sigma) = N - 1 + n_{QS}(\sigma_i) + n_{QS}(\sigma_d)$$

con  $\sigma_i \in \Sigma_{i-1}$ ,  $\sigma_d \in \Sigma_{N-i}$ . Vamos a simplificar la expresión anterior suponiendo que  $n_{QS}(\sigma_i) \simeq A_{QS}(i-1)$  y  $n_{QS}(\sigma_d) \simeq A_{QS}(N-i)$  con lo que se tiene

$$A_{QS}(N) = N - 1 + A_{QS}(i-1) + A_{QS}(N-i),$$

que todavía depende del valor de  $i$ . Para eliminarlo, suponemos **equiprobabilidad** en  $i$  y llegamos a

$$A_{QS}(N) = N - 1 + \frac{1}{N} \sum_{i=1}^N \{A_{QS}(i-1) + A_{QS}(N-i)\}$$

junto con

$$A_{QS}(1) = 0 \quad (2.40)$$

En este caso tenemos que resolver una igualdad recurrente algo más complicada, con lo cual, vamos a intentar simplificarla. Si desarrollamos el sumatorio en  $i$  vemos que cada término se repite dos veces, por tanto:

$$\begin{aligned} A_{QS}(N) &= N - 1 + \frac{1}{N} \sum_{i=1}^N \{A_{QS}(i-1) + A_{QS}(N-i)\} \\ &= N - 1 + \frac{2}{N} \sum_{i=1}^{N-1} A_{QS}(i) \end{aligned}$$

La división por  $N$  complica las operaciones, por lo que multiplicamos ambos términos por  $N$  y se tiene:

$$NA_{QS}(N) = N(N-1) + 2 \sum_{i=1}^{N-1} A_{QS}(i).$$

El sumatorio también complica las operaciones, pero es fácil de quitar, pues sustituyendo  $N$  por  $N-1$  se tiene

$$(N-1)A_{QS}(N-1) = (N-1)(N-2) + 2 \sum_{i=1}^{N-2} A_{QS}(i),$$

cuyo sumatorio tiene los mismos términos que el anterior menos el correspondiente a  $N-1$ . Estos términos repetidos se eliminan restando término a término estas dos últimas expresiones y viendo que  $\sum_{i=1}^{N-1} A_{QS}(i) - \sum_{i=1}^{N-2} A_{QS}(i) = A_{QS}(N-1)$  obtenemos

$$\begin{aligned} NA_{QS}(N) - (N-1)A_{QS}(N-1) &= 2(N-1) + 2A_{QS}(N-1) \\ \Rightarrow NA_{QS}(N) &= 2(N-1) + (N+1)A_{QS}(N-1) \\ \Rightarrow \frac{NA_{QS}(N)}{N(N+1)} &= \frac{2(N-1)}{N(N+1)} + \frac{(N+1)A_{QS}(N-1)}{N(N+1)} \\ \Rightarrow \frac{A_{QS}(N)}{N+1} &= \frac{2(N-1)}{N(N+1)} + \frac{A_{QS}(N-1)}{N}. \end{aligned}$$

Por tanto, si definimos la función

$$B(N) = \frac{A_{QS}(N)}{N+1} \tag{2.41}$$

se tiene  $B(1) = \frac{A_{QS}(1)}{2} = 0$  y la última identidad se puede escribir:

$$\begin{aligned} B(N) &= \frac{2(N-1)}{N(N+1)} + B(N-1) \\ B(1) &= 0 \end{aligned}$$

Se trata de una recurrencia mucho más sencilla que se resuelve desplegando ahora  $B(N)$  y obtenemos:

$$\begin{aligned} B(N) &= \frac{2(N-1)}{N(N+1)} + B(N-1) = \frac{2(N-1)}{N(N+1)} + \frac{2(N-2)}{(N-1)(N)} + B(N-2) = \dots \\ &= \frac{2(N-1)}{N(N+1)} + \frac{2(N-2)}{(N-1)N} + \frac{2(N-3)}{(N-2)(N-1)} + \dots + \underbrace{B(1)}_0 \\ &= 2 \sum_{j=1}^{N-1} \frac{j}{(j+1)(j+2)} = 2 \sum_{j=1}^{N-1} \frac{j+1-1}{(j+1)(j+2)} \\ &= 2 \underbrace{\sum_{j=1}^{N-1} \frac{1}{j+2}}_{H_N + O(1) \sim \log N + O(1)} - 2 \underbrace{\sum_{j=1}^{N-1} \frac{1}{(j+1)(j+2)}}_{O(1)} = 2 \log N + O(1) \end{aligned}$$

Por tanto

$$B(N) = \frac{A_{QS}(N)}{N+1} = 2 \log N + O(1),$$

y ahora deshacemos el cambio de función llegando a

$$A_{QS}(N) = (N+1)(2 \log N + O(1)) = A_{QS}(N) = 2N \log N + O(N).$$

Por tanto Quicksort tiene un caso medio muy bueno y además no requiere memoria adicional; sin embargo tiene dos inconvenientes

1. Su caso peor es malo.
2. Es recursivo.

Vemos cómo eliminar ambos.

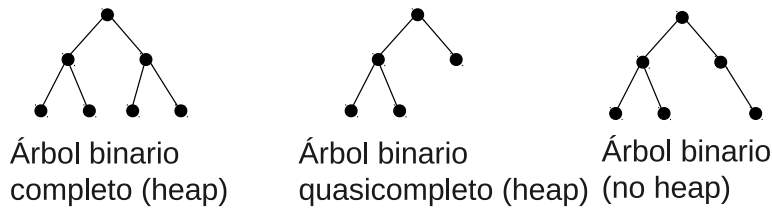


Figura 2.1: ABs que son (o no).

### 2.4.8. Heapsort

Definimos un heap como un árbol binario completo o quasi-completo. En la figura 2.1 se pueden ver dos ejemplos de heaps y un ejemplo de un árbol que no es heap. Una propiedad de los heaps es que se almacenan muy bien en una tabla recorriéndolos de arriba a abajo y de izquierda a derecha.

Un **Max-Heap** es un heap tal que para todo subárbol  $T'$  de  $T$  se verifica

$$\text{info}(T') > \text{info}(T'_i), \text{info}(T'_d) \quad (2.42)$$

Es decir, el valor de cada nodo del árbol es mayor que el valor de cualquiera de los nodos que hay por debajo de él.

Un ejemplo de Max-Heap es el de la figura 2.2

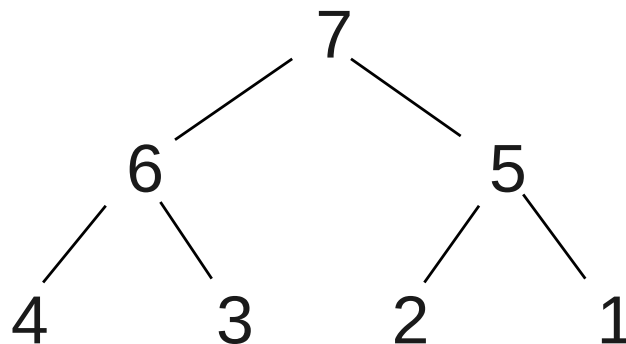
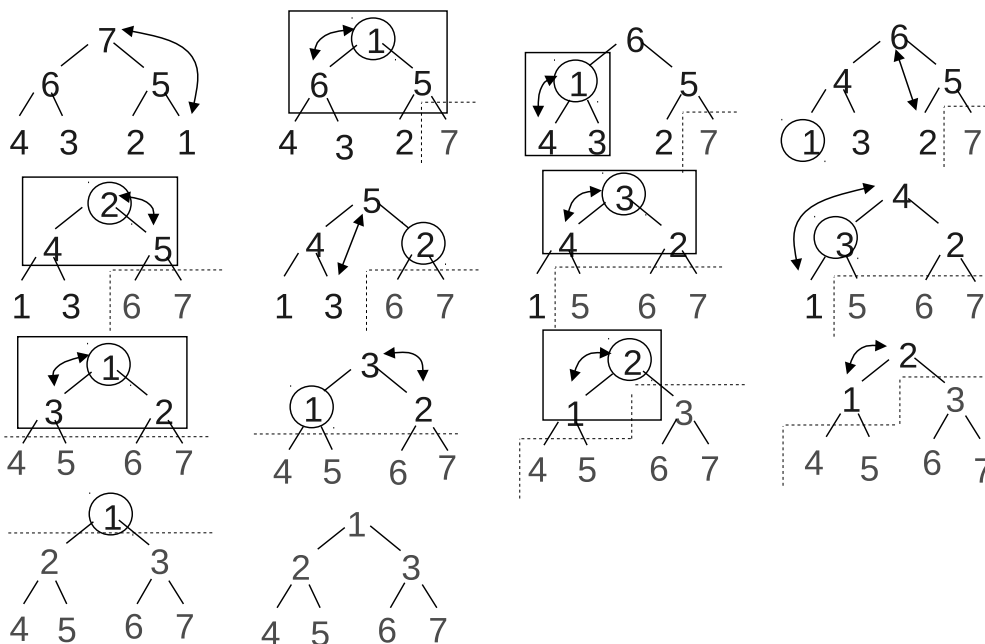


Figura 2.2: Ejemplo de Max-Heap.

Un Max-Heap tiene la propiedad de que es fácilmente ordenable. Para ordenar un Max-Heap se siguen los siguientes pasos:

- Paso 1. Se intercambian el nodo raíz con el último nodo del árbol (el situado abajo a la derecha) y se extrae este nodo del árbol.
- Paso 2. Se mantiene la condición de Heap del nuevo nodo raíz, comparándolo con sus dos hijos e intercambiándolo con el mayor de ellos. Si el nodo ya es mayor que los dos hijos, el árbol es un Max-Heap; si no, se intercambia con el mayor de los hijos y se repite la operación hasta que el nodo no tenga o sea mayor que sus hijos.
- Paso 3. Si quedan nodos en el árbol ir al Paso 1.

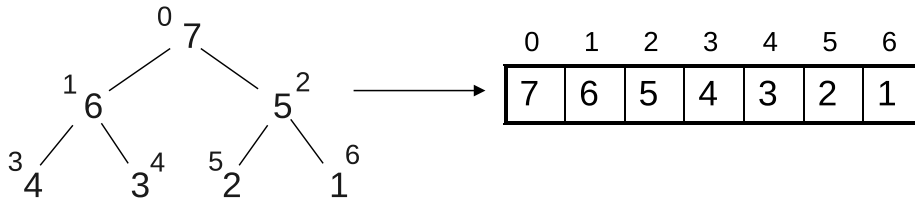
En el dibujo inferior se puede ver el proceso de ordenación del Heap  $T = [7\ 6\ 5\ 4\ 3\ 2\ 1]$ .



Los elementos que están bajo la línea discontinua no se considera que están en el árbol a efectos de desarrollo del método de ordenación de un Max-Heap. El elemento dentro en un círculo es el que se compara (y si es necesario se intercambia) con sus hijos con el fin de mantener la condición de Max-Heap.

Como se ha dicho, una ventaja de los Heaps es que se pueden representar de forma fácil en una tabla, simplemente recorriendo el árbol de arriba a abajo y de





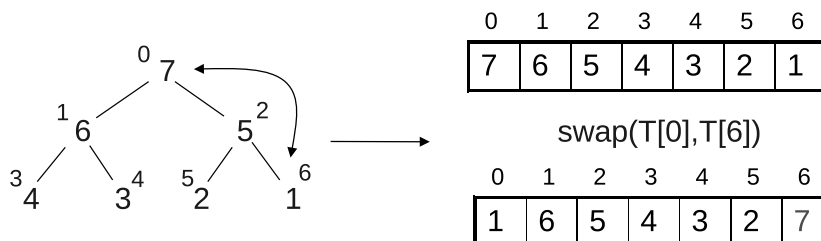
izquierda a derecha. En el siguiente dibujo se puede ver como se puede representar el Max-Heap anterior en una tabla.

A la vista del dibujo anterior, dado el índice de un nodo es fácil saber cual es la posición en la tabla en la que se encuentra el padre o cada uno de sus hijos.

Padre	Hijo Izq	Hijo Der.
$j$	$2j + 1$	$2j + 2$

Hijo	Padre
$j$	$\lfloor (j - 1) / 2 \rfloor$

El intercambio de dos elementos del árbol se puede realizar mediante un swap de elementos de la tabla como se puede ver en el siguiente ejemplo:



No cualquier Heap (tabla desordenada) corresponde a un Max-Heap, por tanto, antes de poder ordenar tabla por el método anterior será necesario convertir el Heap en un Max-Heap. Para ello basta con ir manteniendo desde el penúltimo nivel del heap la condición de Heap para cada nodo del árbol de izquierda a derecha y de abajo arriba. El proceso se puede ver en el siguiente dibujo.

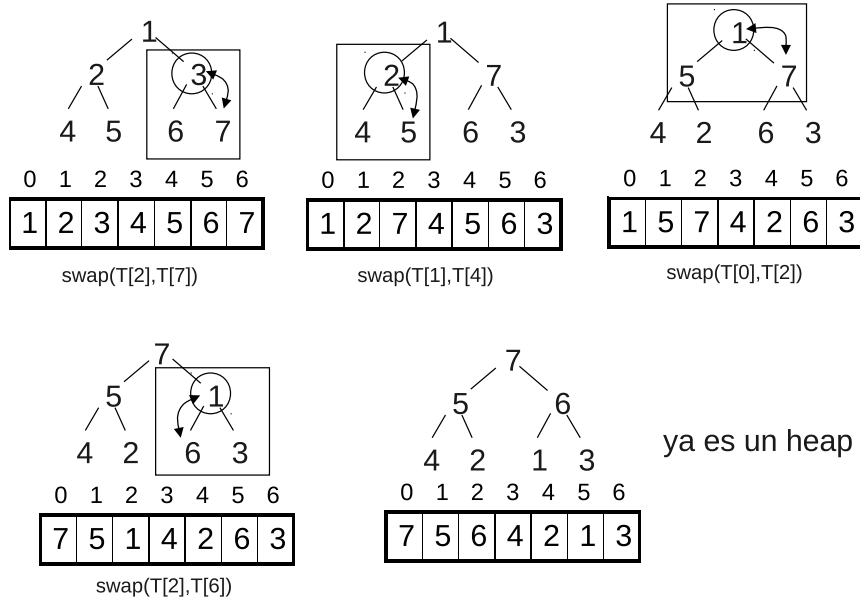
Ahora podemos dar el siguiente pseudocódigo para Mergesort:

```

HeapSort(tabla T, int N)
  CrearHeap(T,N);
  OrdenarHeap(T,N);
    
```

```

CrearHeap(tabla T, int N)
  si N < 2 :
    
```



```

return;
para i de (N-2)/2 a 0 :
    heapify(T,N,i);

```

```

OrdenarHeap(tabla T, int N)
para i de N-1 a 1 :
    swap(T[0],T[i]);
    heapify(T,N,0);

```

```

heapify(tabla T,int N,ind i)
mientras ( 2*i+1 <= N) :
    ind=max(T,N,i,izq(i),der(i));
    si (ind != i) :
        swap(A[i],A[ind]) ;
        i = ind;
    else :
        return;

```

Aquí la función  $\text{max}(T,N,i,\text{izq}(i),\text{der}(i))$  devuelve el índice del elemento de la tabla  $T$  que contiene el valor mayor entre  $i$ ,  $\text{izq}(i)$ ,  $\text{der}(i)$ . Observamos que Heapsort es no recursivo y no necesita memoria auxiliar.

### 2.4.9. Rendimiento de Heapsort

En Heapsort la operación básica es la comparación de clave dentro de **Heapify**. Además se tiene:

$$n_{HS}(T) = n_{CrearHeap}(T) + n_{OrdenarHeap}(T) \quad (2.43)$$

Vamos a calcular  $n_{CrearHeap}(T)$ . Observamos que para cada nodo del árbol, la rutina **CrearHeap** realiza como máximo  $prof(T)$  comparaciones de clave, donde  $prof(T)$  es la profundidad del árbol  $T$ , ya que por cada comparación de clave que realiza un nodo pueden ocurrir dos cosas:

- i. o bien el nodo es mayor que sus hijos (o no tiene hijos) con lo cual el nodo se queda donde está y se pasa al nodo siguiente,
- ii. o bien el nodo es menor que uno de sus hijos y por tanto desciende un nivel, lo cual solo puede ocurrir si el nodo tiene hijos, y por tanto no ha llegado a la máxima profundidad del árbol  $T$ .

Como  $T$  es un árbol quasi-completo con  $N$  nodos se tiene  $prof(T) = \lfloor \log N \rfloor$  y por tanto, dado que  $T$  tiene  $N$  nodos:

$$n_{CrearHeap}(T) \leq N \lfloor \log N \rfloor \quad (2.44)$$

Por un argumento análogo al empleado para  $n_{CrearHeap}(T)$  podemos deducir que:

$$n_{OrdenarHeap}(T) \leq N \lfloor \log N \rfloor \quad (2.45)$$

Por tanto, usando (2.45), (2.46) y (2.47) se tiene, si  $T$  es un Heap de tamaño  $N$ .

$$n_{HS}(T) \leq N \lfloor \log N \rfloor + N \lfloor \log N \rfloor = O(N \log N) \quad (2.46)$$

y por tanto:

$$W_{HS}(N) = O(N \log N) \quad (2.47)$$

Es fácil ver además usando la tabla adecuada ( $T = [1 \ 2 \ 3 \ \dots \ N - 1 \ N]$ ) que:

$$W_{HS}(N) = \Theta(N \log N) \quad (2.48)$$

Es decir, HeapSort es un algoritmo no-recursivo, que no necesita memoria auxiliar y cuyo caso peor es  $\Theta(N \log N)$ . A modo de resumen podemos escribir la siguiente tabla:

Algoritmo	A(N)	W(N)	Observaciones
Algoritmos locales (BS,IS,SS)	$\frac{N^2}{4} + O(N)$	$\frac{N^2}{2} + O(N)$	Deshacen a lo sumo una inversión por cada cdc
ShellSort	$N^k + O(N)$	$N^k + O(N)$	$1 < k < 2$
MergeSort	$\Theta(N \log N)$	$N \log N + O(N)$	Recursivo, Memoria Auxiliar
QuickSort	$\Theta(N \log N)$	$\frac{N^2}{2} + O(N)$	Recursivo, Sin memoria Auxiliar
MergeSort	$\Theta(N \log N)$	$\Theta(N \log N)$	No Recursivo, Sin memoria Auxiliar
<b>Límite</b>	$\Theta(N)$	$\Theta(N)$	¿alcanzable?

## 2.5. Cotas inferiores para algoritmos de ordenación por comparación de claves

De momento no hemos encontrado ningún algoritmo que mejore la cota inferior  $\Theta(N \log N)$ . Vamos a ver si  $\Theta(N \log N)$  es realmente una cota inferior para todos los algoritmos ordenación por comparación de claves o si, por contra, esa cota se puede mejorar.

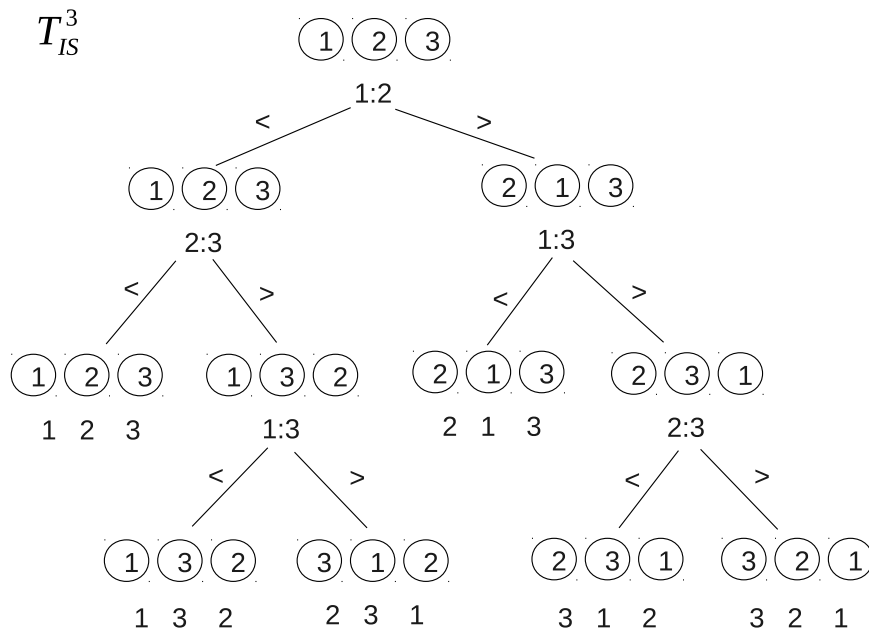
### 2.5.1. Árboles de decisión para algoritmos de ordenación por comparación de claves

Si  $A$  es un algoritmo de ordenación por comparación de claves y  $N$  es un tamaño de tabla, se puede construir su **árbol de decisión de tamaño  $N$** ,  $T_A^N$  para  $\sigma \in \Sigma_N$  como un árbol binario que cumple las siguientes condiciones:

1. Contiene nodos de la forma **i:j** ( $i < j$ ) que indica la comparación de clave entre los elementos inicialmente en las posiciones  $i$  y  $j$ .
2. El subárbol izquierdo del nodo **i:j** en  $T_A^N$  contiene las posibles comparaciones de clave que realiza el algoritmo  $A$  si  $T[i] < T[j]$ .
3. El subárbol derecho del nodo **i:j** en  $T_A^N$  contiene las posibles comparaciones de clave que realiza el algoritmo  $A$  si  $T[i] > T[j]$ .

4. A cada  $\sigma \in \Sigma_N$  le corresponde una única hoja  $H_\sigma$  en  $T_A^N$  y los nodos entre la raíz y la hoja  $H_\sigma$  son las comparaciones de clave que realiza el algoritmo  $A$  al recibir la permutación  $\sigma$  y ordenarla.

En el siguiente dibujo se puede ver el árbol de decisión de InsertSort para  $N = 3$ .



A raíz de la definición anterior podemos establecer algunas propiedades de los árboles de decisión

1. El número de hojas en  $T_A^N$  es  $N! = |\Sigma_N|$ .
2. Para una permutación  $\sigma \in \Sigma_N$  se tiene  $n_A(\sigma) =$  número de comparaciones de clave = profundidad de la hoja  $H_\sigma$  en  $T_A^N$ , y por tanto:

$$n_A(\sigma) = \text{prof}_{T_A^N}(H_\sigma) \tag{2.49}$$


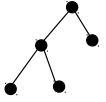
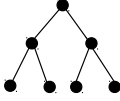
3.  $W_A(N) = \max_{\sigma \in \Sigma_N} n_A(\sigma) = \max_{\sigma \in \Sigma_N} \text{prof}_{T_A^N}(H_\sigma)$ .

Es decir, estamos cambiando el cálculo de una cantidad algorítmica por una cantidad topológica.

### 2.5.2. Cotas inferiores para el caso peor

Vamos a estimar cuál es la profundidad mínima que tiene un árbol binario de  $H$  hojas y cómo podemos, a partir de esta cantidad, estimar una cota inferior para el caso peor.

En la tabla siguiente podemos ver cual es la profundidad menor que puede tener un árbol binario de  $H$  hojas.

Nº de hojas H	AB <sup>Mínimo</sup> (H)	Prof. Mínima
1	•	0
2		1
3		2
4		2
....	....	

Es decir, parece que:

$$\text{Prof mínima AB con H hojas} = \lceil \log H \rceil, \quad (2.50)$$

con lo cual se tiene para el caso peor:

$$\begin{aligned} W_A(N) &= \max_{\sigma \in \Sigma_N} \text{prof}_{T_A^N}(H_\sigma) \geq \text{prof. mín. de un árbol binario con } N! \text{ hojas} \\ &= \lceil \log N! \rceil \end{aligned}$$

Por (1.30) sabemos que  $\lceil \log N! \rceil = \Theta(N \log N)$  y por tanto podemos decir que si  $A$  es un algoritmo de ordenación por comparación de clave

$$W_A(N) = \Omega(N \log N) \quad (2.51)$$

Es decir, no se puede construir un algoritmo de ordenación por comparación de claves cuyo tiempo peor sea mejor que  $(N \log N)$ . Por tanto dado un algoritmo de la familia

$$\xi = \{A : \text{algoritmos de ordenación por comparación de clave}\}$$

su rendimiento nunca va a ser mejor que el de HeapSort o MergeSort. Es decir, HeapSort y MergeSort son óptimos en el caso peor.

### 2.5.3. Cotas inferiores para el caso medio

Sea  $A \in \xi = \{A : \text{algoritmos de ordenación por comparación de clave}\}$ . Por (1.39) se tiene:

$$\begin{aligned} A_A(N) &= \frac{1}{N!} \sum_{\sigma \in \Sigma_N} n_A(\sigma) = \frac{1}{N!} \sum_{H \in T_A^N} \text{prof}_{T_A^N}(H) \\ &= \text{profundidad media de } T_A^N \geq P_{Mmin}(N!) \end{aligned}$$

y por tanto:

$$A_A(N) \geq P_{Mmin}(N!) \quad (2.52)$$

donde  $P_{Mmin}(k)$  es la mínima profundidad media de un árbol binario de  $k$  hojas.

$$P_{Mmin}(k) = \text{mín}\{\text{profundidad media}(T) : T \text{ árbol binario de } k \text{ hojas}\} \quad (2.53)$$

Observamos que de nuevo hemos pasado de una cantidad algorítmica a una cantidad topológica.

Definimos ahora la **longitud de caminos externos** de un árbol binario  $T$  como:

$$LCE(T) = \sum_{H \text{ hoja de } T} \text{prof}_T(H) \quad (2.54)$$

En la figura 2.3 se puede ver el cálculo de la  $LCE$  para un cierto árbol  $T$ .

Con esta definición se tiene:

$$A_A(N) \geq \frac{1}{N!} LCE_{min}(N!) \quad (2.55)$$

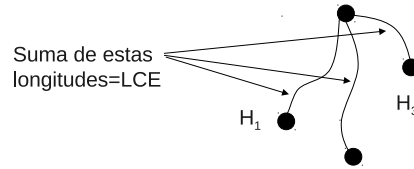


Figura 2.3: Cálculo de la LCE.

donde

$$LCE_{min}(k) = \min\{LCE(T), T \text{ rmtiene } k \text{ hojas}\} \quad (2.56)$$

Al igual que hicimos para la cota inferior del caso peor, vamos a calcular  $LCE_{min}(k)$  para algunos valores de  $k$  con el fin de obtener una idea de cómo puede comportarse esta cantidad. En la tabla 2.4 siguiente podemos ver cuál es el  $LCE_{min}$  que puede tener un árbol binario de  $k$  hojas.

k	T Óptimo	$LCE_{min}(k)$
1	•	0
2	<pre>       •      / \     •   •           </pre>	$2(1+1)$
3	<pre>       •      / \     •   •    / \   •   •           </pre>	$5(2+2+1)$
4	<pre>       •      / \     •   •    / \ / \   •   • • •           </pre>	$8(2+2+2+2)$
5	<pre>       •      / \     •   •    / \ / \   •   • • •  / \ •   •           </pre>	$12(3+3+2+2+2)$
....	....	....

Figura 2.4: Posibles valores de LCE mínimas.

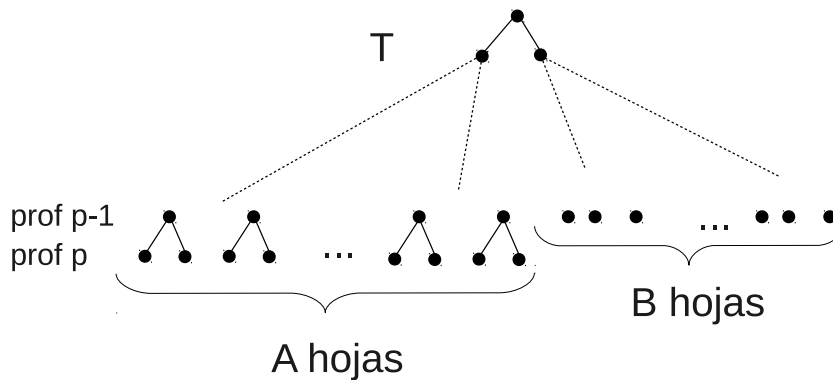
Vamos a demostrar que:

$$LCE_{min}(k) = k \lceil \log k \rceil + k - 2^{\lceil \log k \rceil} \quad (2.57)$$



Observamos que las hojas de estos árboles mínimos solamente están a profundidad  $p$  o profundidad  $p - 1$ : de estar a profundidad mayor es claro que no darían lugar a LCEs óptimas. Observamos también que estos árboles tienen siempre un número par de hojas a profundidad  $p$ , pues de haber una hoja “suelta.” a profundidad  $p$ , se la puede subir a profundidad  $p - 1$  reduciéndose la LCE.

Sea  $A$  el número de hojas de  $T$  a profundidad  $p$  y  $B$  el número de hojas a profundidad  $p - 1$ . En el dibujo 2.5.3 se puede ver el significado de los valores  $A$  y  $B$ .



Se tiene:

$$A + B = k \tag{2.58}$$

y además

$$\frac{A}{2} + B = \text{Número de hojas a profundidad } p - 1$$

Un árbol completo de profundidad  $p$  tiene  $2^p$  hojas, por tanto:

$$\frac{A}{2} + B = 2^{p-1} \Rightarrow A + 2B = 2^p \tag{2.59}$$

Por otro lado si restamos (2.60) y (2.61) se tiene

$$\begin{array}{r} A + 2B = 2^p \\ - \quad A + B = k \\ \hline 0 + B = 2^p - k \end{array}$$

Considerando que  $p = \lceil \log k \rceil$  tenemos

$$B = 2^{\lceil \log k \rceil} - k$$

Como hay  $A$  hojas a profundidad  $p$  y  $B$  no hojas a profundidad  $p - 1$  se tiene:

$$LCE_{min}(k) = Ap + B(p - 1) = (A + B)p - B = kp - B = k \lceil \log k \rceil - 2^{\lceil \log k \rceil} + k$$

según queríamos demostrar.

Por tanto, tenemos:

$$\begin{aligned} A_A(N) &\geq \frac{1}{N!} LCE_{min}(N!) = \frac{1}{N!} (N! \lceil \log N! \rceil - 2^{\lceil \log N! \rceil} + N!) \\ &= \lceil \log N! \rceil + 1 + \frac{2^{\lceil \log N! \rceil}}{N!} = \Omega(N \log N) \end{aligned}$$

ya que

$$\frac{2^{\lceil \log N! \rceil}}{N!} \leq \frac{2^{\log N! + 1}}{N!} = \frac{2 \cdot 2^{\log N!}}{N!} = \frac{2N!}{N!} = O(1)$$

por tanto se tiene:

$$A_A(N) = \Omega(N \log N). \quad (2.60)$$

Por tanto, Mergesort, Quicksort y Heapsort son óptimos en  $\xi$  para el caso medio.

#### 2.5.4. Radixsort \*

Hemos visto que mediante comparación de claves no es posible tener un algoritmo tal que  $A_A(N) = O(N)$ . Sin embargo podría existir otro tipo de operación básica que sí permita ordenar en tiempo  $O(N)$ . Vamos a explorar la idea de comparar trozos de claves en lugar de comparar claves completas.

Con esta idea dada una tabla  $T$  a ordenar, cada elemento  $T[i]$ ,  $i = 1, \dots, N$  será una cadena de longitud fija  $k$  de símbolos de un alfabeto  $\Sigma$ , es decir:

$$T[i] = [d_1, d_2, \dots, d_k] \quad \text{cond } d_j \in \Sigma \quad 1 \leq j \leq k$$

El pseudocódigo general de Radixsort es el siguiente:

```
RadixSort(tabla T, ind P, ind U, int k, int M)
para j de 1 a M : // Bucle 1
  InicializarQ(Qj);
Inicializar(Q);
```

```

para i de P a U :           // Bucle 2
  addQ(T[i],Q);
para j de k a 1 :         // Bucle 3
  mientras Q no vacia :   // Bucle 4
    extQ(t,Q);
    y=pos(t,k);
    addQ(t,Qy);
  para j de 1 a M :       // Bucle 5
    mientras Qj no vacia : // Bucle 6
      extQ(t,Qj);
      addQ(t,Q);
para i de P a U :       // Bucle 7
  extQ([1],Qj);

```

Como se ve, Radixsort necesita crear  $|\Sigma|$  colas auxiliares. El algoritmo realiza exactamente  $k$  iteraciones; en cada iteración consideramos un símbolo de las claves  $T[i]$  **de derecha a izquierda**.

La función **pos(t,k)** devuelve el índice del elemento  $t$  en el alfabeto  $\Sigma$ . Por ejemplo si  $\Sigma = \{0, 1, 2\}$  y  $t = (1201)$ ,  $pos(t, 3)$ , devolvería 1, ya que el tercer carácter de  $t$  es un 0, que ocupa la primera posición en el alfabeto.

**Ejemplo** Consideramos la tabla

$$T = [1021 \ 2100 \ 0121 \ 2121 \ 1212 \ 2111], \ \Sigma = \{0, 1, 2\}.$$

En cada iteración extraemos los elementos de arriba a abajo y de izquierda a derecha y los ubicamos en la correspondiente cola.

Iteración 1.

$Q_1$	2100
$Q_2$	1021 0121 2121 2111
$Q_3$	1212

Iteración 2.

$Q_1$	2100
$Q_2$	2111 1212
$Q_3$	1021 0121 2121

Iteración 3.

Iteración 4.

Ahora extraemos los elementos  $T = [0121 \ 1021 \ 1212 \ 2100 \ 2111 \ 2121]$  y la tabla queda ordenada.

$Q_1$	1021
$Q_2$	2100 2111 0121 2121
$Q_3$	1212

$Q_1$	0121
$Q_2$	1021 1212
$Q_3$	2100 2111 2121

No es fácil decidir cual es la operación básica de Radixsort. Podría considerarse, por ejemplo, algún tipo de comparación de dígitos dentro de la rutina  $\mathbf{pos}(\mathbf{t}, \mathbf{k})$ . Sin embargo ya vamos viendo que no es estrictamente necesario determinar cuál es la operación básica, pues el rendimiento de un algoritmo lo va a determinar la profundidad de los bucles anidados y la longitud de estos bucles. Lo que vamos a hacer es un análisis de los bucles del algoritmo. Denominaremos  $n_i$  al coste del bucle  $i$ . Suponemos que  $M = |\Sigma|$  y  $k = \text{longitud de cada clave } T[i]$ . Consideraremos también que la inicialización de colas tiene un coste constante, al igual que añadir o eliminar un elemento de una cola.

- Bucle 1:  $n_1 = M \cdot \underbrace{C(\text{inicializar } Q)}_{\Theta(1)} = \Theta(M)$
- Bucle 2 y Bucle 7:  $n_2 = n_7 = N \cdot \underbrace{C(\text{añadir } Q)}_{\Theta(1)} = \Theta(N)$
- Bucle 4:  $n_4 = \Theta(N)$
- Bucle 5:  $n_5 = \Theta(N)$ . El bucle 6 va repartiendo los  $N$  elementos en las colas  $Q_j$ .
- Bucle 3:  $n_3 = k \cdot \Theta(N) = \Theta(kN)$

Juntando el coste de todos los bucles se tiene que el coste total es  $\Theta(M + Nk)$ . Es razonable esperar que  $N \gg k, M$ , con lo que tenemos

$$n_{RS}(T, P, U, k, M) = \Theta(kN) = \Theta(N) \text{ si } k \text{ es constante}$$

El problema de la fórmula anterior es que  $N = U - P + 1$  y  $k$  pueden no ser independientes. Por ejemplo si  $\Sigma = \{0, 1, 2, \dots, 9\}$ ,  $M = 10$  y  $k = 4$  tenemos que el máximo numero de elementos distintos es  $N = M^k = 10000$ . Es decir  $\log N \leq k$  si queremos que todos los elementos de  $T$  sean distintos. Pero si  $k \geq \log N$  entonces

$$n_{RS}(T) = \Theta(kN) \ll N \log N,$$

con lo que ya no tenemos un coste lineal, y de hecho es un poco peor que Heapsort.

Es decir, Radixsort será mejor que Heapsort, Quicksort o Mergesor solamente si  $T$  tiene muchos elementos repetidos, de tal forma que  $k < \log N$ , es decir  $M^k \ll N$ .



# Capítulo 3

## Métodos de búsqueda

### 3.1. Caso medio de la búsqueda binaria

En el capítulo 1 ya vimos algunos algoritmos de ordenación y obtuvimos algunos resultados previos, por ejemplo:

- Búsqueda lineal

- $W_{BLin}(N) = N$  con la comparación de claves como operación básica.
- $A_{BLin}^e(N) = \sum_1^n \underbrace{n_{BLin}(T[i])}_i \cdot prob(k == T[i])$  que generalmente se reduce a estimar un cociente  $\approx \frac{S_N}{C_N}$

- Búsqueda binaria

- $W_{BBin}(N) = \lceil \log N \rceil = \log N + O(1) = A_{BBin}^f(N)$  donde  $A_{BBin}^f(N)$  es el coste de las búsquedas fallidas en tablas de tamaño  $N$  pues en ese caso siempre se hace el número máximo  $\lceil \log N \rceil$  de iteraciones.

Nos faltaría calcular  $A_{BBin}^e(N)$ . Para ello consideremos a modo de ejemplo el caso particular  $N = 7$ , esto es, la tabla  $T = [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7]$ . Se tiene entonces, suponiendo equiprobabilidad:

$$A_{BBin}^e(7) = \frac{1}{7} \sum_{i=1}^7 n_{BBin}(i) = \frac{1}{7} \left( \underbrace{1}_4 + \underbrace{2}_2 + \underbrace{2}_6 + \underbrace{3}_1 + \underbrace{3}_3 + \underbrace{3}_5 + \underbrace{3}_7 \right) =$$
$$\frac{1}{7} (1 + 2 \cdot 2 + 3 \cdot 4) = \frac{1}{7} (1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2) = \frac{1}{7} \sum_{i=1}^3 i2^{i-1} \text{ con } 7 = 2^3 - 1$$

En general, si  $N = 2^k - 1$  se tiene

$$A_{BBin}^e(N) = \frac{1}{N} \sum_{i=1}^k i2^{i-1} = \frac{1}{N}(k2^k - 2^k + 1)$$

ya que

$$\begin{aligned} \sum_{i=1}^N ix^{i-1} &= \frac{d}{dx} \left( \sum_{i=1}^N x^i \right) = \frac{d}{dx} \left( \frac{x^{N+1} - x}{x - 1} \right) \\ &= \frac{kx^{k+1} - (k+1)x^k + 1}{(x-1)^2} \\ &= \frac{k2^{k+1} - (k+1)2^k + 1}{(2-1)^2} \\ &\stackrel{\text{para } x=2}{=} 2k2^k - k2^k - 2^k + 1 = k2^k - 2^k + 1 \end{aligned}$$

Si  $N = 2^k - 1$  entonces  $k \approx \log N$  con lo que se obtiene:

$$A_{BBin}^e(N) = \frac{1}{N}(k2^k - 2^k + 1) \approx \frac{1}{N}(\log N 2^{\log N} - 2^{\log N} + 1) = \log N - 1 + \frac{1}{N}$$

y por tanto:

$$A_{BBin}^e(N) = \log N + O(1) \quad (3.1)$$

que también vale para un  $N$  general.

Hasta ahora hemos obtenido las siguientes cotas para algoritmos por comparación de clave:

	Ordenación	Búsqueda
Locales (malos)	$N^2$	$N$
Otros (buenos)	$N \log N$	$\log N$
Cota inferior	$N \log N$	$\log N$ ?

### 3.2. Cotas inferiores para algoritmos de búsqueda mediante comparación de clave

Viendo la tabla anterior parece razonable pensar que la cota inferior de los algoritmos de búsqueda mediante comparación de clave debería ser  $\log N$ . Al

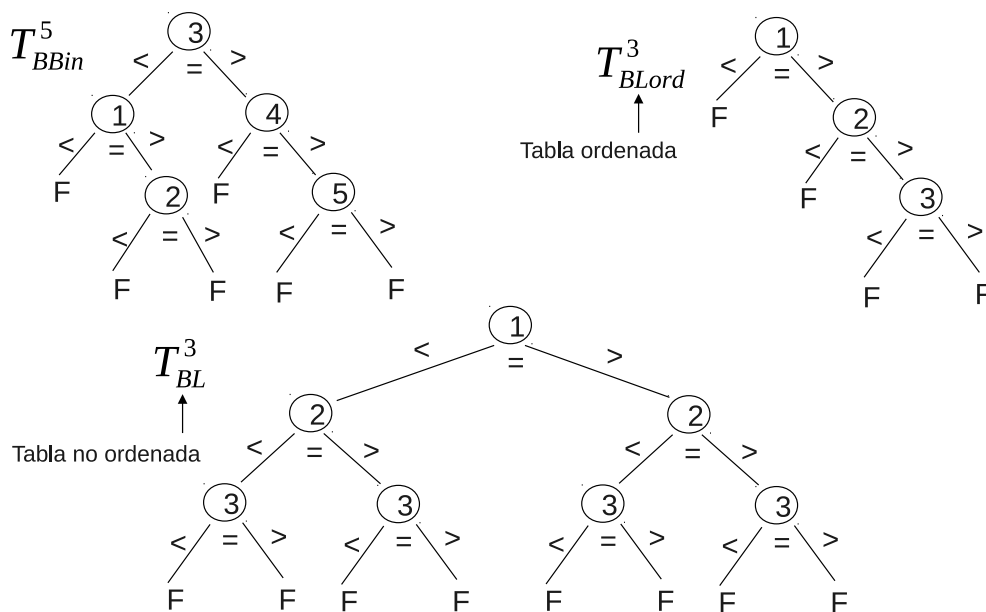


igual que hicimos en la sección 2.5, vamos a emplear árboles de decisión para deducir este resultado.

Sea  $\mathcal{B} = \{ \text{Familia de algoritmos de búsqueda por comparación de clave} \}$   
 Si  $A \in \mathcal{B}$  su árbol de decisión  $T_A^N$  cumple que:

1. Contiene nodos de la forma  $i$  que indica la comparación de clave entre el elemento  $i$ -ésimo de la tabla y la clave  $k$ .
2. Si  $k$  coincide con el elemento  $i$ -ésimo ( $T[i] == k$ ) entonces la búsqueda de la clave  $k$  termina en el nodo  $i$ .
3. El subárbol izquierdo del nodo  $i$  en  $T_A^N$  contiene el trabajo subsiguiente en comparaciones de clave que realiza el algoritmo  $A$  si  $k < T[i]$ .
4. El subárbol derecho del nodo  $i$  en  $T_A^N$  contiene el trabajo subsiguiente que realiza el algoritmo  $A$  si  $k > T[i]$ .
5. Las hojas  $H_\sigma$  en  $T_A^N$  recogen la evolución de las búsquedas fallidas.

En el siguiente dibujo se pueden ver los árboles de decisión de la búsqueda binaria en una tabla de tamaño  $N = 5$ , la búsqueda lineal en una tabla ordenada de tamaño  $N = 3$  y la búsqueda lineal en una tabla de tamaño  $N = 3$




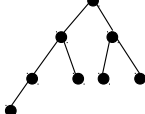
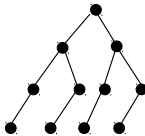


Observamos que  $T_A^N$  es un árbol binario con **al menos**  $N$  nodos internos. Dado que

$$W_A(N) \geq prof_{min}(N)$$

donde  $prof_{min}(N)$  es la profundidad mínima de un árbol binario con al menos  $N$  nodos, vamos a estimar esta cantidad.

Al igual que en el caso de los árboles de decisión para los algoritmos de ordenación vamos a ver como son estos árboles binarios con  $N$  nodos internos con la menor profundidad posible.

N	T	Prof <sub>min</sub> (N)
1		1
2		2
3		2
4		3
7		3

Se puede ver que  $prof_{min}(N) = \lfloor \log N \rfloor + 1$ , y por tanto:

$$W_A(N) \geq prof_{min}(N) = \lfloor \log N \rfloor + 1 \Rightarrow W_A(N) = \Omega(N) \quad \forall A \in \mathcal{B} \quad (3.2)$$

De forma análoga se puede ver también que

$$A_A(N) = \Omega(\log N) \quad (3.3)$$

y por tanto la búsqueda binaria es óptima en  $\mathcal{B}$  para el caso peor y medio.

### 3.3. Tipo abstracto de datos diccionario

En la práctica Buscar no es una operación que se efectúe aislada sino que suele ir acompañada de la inserción de los datos en la estructura de búsqueda (tras una búsqueda fallida) y de su eventual borrado (tras una búsqueda con éxito). Esto lleva al concepto de **diccionario** como un conjunto ordenado de datos junto con las siguientes primitivas:

1. **pos Buscar(clave k, dicc D)**

Devuelve la posición de la clave k en el diccionario D o un código de error ERR si k no está en D.

2. **status Insertar (clave k, dicc D)**

Inserta la clave k en el diccionario D, devuelve la posición de la clave k o o un código de error ERR si k no se pudo incorporar a D.

3. **void Borrar (clave k, dicc D)**

Elimina la clave k en el diccionario D

En un diccionario podemos usar diferentes estructuras de datos. El rendimiento del diccionario dependerá de la estructura de datos elegida.

Una primera idea podría ser una tabla ordenada, ya que por la sección anterior vimos que la búsqueda en tablas ordenadas tenía un rendimiento óptimo, por tanto se tendría  $n_{\text{Buscar}}(k, D) = O(\log N)$ .

Sin embargo, insertar resulta una operación costosa, ya que cada vez que se inserta un nuevo elemento es necesario mantener el orden de la tabla, y para ello hay que desplazar una posición a la derecha los elementos mayores que el elemento que se está insertando. Por tanto, si  $N = |D|$  entonces  $n_{\text{Insertar}}(k, D) = \Theta(N)$  ya que el caso peor supondrá desplazar  $N$  elementos y el caso medio  $\frac{N}{2}$  elementos, lo cual es un mal rendimiento.

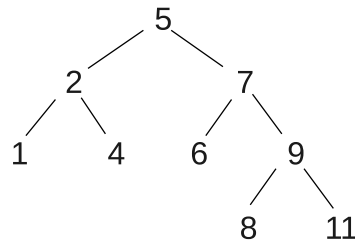
Otra opción sería usar un árbol binario de búsqueda.

**Definición:**  $T$  es un árbol binario de búsqueda si  $T$  es un árbol binario y para todo nodo  $T' \in T$  se cumple

$$\text{info}(T'') < \text{info}(T') < \text{info}(T''') \quad (3.4)$$

para todo nodo  $T''$  a la izquierda de  $T'$  y todo nodo  $T'''$  a la derecha de  $T'$ .

Es decir, todos los nodos que se encuentran en el subárbol izquierdo de cada nodo  $T'$  ( $\text{izq}(T')$ ) tienen un valor menor que el nodo  $T'$  y todos los nodos que se encuentran en el subárbol derecho de  $T'$  ( $\text{der}(T')$ ) tienen un valor mayor que el nodo  $T'$ . En el dibujo inferior se puede ver un ejemplo de árbol binario de búsqueda



Con esta estructura de datos el pseudocódigo de la primitiva **Buscar** es el siguiente:

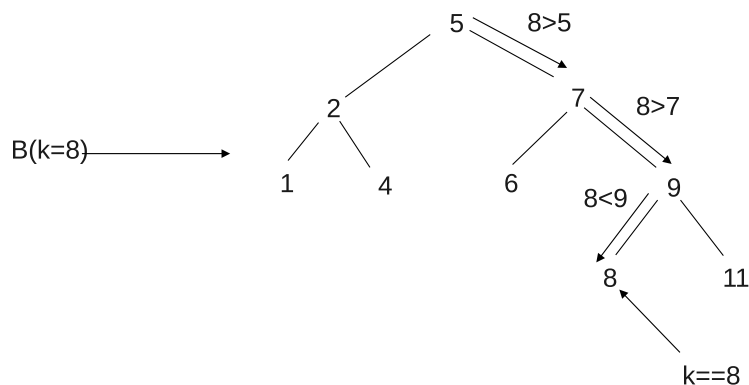
```

AB Buscar (clave k, AB T)
  si T==NULL : return NULL;
  si info(T)==k : return T;
  si k<info(T)
    return Buscar(k,izq(T));
  si k>info(T)
    return Buscar(k,der(T));
  
```

Se puede observar que usando la cdc como operación básica

$$n_{\text{Buscar}}(k, T) = O(\text{prof}(T)) \quad (3.5)$$

En el siguiente dibujo se puede ver el funcionamiento de la rutina **Buscar** para la clave  $k = 8$



Para la rutina **Insertar** tenemos el siguiente pseudocódigo:

```

status Insertar (clave k, AB T)
    T'=BuscarUltimo(k,T);
    T''=GetNodo();
    si T''==NULL : return ERR;
    info(T'')=k;
    si k<info(T')
        izq(T')=T''
    else
        der(T')=T'';
    return OK:

```

**Insertar** utiliza la rutina **BuscarUltimo**, que devuelve el último nodo visitado por **Buscar** antes de devolver *NULL*; por tanto, el nodo  $T'$  devuelto por **BuscarUltimo** debe tener alguno de sus hijos a **NULL**. La rutina **BuscarUltimo** es similar a la rutina **Buscar** con búsqueda fallida y devolviendo el último nodo visitado, en lugar de **NULL**.

```

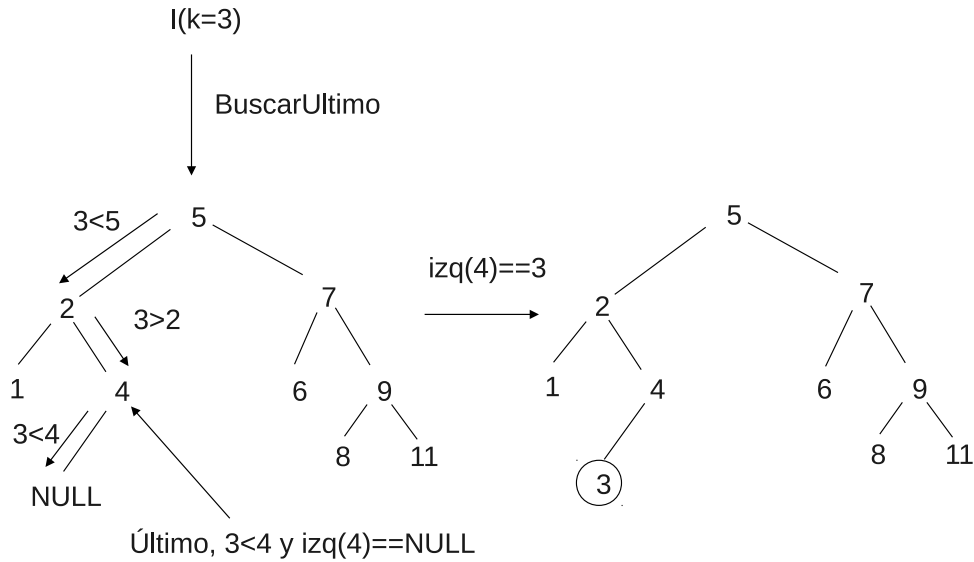
AB BuscarUltimo(clave k AB T)
    si k<info(T) y izq(T) !=NULL
        return BuscarUltimo(k,izq(T));
    si k>info(T) y der(T) !=NULL
        return BuscarUltimo(k,der(T));
    return T

```

En cuanto a rendimiento se tiene que:

$$n_{\text{Insertar}}(k, T) = n_{\text{BuscarUltimo}}(k, T) + 1 \Rightarrow n_{\text{Insertar}}(k, T) = O(\text{prof}(T)) \quad (3.6)$$

En el siguiente dibujo se puede ver el funcionamiento de **Insertar** para la clave  $k = 3$ .



El siguiente pseudocódigo corresponde a la primitiva **Borrar**

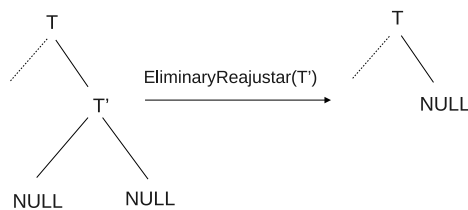
```
void Borrar (clave k, AB T)
  T'=Buscar(k,T);
  si T' !=NULL
    EliminaryReajustar(T',T);
```

Por tanto tenemos que

$$n_{\text{Borrar}}(k, T) = n_{\text{Buscar}}(k, T) + n_{\text{EliminaryReajustar}}(T', T) \quad (3.7)$$

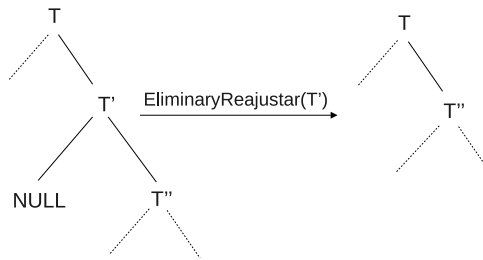
En la rutina **EliminaryReajustar** hay tres posibles casos cada uno con un rendimiento diferente:

- **Caso 1:** El nodo  $T'$  no tiene hijos. En este caso basta realizar **free(T')** y asignar a **NULL** el puntero correspondiente en el padre de  $T'$ . En el dibujo inferior se puede ver el proceso de liberar un nodo  $T'$  que no tiene hijos.



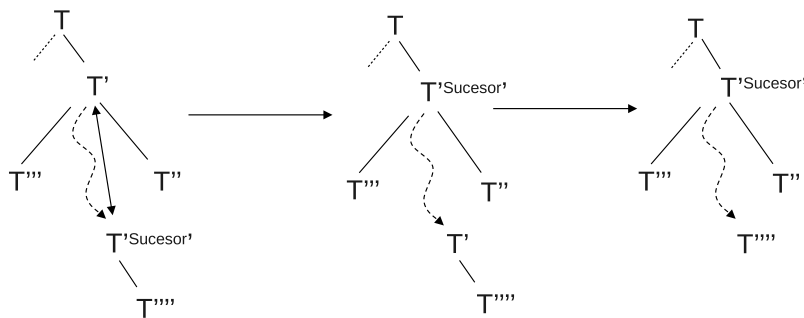
En este caso se tiene que  $n_{\text{EliminaryReajustar}}(T', T) = O(1)$ .

- Caso 2:** El nodo  $T'$  tiene un solo hijo. En este caso el puntero del padre de  $T'$  que apuntaba a  $T'$  se hace apuntar al único hijo de  $T'$ , posteriormente se libera  $T'$  ( $\text{free}(T')$ ). En el dibujo inferior se puede ver el proceso de liberar un nodo  $T'$  con un único hijo.



En este caso se tiene que  $n_{\text{EliminaryReajustar}}(T', T) = O(1)$ .

- Caso 3:** En nodo  $T'$  tiene dos hijos. En este caso en el campo info del nodo  $T'$  se sitúa la clave contenida en el nodo que contiene al **sucesor**, esto es, el elemento siguiente a  $\text{info}(T')$  en la tabla ordenada, y después se elimina el nodo  $T''$  que contiene al sucesor según el caso 1 o 2. En el dibujo inferior se puede ver el proceso de reajustar un nodo  $T'$  con dos hijos.

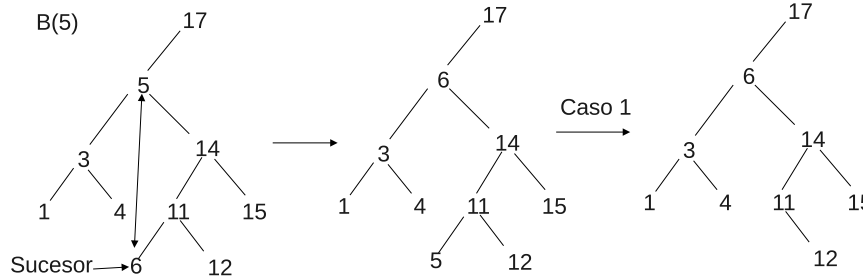


En el siguiente ejemplo se puede ver cómo se elimina la clave  $k = 5$ . Recorriendo el árbol en orden medio vemos que el sucesor del 5 es el 6.

Vamos ahora a nombrar a un nodo por su valor, es decir, el nodo  $k$  es aquel cuyo valor es  $k$ .

**Observación:** Si  $k'$  es el sucesor de  $k$  en un árbol binario de búsqueda, entonces  $\text{izq}(k') = \text{NULL}$ .

Para demostrar esto, supongamos que existe  $k'' = \text{izq}(k')$ , entonces se tiene:



- $k'' < k'$  ya que  $k''$  está a la izquierda de  $k'$  y estamos en un árbol binario de búsqueda.
- $k'' > k$  ya que al ser  $k'$  el sucesor de  $k$  tiene que ocurrir que  $k'$  así como todos sus descendientes sean mayores que  $k$ .

Por tanto tenemos  $k < k'' < k'$ , lo cual es una contradicción con que  $k'$  sea el sucesor de  $k$ . Por tanto, el sucesor de un nodo  $k$  tiene a lo sumo un hijo derecho y es fácilmente eliminable (casos 1 o 2).

En este caso se tiene que

$$\begin{aligned} n_{\text{EliminaryReajustar}}(T', T) &= n_{\text{BuscarSucesor}}(T', T) + n_{\text{ReajustarPunteros}}(T', T) \\ &= O(\text{prof}(T)) + O(1) = O(\text{prof}(T)). \end{aligned}$$

El siguiente pseudocódigo implementa la rutina **BuscarSucesor** que devuelve el sucesor de un nodo  $T'$ .

```

AB BuscarSucesor(AB T')
  T'' = der(T');
  mientras izq(T'') != NULL
    T'' = izq(T'');
  return T''

```

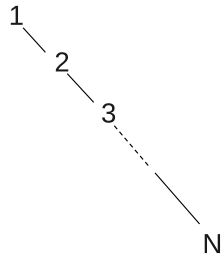
Por tanto, a la vista de que el coste máximo visto para **EliminaryReajustar** es  $n_{\text{EliminaryReajustar}}(T', T) = O(\text{prof}(T))$  se tiene que

$$n_{\text{Borrar}}(k, T) = \underbrace{O(\text{prof}(T))}_{\text{Buscar}} + \underbrace{O(\text{prof}(T))}_{\text{EliminaryReajustar}} \Rightarrow n_{\text{Borrar}}(k, T) = O(\text{prof}(T)) \quad (3.8)$$

El resultado anterior nos permite afirmar que un árbol binario de búsqueda será una estructura de datos eficaz para diccionarios con  $N$  elementos siempre y cuando  $\text{prof}(T) = \Theta(\log N)$ .



Sin embargo, no cualquier árbol binario de búsqueda tiene profundidad  $\Theta(\log N)$ . Véase, por ejemplo, el siguiente árbol binario de búsqueda



Con este ejemplo vemos que

$$W_{\text{Buscar}}(N) = N \quad (3.9)$$

lo cual es un mal rendimiento. Luego veremos qué se puede hacer en este caso pero ahora vamos a calcular ahora el caso medio  $A_{\text{Buscar}}^e(N)$  trabajando con ABdBs  $T_\sigma$  asociados a permutaciones  $\sigma$ . El cálculo del coste medio tiene dos componentes,

- i. El coste promedio de buscar todas y cada una de las claves  $\sigma(i)$ ,  $i = 1, \dots, N$  en un árbol binario de búsqueda  $T_\sigma$  asociado a una tabla  $\sigma \in \Sigma_N$ .
- ii. El promedio de cada una las cantidades anteriores entre todas las permutaciones  $\sigma \in \Sigma_N$ .

Es decir:

$$A_{\text{Buscar}}^e(N) = \frac{1}{N!} \underbrace{\sum_{\sigma \in \Sigma_N} \underbrace{A_{\text{Buscar}}^e(T_\sigma)}_{(i)}}_{(ii)}$$

donde  $A_{\text{Buscar}}^e(T_\sigma)$  es el coste promedio de buscar todas y cada una de las claves  $\sigma(i)$ ,  $i = 1, \dots, N$  en el árbol binario de búsqueda  $T_\sigma$  asociado a la permutación  $\sigma \in \Sigma_N$ . Por tanto:

$$\begin{aligned} A_{\text{Buscar}}^e(T_\sigma) &= \frac{1}{N} \sum_{i=1}^N n_{\text{Buscar}}(\sigma(i), T_\sigma) = \frac{1}{N} \sum_{i=1}^N (\text{prof}(\sigma(i)) + 1) \\ &= 1 + \frac{1}{N} \sum_{i=1}^N \text{prof}(\sigma(i)) = 1 + \frac{1}{N} n_{\text{Crear}}(T_\sigma), \end{aligned}$$

pues  $prof(\sigma(i)) = n_{Insertar}(\sigma(i))$ . Por tanto:

$$\begin{aligned} A_{\text{Buscar}}^e(N) &= \frac{1}{N!} \sum_{\sigma \in \Sigma_N} \left( 1 + \frac{1}{N} n_{\text{Crear}}(T_\sigma) \right) \\ &= 1 + \frac{1}{N} \left( \frac{1}{N!} \sum_{\sigma \in \Sigma_N} n_{\text{Crear}}(T_\sigma) \right) \end{aligned} \quad (3.10)$$

$$\Rightarrow A_{\text{Buscar}}^e(N) = 1 + \frac{1}{N} A_{\text{Crear}}(N) \quad (3.11)$$

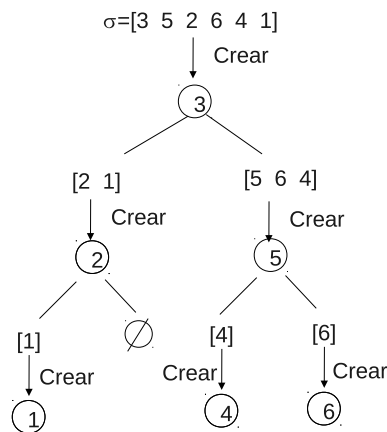
Nos faltaría calcular  $A_{\text{Crear}}(N)$ , Para ello, en lugar de realizar el calculo introduciendo los elementos de la permutación  $\sigma$  en el árbol de uno en uno y sumando el coste correspondiente a cada inserción, consideremos el siguiente pseudocódigo de creación de árboles binarios de búsqueda a partir de una permutación  $\sigma$ .

```

AB Crear (tabla S)
  T=IniAB(S)
  InsAB(S(1),T)
  Repartir(S, Si, Sd)
  Ti=Crear(Si)
  Td=Crear(Sd)
  izq(T)=Ti
  der(T)=Td
  return T

```

Por ejemplo, la rutina **Crear** funcionaría según el dibujo siguiente para la permutación  $\sigma = [3 \ 5 \ 2 \ 6 \ 4 \ 1]$



Como el funcionamiento de `Repartir` es similar al de la rutina `Partir` de Quicksort, observamos que el pseudocódigo anterior sigue una ecuación en recurrencia igual a la de Quicksort, es decir

$$n_{Crear}(\sigma) = N - 1 + n_{Crear}(\sigma_i) + n_{Crear}(\sigma_d)$$

y por tanto, sus casos medio (y peor) son iguales que los de Quicksort; esto es, usando (2.43) se tiene que:

$$A_{Buscar}^e(N) = 1 + \frac{1}{N} A_{Crear}(N) = 2N \log N + O(N)$$

y por tanto

$$A_{Buscar}^e(N) = \Theta(\log N) \tag{3.12}$$

que es un buen rendimiento.

### 3.4. Árboles AVL (Adelson-Velskii-Landis)

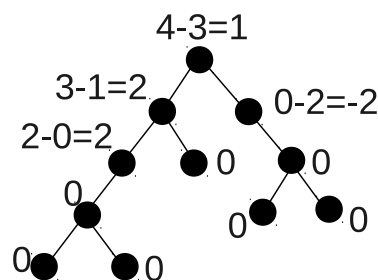
Nos gustaría que no sólo el caso medio de las búsquedas en diccionarios fuese  $\Theta(N)$  sino que también lo fuese el caso peor. Para ello es necesario modificar la estructura de datos que empleamos. La idea es buscar un método de construcción de árboles binarios de búsqueda tal que  $\forall \sigma \in \Sigma_N$  se construya un árbol binario de búsqueda tal que  $prof(T_\sigma) = \Theta(\log N)$ .

Dado un árbol binario definimos el **factor de equilibrio** de un nodo  $T$  como:

$$FE(T) = prof(T_i) - prof(T_d) \tag{3.13}$$

donde  $T_i$  es el subárbol izquierdo del nodo  $T$  y  $T_d$  es el subárbol derecho. Consideraremos que el árbol vacío tiene profundidad 0.

En el dibujo inferior se puede ver un ejemplo en el que se calcula el factor de equilibrio de todos los nodos de un árbol binario.



Definimos como árbol **AVL** a un árbol binario de búsqueda  $T$  tal que para todo nodo  $T'$  de  $T$  se verifica que el factor de equilibrio toma solo los valores  $0, 1$  ó  $-1$

$$|FE(T')| \leq 1 \quad (3.14)$$

Vamos a demostrar que si  $T$  es un AVL, entonces

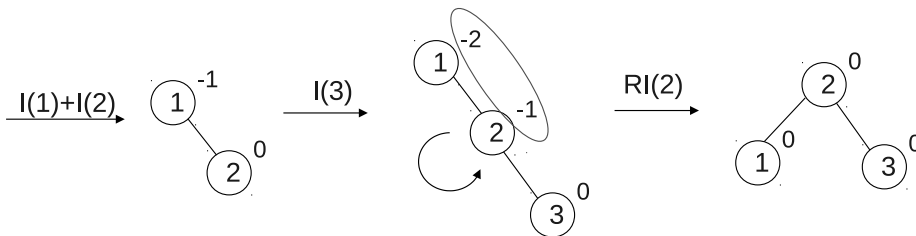
$$prof(T) = \Theta(\log N) \quad (3.15)$$

### 3.4.1. Construcción de AVLs

La construcción de un AVL se realiza en dos pasos

1. Se realiza una inserción normal en un árbol binario de búsqueda.
2. Si es necesario se corrigen posibles desequilibrios que puedan aparecer tras la inserción.

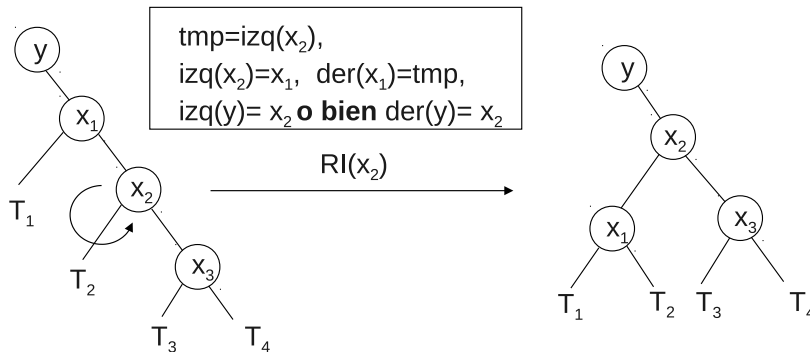
Veamos la construcción de AVLs mediante un ejemplo. Vamos a construir un AVL correspondiente a la tabla:  $T = [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 15 \ 14 \ 13 \ 12 \ 11 \ 10 \ 9 \ 8]$



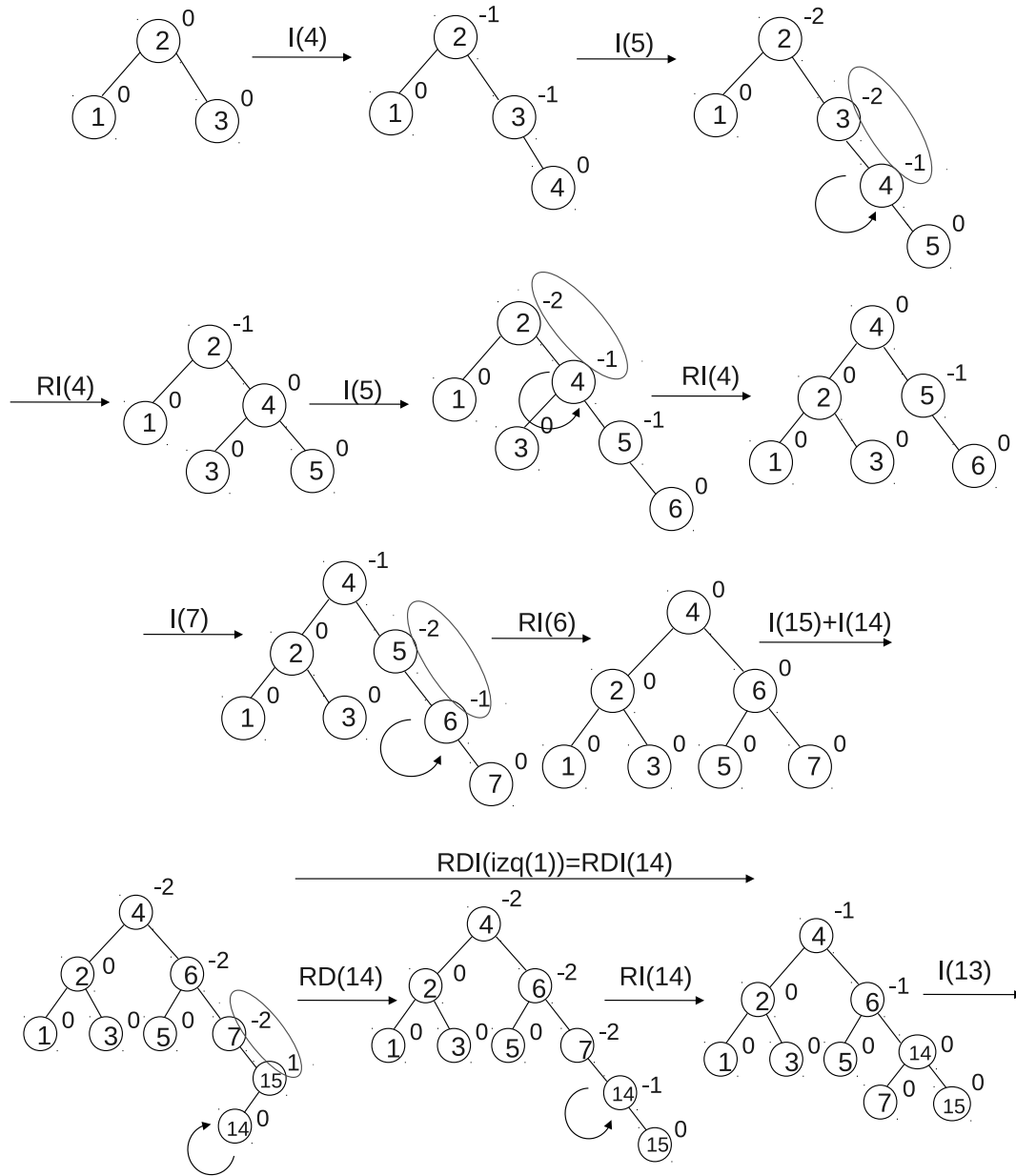
Tras insertar el elemento 3 aparece un nodo con un factor de equilibrio  $-2$ , para solucionar ese desequilibrio nos tenemos que fijar en la pareja de nodos con desequilibrios  $(-2, -1)$ , es decir los nodos 1 y 2. En caso que aparezcan varios nodos con factor de equilibrio 2 o  $-2$ , siempre realizaremos la operación sobre la pareja de nodos que tenga el desequilibrio de valor 2 a mayor profundidad en el árbol.

La última operación realizada se llama **rotación izquierda en el -1**, es decir, sobre el elemento 2, ya que este es el que tiene el desequilibrio con valor  $-1$ . La rotación a la izquierda sobre el nodo  $x_2$  corresponde con la siguiente reasignación de punteros:

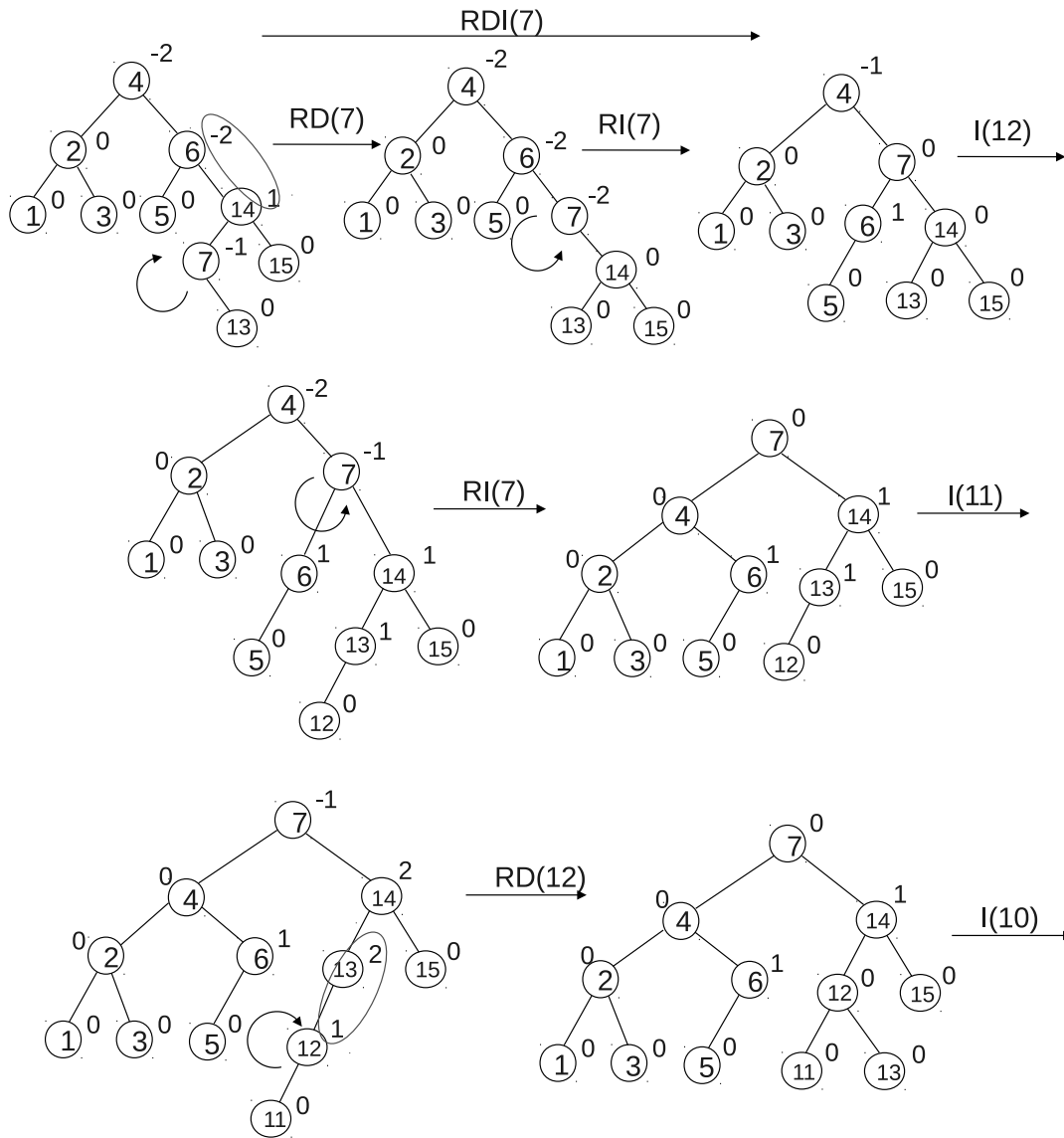
Ahora volvemos a tener un árbol sin desequilibrios, con lo cual seguimos insertando elementos de forma normal



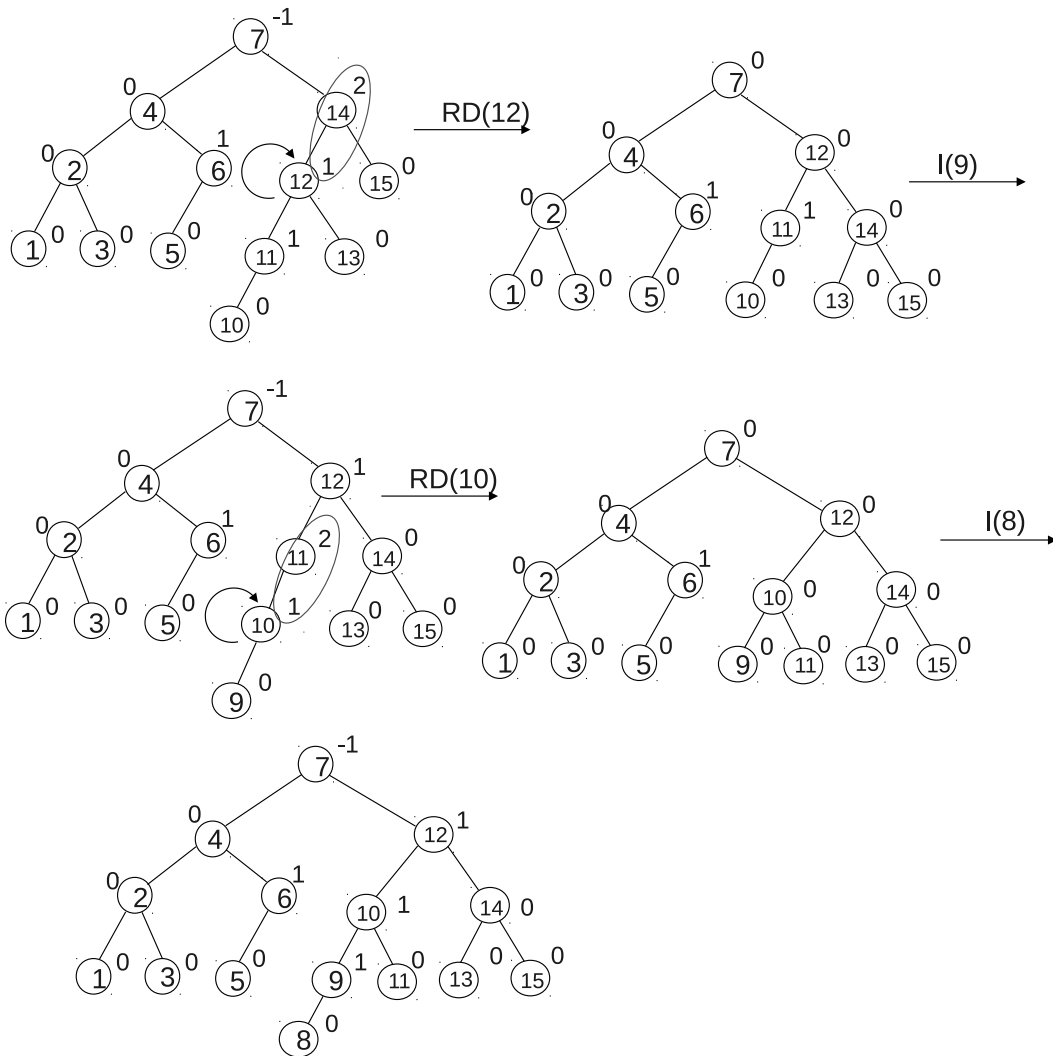
Tras la inserción del elemento 14 aparece un nuevo tipo de desequilibrio, el desequilibrio  $(-2, 1)$ . Este desequilibrio se puede deshacer mediante dos rotaciones consecutivas, primero una **rotación a la derecha en el elemento a la izquierda del 1**, es decir, en este caso el 14, seguido de una **rotación a la derecha** en ese mismo elemento. A esta operación se la denomina **rotación derecha izquierda en la izquierda del 1**.



Tras la inserción del elemento 11 aparece un desequilibrio del tipo (2, 1). Este desequilibrio es muy similar al del tipo  $-2, -1$ . En este caso el desequilibrio se resuelve mediante una **rotación a la derecha en el 1**, que en este caso es el elemento 12. Una vez resuelto el desequilibrio seguimos insertando elementos.



con lo que concluye la construcción del AVL.



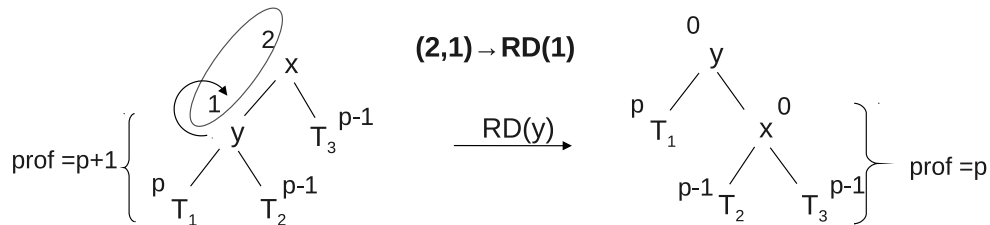
La siguiente tabla resume las rotaciones a realizar para resolver los diferentes tipos de desequilibrios que hemos encontrado.

Para ver que efectivamente estas rotaciones deshacen los desequilibrios es necesario estudiar cada uno de los casos. Por ejemplo, el caso del desequilibrio (2, 1) se resuelve mediante una rotación a la derecha que, como vimos corresponde a la siguiente reasignación de punteros.

Para ver que la rotación funciona fijamos la profundidad de uno de los subárboles a  $p$ , por ejemplo,  $prof(T_1) = p$ . Debido a los valores de los desequilibrios, esto nos permite calcular la profundidad de todos los demás subárboles. Por ejemplo,



Desequilibrio	Rotación
$(-2, -1)$	Rotación a la izquierda en el -1 (hijo izq. de -1 pasa a hijo der. de -2)
$(2, 1)$	Rotación a la derecha en el 1 (hijo der. de 1 pasa a hijo izq. de 2)
$(-2, 1)$	Rotación derecha izquierda en la izquierda del 1 $RD(izq(1))+RI(izq(1))$
$(2, -1)$	Rotación izquierda derecha en la derecha del -1 $RI(der(-1))+RD(der(-1))$



dado que el factor de equilibrio del nodo  $y$  es uno, y dado que

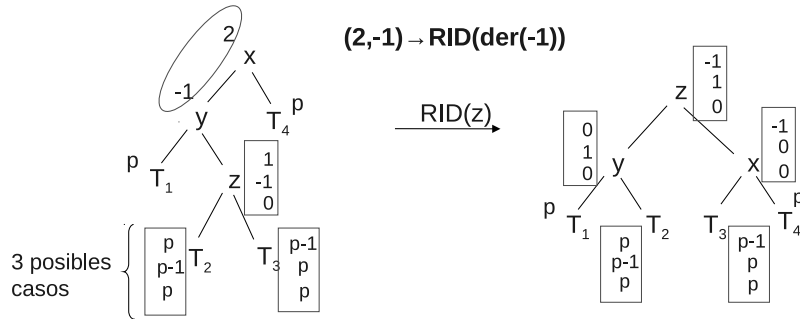
$$\begin{aligned}
 1 &= FE(y) = prof(izq(y)) - prof(der(y)) = prof(T_1) - prof(T_2) \\
 &= p - prof(T_2) \\
 \Rightarrow & prof(T_2) = p - 1
 \end{aligned}$$

Por un argumento similar se puede ver que  $prof(T_3) = p$ .

Una vez reasignados los punteros, y dado que la estructura de los subárboles  $T_1, T_2$  y  $T_3$  no ha cambiado, estos tienen mantienen la profundidad. Por tanto ahora se tiene

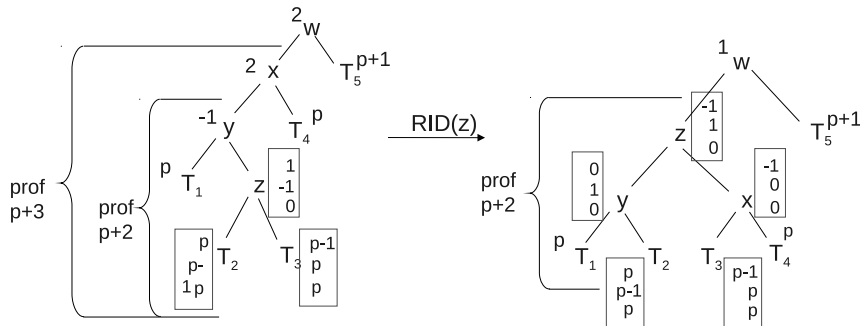
$$\begin{aligned}
 FE(x) &= prof(izq(x)) - prof(der(x)) = prof(T_2) - prof(T_3) = (p - 1) - p = -1 \\
 FE(y) &= prof(izq(y)) - prof(der(y)) = prof(T_1) - prof(x) = p - p = 0
 \end{aligned}$$

Por ejemplo, en el caso del desequilibrio  $(2, -1)$  aplicamos una rotación izquierda derecha a la derecha del  $-1$ , en este caso en el nodo  $z$ . Los nuevos factores de equilibrio resultan de la siguiente manera:



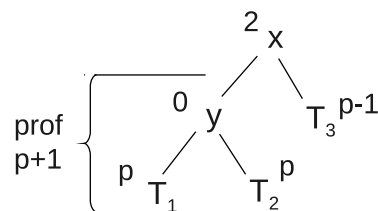
En este caso, de nuevo fijamos la profundidad de uno de los subárboles,  $prof(T_1) = p$ , sin embargo existen tres posibilidades en función del factor de equilibrio que tenga el nodo  $z$  ( $1, 0$  ó  $-1$ ). En el dibujo anterior se puede ver que en cualquiera de los tres casos, después de reasignar punteros el árbol queda equilibrado. El detalle del argumento se deja como ejercicio.

Al inicio de la sección indicamos que tras insertar un elemento podían aparecer varios nodos con factor de equilibrio  $2$  o  $-2$ , e indicamos que siempre había que tratar de resolver el desequilibrio en aquel nodo que estuviese a mayor profundidad dentro el árbol. Hemos visto que las rotaciones resuelven el desequilibrio en los nodos sobre los que realizamos la rotación. Sin embargo las rotaciones resuelven también aquellos desequilibrios que están por encima de los nodos sobre los que hemos realizado la rotación. Esto es debido a que la rotación reduce en  $1$  la profundidad del subárbol cuyo nodo raíz es el elemento con factor de equilibrio  $2$  o  $-2$  mas bajo (en el gráfico el nodo  $x$ ), y por tanto, esta reducción de profundidad afecta también a los nodos que están por encima de  $x$ . Este efecto se puede ver en el siguiente dibujo.

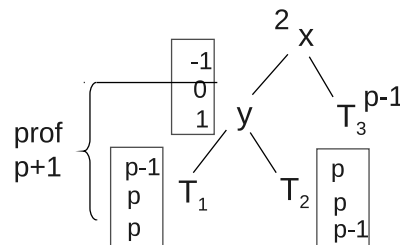


Vemos que el subárbol izquierdo del nodo  $w$  con factor de equilibrio 2 tiene profundidad  $p + 3$  antes de la rotación. Al tratar de resolver el desequilibrio del elemento mas bajo, en este caso  $x$ , vemos que el nuevo subárbol izquierdo del nodo  $w$  tiene profundidad  $p + 2$ , y por tanto el factor de equilibrio del nodo  $w$ , al no haberse modificado su subárbol derecho, disminuye una unidad y pasa a tomar el valor 1.

Hasta ahora hemos visto cuatro tipos de desequilibrios  $(2, 1)$ ,  $(-2, -1)$ ,  $(2, -1)$  y  $(-2, 1)$ , vamos a ver que no pueden existir otros tipos de desequilibrios, como por ejemplo el  $(2, 0)$  o el  $(-2, 0)$ . Es decir, en un AVL tras la inserción de un nuevo nodo, no puede aparecer un desequilibrio de la forma  $(2, 0)$ . Para ver que efectivamente no puede haber tal desequilibrio supongamos que tenemos un AVL y que tras la inserción de un nodo obtenemos un desequilibrio de la forma  $(2, 0)$ . En ese caso la forma del árbol tras la inserción debe ser la siguiente:



De nuevo fijamos  $prof(T_1) = p$ , con lo cual, dado que  $FE(y) = 0 \Rightarrow prof(T_2) = p$ , además se tiene  $prof(T_3) = p - 1$ . Hay que señalar que el nodo no pudo insertarse en el árbol  $T_3$ , ya que al insertar un nodo, o bien la profundidad del árbol se mantiene, o aumenta en uno, con lo cual, antes de la inserción, el nodo  $x$  debería haber tenido factor de equilibrio 2 o 3, lo cual no es posible ya que suponemos que el árbol era un AVL antes de la inserción. Con esta observación el aspecto del árbol antes de la inserción del nuevo nodo sería el siguiente:



Vemos que la inserción de un nuevo nodo, solo puede afectar a la profundidad de uno de los árboles  $T_1$  o  $T_2$ , y en cualquiera de los dos caso ésta aumenta, a lo sumo, una unidad. Eso implica que, (i) el factor del equilibrio del nodo  $y$  antes



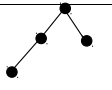

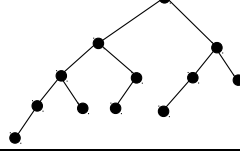
de la inserción era 1, 0 ó  $-1$  y (ii) que la profundidad del árbol cuya raíz es  $y$  no ha variado. Por tanto, el factor de equilibrio del nodo  $x$  ya era 2 **antes de la inserción del nuevo nodo**, lo cual es imposible si el árbol previo a la inserción del nodo es un AVL.

### 3.4.2. Profundidad de un AVL

Vamos a demostrar la siguiente proposición.

**Proposición.** Si  $T$  es un AVL con  $N$  nodos, entonces  $\text{prof}(T) = O(\log N)$ . Además dado que en cualquier árbol de  $N$  nodos se verifica  $\text{prof}(T) = \Omega(\log N)$ , tenemos que  $\text{prof}(T) = \Theta(\log N)$ .

De nuevo vamos a construir ejemplos de AVLs de profundidad  $p = 0, 1, 2, \dots$  y vamos a ver cual es el número mínimo de nodos que puede tener un AVL de esa profundidad. El resultado se puede ver en la siguiente tabla:

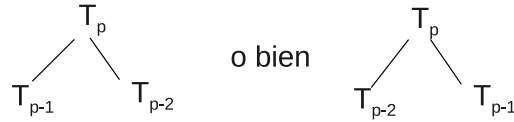
$p$	AVL	$n_p$	$n_{p+1}$	$F_{p+2}$
0		1	2	$F_2$
1		2	3	$F_3$
2		4	5	$F_4$
3		7	8	$F_5$
4		12	13	$F_6$
...			....	....

Aquí  $n_p$  es el número de nodos del AVL, y  $F_{p+2}$  es el  $(p+2)$ -ésimo número de Fibonacci ( $F_0 = F_1 = 1$ ,  $F_i = F_{i-1} + F_{i-2}$ ).

Parece que

$$F_{p+2} = n_p + 1$$

Podemos ver que la fórmula anterior es cierta fijándonos que la estructura de un AVL de profundidad  $p$  consiste en un nodo raíz del cual cuelgan un AVL de profundidad  $p - 1$  y un AVL de profundidad  $p - 2$ , es decir:



por tanto se tiene:

$$n_p = 1 + n_{p-1} + n_{p-2} \Rightarrow 1 + n_p = 1 + n_{p-1} + 1 + n_{p-2}$$

Si llamamos  $H_p = 1 + n_p$ , se tiene

$$H_p = H_{p-1} + H_{p-2}$$

con  $H_0 = 2 = F_2$ , y  $H_1 = 3 = F_3$ . Por tanto  $H_p$  es la sucesión de Fibonacci desplazada dos pasos hacia adelante.

Se puede demostrar que:

$$F_N = \frac{1}{\sqrt{5}} (\Phi^{N+1} + \Psi^{N+1})$$

con  $\Phi = \frac{1+\sqrt{5}}{2}$  y por tanto  $\Phi > 1$ , y  $\Psi = \frac{1-\sqrt{5}}{2}$ , y por tanto  $|\Psi| < 1$  y por tanto  $\Phi^{N+1} \xrightarrow{N \rightarrow \infty} \infty$  y  $\Psi^{N+1} \xrightarrow{N \rightarrow \infty} 0$ . En consecuencia

$$F_N \sim \frac{\Phi}{\sqrt{5}} \Phi^N \Rightarrow n_p \sim F_{p+2} \sim \frac{\Phi^3}{\sqrt{5}} \Phi^p$$

Por tanto, si  $T$  es un AVL con  $N$  nodos y profundidad  $p$  tenemos que  $N \geq n_p$  y  $n_p \sim C\Phi^p$ , con  $C = \frac{\Phi^3}{\sqrt{5}}$  por tanto  $\log n \geq p \cdot \log \Phi + \log C$  y por tanto

$$\log N = \Omega(\text{prof}(T)),$$

con lo cual  $\text{prof}(T) = O(\log N)$  según queríamos demostrar.

Por tanto si usamos como estructura de datos para el diccionario un AVL tenemos que

$$W_{\text{Buscar}}(N) = O(\log N) \tag{3.16}$$

$$W_{\text{Insertar}}(N) = O(\log N) \tag{3.17}$$

Faltaría estudiar el caso de la primitiva **Borrar**; sin embargo, borrar nodos en un AVL es una tarea complicada, esencialmente debido a que es muy complicado

(aunque posible) reajustar los nodos para que el árbol continúe siendo AVL tras la eliminación. La solución que se suele implementar para el borrado de AVLs consiste en el llamado *borrado perezoso*, en el cual, en lugar de eliminar el nodo y reajustar el árbol, simplemente marcamos el nodo como eliminado y dejamos libre esa posición por si el nodo se reinserta, o bien, cada cierto tiempo, cuando hay ya un número elevado de posiciones libres, se comprime el AVL haciendo una reconstrucción completa con los elementos que no se han eliminado.

# Capítulo 4

## Hashing

Los resultados vistos hasta ahora pueden resumirse en la siguiente tabla:

Rendimiento	Ordenación	Búsqueda
Normal	$N^2$	$N$
Bueno	$N \log N$	$\log N$
Óptimo	$N$	¿ $O(1)$ ?

Parece que el rendimiento de los algoritmos de búsqueda corresponde al rendimiento de los algoritmos de ordenación divididos por  $N$ . Por tanto podemos preguntarnos si es posible hacer búsquedas en tiempo  $O(1)$ . Tal como vimos en el capítulo anterior, la respuesta es no si la búsqueda se realiza mediante comparaciones de clave y por tanto habrá que buscar otro tipo de algoritmo de búsqueda que no funcione mediante comparaciones de clave.

El escenario que nos encontramos es el siguiente:

1. Tenemos un cierto diccionario  $\mathcal{D} = \{D : \text{datos}\}$ .
2. Cada dato  $D \in \mathcal{D}$  tiene una clave única  $k$ .
3. Buscamos el dato en  $\mathcal{D}$  por la clave, pero no mediante comparaciones de clave.

Una primera aproximación al escenario anterior es la siguiente:

1. Calculamos  $k^* = \max\{k(D) : D \in \mathcal{D}\}$  (i.e buscamos la clave más grande)
2. Guardamos  $D$  en una tabla  $T$  de tamaño  $k^*$ . Así, si  $k = k(D) \Rightarrow T[k] = D$

Con esta aproximación podemos buscar con el siguiente pseudocódigo:

```
ind Buscar(dato D, tabla T)
  si T[k(D)]==D
    return k(D);
  else
    devolver NUL
```

Es fácil ver que  $n_{\text{Buscar}}(D, T) = O(1)$ . Sin embargo esta aproximación tiene el problema que el coste en espacio puede ser exagerado con respecto al número de datos a considerar.

Por ejemplo, consideremos que queremos realizar búsquedas sobre los 200 estudiantes de un curso de la EPS, es decir,  $(D) = \{\text{Estudiantes de segundo de la EPS}\}$ . Si la clave de búsqueda correspondiente a cada dato (estudiante) es el número de DNI tendríamos  $k^* = \text{máx DNI} \simeq 7 \times 10^7$ , lo cual sería una cantidad exagerada de memoria para almacenar solamente 200 datos. Obviamente si el problema anterior no existe para nuestro conjunto de datos esta es una excelente aproximación a considerar; sin embargo, como hemos apuntado, no siempre (de hecho, casi nunca) es una aproximación viable.

Podemos mejorar la idea anterior utilizando tablas más pequeñas:

1. Fijamos  $M \gtrsim |\mathcal{D}|$
2. Definimos una función  $\mathcal{K} : \{k(D) : D \in \mathcal{D}\} \longrightarrow \{1, 2, 3, \dots, M\}$  que sea inyectiva (i.e  $x \neq y \Rightarrow \mathcal{K}(x) \neq \mathcal{K}(y)$ ).
3. Insertamos el elemento  $D$  en  $T[\mathcal{K}(k(D))]$  en lugar de en  $T[k(D)]$ .

Ahora la dimensión de la tabla  $T$  es  $M$  y el pseudocódigo de **Buscar** es muy similar al anterior:

```
ind Buscar2(dato D, tabla T)
  si T[K(k(D))]==D
    return K(k(D));
  else
    devolver NULL
```

De nuevo es fácil ver que  $n_{\text{Buscar2}}(D, T) = O(1)$  y además hacemos un buen uso de espacio en memoria. El problema ahora surge en que, en general no es posible construir una función  $\mathcal{K}$  universal sobre cualquier conjunto de claves que garantice la inyectividad. Sí es posible construir funciones que sean inyectivas para un conjunto particular de datos; sin embargo esas mismas funciones pueden (de hecho suelen) no ser inyectivas para otro conjunto de datos distinto o incluso



en el mismo conjunto si simplemente se quitan o añaden datos, y por tanto el pseudocódigo anterior deja de ser válido.

La solución al problema anterior podría ser usar funciones  $\mathcal{K}$  universales (independientes del conjunto de claves concreto que se usa) pero relajando la condición de inyectividad, es decir, vamos a permitir que haya claves  $k \neq k'$  pero  $\mathcal{K}(k) = \mathcal{K}(k')$ . Si estamos en la situación anterior, es decir  $k \neq k'$  pero  $\mathcal{K}(k) = \mathcal{K}(k')$ , decimos que ha habido una **colisión**. En un colisión hay un choque de dos claves distintas por entrar en el mismo lugar de la tabla; sin embargo esperamos que las colisiones sean pocas. El hecho de que existan las colisiones (aunque estas sean poco frecuentes) implica que tenemos que implementar un **mecanismo de resolución de colisiones**.

## 4.1. Construcción de funciones hash

A partir de ahora, como notación usaremos la letra  $h$  para denominar una función hash genérica.

Nuestro objetivo ahora será construir una función  $h$  tal que si  $k \neq k'$  la probabilidad de que  $h(k) = h(k')$  sea pequeña. Dado que en la tabla  $T$  hay  $M$  posiciones lo mejor a lo que podemos aspirar es a que dadas dos claves  $k \neq k'$

$$p(h(k) = h(k')) = \frac{1}{M} \quad (4.1)$$

Una función hash que verifique la condición anterior se dice función hash **uniforme**. Las funciones hash uniformes son ideales y no alcanzables mediante funciones algorítmicas; sin embargo, cabe esperar que el rendimiento para ellas será el mejor posible.

Una primera idea para tal función sería generar un número aleatorio entre 1 y  $M$  (es decir, tirar una especie de dado de  $M$  caras), por ejemplo, empleando la función **rand** de C. Obviamente esta solución es inviable ya que la asignación de  $h(k)$  para sucesivas aplicaciones de la función  $h$  a la clave  $k$  sería distinta. Además deseamos usar la función hash también para buscar, con lo cual, es necesario que  $h(k)$  devuelva siempre el mismo valor al recibir  $k$ . Esto es, queremos una función determinista pero que tenga un comportamiento casi aleatorio. Existen funciones que simulan series aleatorias de datos y que devuelven siempre el mismo resultado para una entrada dada. La idea sería este tipo de funciones (que son, por ejemplo, las que usa la rutina **rand** de C) para construir nuestra función hash.

Para esto consideramos dos técnicas, funciones hash de división y de multiplicación.

### 4.1.1. Hash de división

Dado un diccionario  $(D)$  fijamos un número  $m \gtrsim |(D)|$  que sea primo. Definimos

$$h_d(k) = k \% m \quad (4.2)$$

donde  $\%$  es el operador módulo que devuelve el resto de la división entre  $k$  y  $m$ . Con alguna condición adicional (por ejemplo que  $m$  este lejos de potencias de 2). Junto con otras modificaciones técnicas, esto garantiza que si  $k \neq k' \Rightarrow p(k \% m = k' \% m) \sim \frac{1}{M}$ , esto es,  $p(h_d(k) = h_d(k')) \sim \frac{1}{M}$ .

### 4.1.2. Hash de multiplicación

En este caso elegimos  $m \gtrsim |(D)|$  tal que  $m$  no sea primo (tienen un buen comportamiento los números de la forma  $m = 2^n$  o  $m = 10^n$ ) y un cierto número  $\Phi$  irracional por ejemplo  $\Phi = \left(\frac{1+\sqrt{5}}{2}\right)$  o  $\Phi = \left(\frac{1-\sqrt{5}}{2}\right)$ , y ahora definimos

$$h_m(k) = \lfloor m(k \times \Phi) \rfloor \quad (4.3)$$

donde en este caso el operador  $\lfloor \cdot \rfloor$  es el operador **parte fraccionaria**, es decir  $\lfloor x \rfloor = x - [x]$ . Es fácil ver que  $0 \leq h(k) \leq m - 1$  ya que  $0 \leq (k \times \Phi) < 1$ . De nuevo en cierto sentido el hash de multiplicación garantiza que si  $k \neq k' \Rightarrow p(k \% m = k' \% m) \sim \frac{1}{M}$ , esto es,  $p(h_m(k) = h_m(k')) \sim \frac{1}{M}$ .

## 4.2. Mecanismos de resolución de colisiones

Hemos visto que las funciones hash de división y multiplicación tienen una baja probabilidad de que exista una colisión; sin embargo no pueden garantizar que no haya colisiones. Por tanto es necesario implementar algún tipo de mecanismo de resolución de colisiones.

### 4.2.1. Hash con encadenamiento

La idea del hash con encadenamiento consiste en que la tabla  $T$  no albergue directamente los datos  $D \in (D)$ , sino que cada posición de la tabla almacena un puntero a una lista enlazada, es decir, la tabla  $T$  es una tabla de punteros a  $N$  listas enlazadas. En la figura 4.1 se puede ver el aspecto de una tabla hash con encadenamiento.

Con este tipo de tabla hash tenemos el siguiente pseudocódigo para **Buscar**

```
ind Buscar(dato D, tabla T)
  return BLin(D, T[h(k(D))]);
```

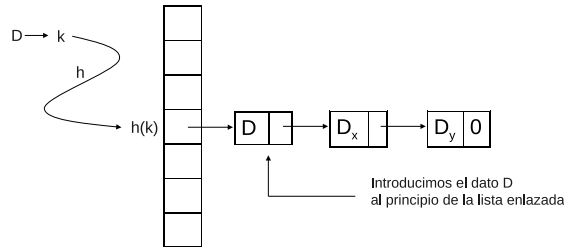


Figura 4.1: Tabla hash con encadenamiento.

donde **Blin** es una búsqueda lineal en listas enlazadas, que devuelve **NULL** si no encuentra la clave.

Ahora **Buscar** ya no es  $O(1)$  ya que en **Blin** hay un bucle **while**, por tanto tenemos que en el caso peor  $W_{Buscar}(N) = N$ . Este caso se da cuando todos los elementos  $D \in (D)$  se encuentran en la misma lista enlazada, es decir,  $W_{Buscar}(N) = N \Leftrightarrow (h(k) = h(k') \forall k, k')$ . En la figura 4.2 se puede ver esta situación.

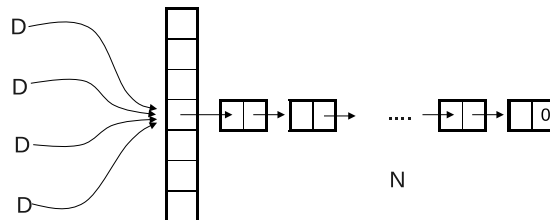


Figura 4.2: Caso peor de búsquedas hash con encadenamiento.

La situación anterior, aunque puede ocurrir, es muy poco probable si la función hash está bien construida (es uniforme), ya que en ese caso si,  $k \neq k'$ ,  $p(h(k) = h(k')) = \frac{1}{m}$ . En todo caso, si vemos que con un conjunto de datos y una función hash se da la situación anterior, será conveniente emplear una función hash distinta.

Vamos a analizar el rendimiento de las búsquedas en tablas hash con encadenamiento. Para ello damos la siguiente definición.

Dada una tabla hash de dimensión  $m$  que contiene  $N$  datos definimos el **factor de carga** de la tabla hash como:

$$\lambda = \frac{N}{m} \quad (4.4)$$

**Proposición.** Sea  $h$  función hash uniforme en una tabla hash con encadenamiento, dimensión  $m$  y  $N$  datos; entonces

$$A_{BHE}^f(N, m) = \frac{N}{m} = \lambda \quad (4.5)$$

$$A_{BHE}^e(N, m) = 1 + \frac{\lambda}{2} + O(1) \quad (4.6)$$

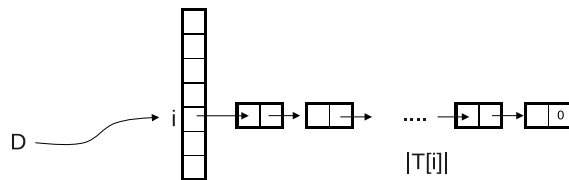
donde  $A_{BHE}^f(N, m)$  es el coste medio de las búsquedas sin éxito (la clave no está en la tabla) en una tabla hash con encadenamiento y  $A_{BHE}^e(N, m)$  es el coste medio de las búsquedas con éxito.

**Demostración.** Como operación básica usaremos el **sondeo** entendido como sondeo el acceso a un nodo de la lista enlazada. Por comodidad no vamos a distinguir a partir de ahora entre la clave y el dato, es decir  $i = h(k(D))$  lo denotaremos simplemente como  $i = h(D)$ .

Para demostrar (4.5) sea  $D$  un dato que **no** está en la tabla hash y, por tanto,

$$n_{BHE}(D, T) = |T[i]|,$$

donde  $|T[i]|$  es la longitud de la lista enlazada apuntada por  $T[i]$ .



Por tanto

$$A_{BHE}^f(N, m) = \sum_{i=1}^m \underbrace{p(h(D) = i)}_{=\frac{1}{m} \text{ (f uniforme)}} \cdot |T[i]| = \frac{1}{m} \sum_{i=1}^m |T[i]| = \frac{N}{m} = \lambda$$

donde se ha usado  $\sum_{i=1}^m |T[i]| = N$  ya que la suma de los elementos que hay en cada lista enlazada debe ser el total de los elementos que hay en tabla, es decir,  $N$ .

Para demostrar (4.6) vamos a reducir la búsqueda hash con encadenamiento con éxito a una con fracaso en una tabla más pequeña. Para ello supongamos que los datos de la tabla  $T$  están numerados  $\{D_1, D_2, \dots, D_N\}$  por el orden de inserción en la tabla hash  $T$ . Denotamos por  $T_i$  el estado de  $T$  tras la inserción de los elementos  $D_1, D_2, \dots, D_{i-1}$ ; por tanto, en elemento  $D_i$  no está en la tabla  $T_i$ .

Vamos a considerar además, sin pérdida de generalidad, que los datos se insertan al final de la listas enlazadas, con lo cual se tiene:

$$n_{BHE}^e(D_i, T) = 1 + n_{BHE}^f(D_i, T_i);$$

es decir, el coste de la búsqueda con éxito en  $T$  es igual al coste de la búsqueda con fracaso más un sondeo extra. Por tanto, como  $n_{BHE}^f(D_i, T) \simeq A_{BHE}^f(i-1, m)$ , tenemos

$$n_{BHE}^e(D_i, T) \simeq 1 + A_{BHE}^f(i-1, m) = 1 + \frac{i-1}{m}$$

y en consecuencia

$$\begin{aligned} A_{BHE}^e(N, m) &= \frac{1}{N} \sum_{i=1}^N n_{BHE}^e(D_i, T) = \frac{1}{N} \sum_{i=1}^N \left(1 + \frac{i-1}{m}\right) \\ &= 1 + \frac{1}{Nm} \sum_{i=1}^{N-1} j = 1 + \frac{1}{Nm} \frac{N(N-1)}{2} = 1 + \frac{N}{2m} - \frac{1}{2m} \\ &= 1 + \frac{\lambda}{2} + O(1) \end{aligned}$$

como queríamos demostrar.

Por tanto tenemos que  $A_{BHE}^f(N, m) = \lambda$  y  $A_{BHE}^e(N, m) = 1 + \frac{\lambda}{2} + O(1)$  son en tiempo  $O(1)$  si  $h$  es uniforme y  $N \simeq m \Rightarrow \lambda = \theta(1)$ .

Por ejemplo, si  $N = 500$  y  $m = 100$  se tiene que

$$A_{BHE}^f(500, 100) \simeq 5 \text{ y } A_{BHE}^f(100, 500) \simeq 1 + 5/2 = 3,5.$$

### 4.2.2. Hash con direccionamiento abierto

En este casos situaremos los datos en la propia tabla; por tanto, ahora cada elemento  $T[i], i = 1, \dots, m$  de la tabla no alberga un puntero a una lista enlazada sino que alberga un dato. Por tanto, en la tabla tendremos posiciones ocupadas por datos junto a posiciones que están libres. En la figura 4.3 se puede ver el aspecto de una tabla hash con direccionamiento abierto así como el estado de la tabla antes y después de insertar un dato.

Para la resolución de colisiones veremos tres métodos; todos ellos son mecanismos de repetición de sondeos.

1. **Sondeo lineal:** Si la posición  $p_0 = T[h(D)]$  está ocupada, intentamos situar el dato sucesivamente en las posiciones  $(p+1) \% m, (p+2) \% m, \dots$  hasta llegar a una posición  $p_i = (p+i) \% m$  que esté libre.

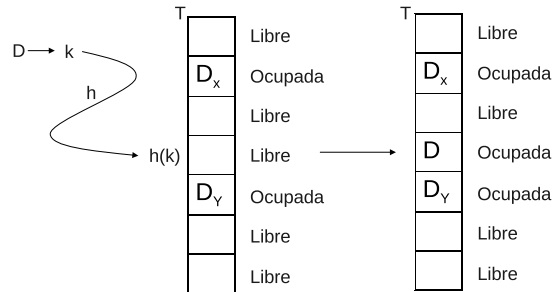


Figura 4.3: Inserción en tablas hash con direccionamiento abierto.

En la figura 4.4 se puede ver el proceso de inserción de un nuevo dato mediante sondeos lineales

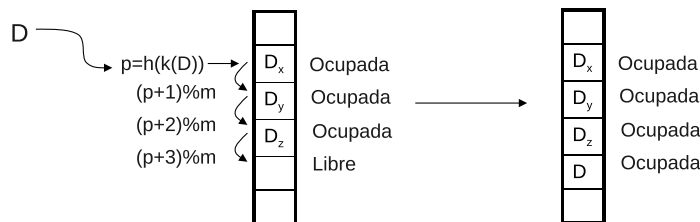


Figura 4.4: Inserción con sondeos lineales.

2. **Sondeo cuadrático** El funcionamiento es similar al de los sondeos lineales. Si la posición  $p_0 = T[h(D)]$  está ocupada probamos las posiciones  $(p + 1^2)\%m$ ,  $(p + 2^2)\%m$ ,  $(p + 3^2)\%m$ ... hasta llegar a una posición  $p_i = (p + i^2)\%m$  que esté libre
3. **Sondeo aleatorio** Si la posición  $p_0 = T[h(D)]$  está ocupada se intenta en las posiciones  $p_1, p_2, \dots, p_{i-1}$  generadas aleatoriamente hasta llegar a una posición  $p_i$  libre.

Obviamente esta ultima opción es inviable en la práctica pero presenta una situación ideal en tablas hash y además es útil para obtener estimaciones de rendimiento.

A diferencia del hash con encadenamiento, donde la dirección de un dato  $D$  depende exclusivamente a la posición de  $D$  en la tabla hash (la cual viene determinada de forma fija por el valor de  $h(D)$  y de una cierta subdirección de una lista enlazada), en el hash con direccionamiento abierto la dirección de  $D$  va a depender:

- i: Del valor de  $h(D)$
- ii: Del estado de la tabla antes de la inserción.

Por tanto el dato  $D$  decimos que tiene una dirección **abierta** ya que no depende solo del dato  $D$ .

Además en el hashing con encadenamiento se tiene que  $\lambda = \frac{N}{M}$  puede ser mayor que 1; sin embargo en el hashing con direccionamiento abierto  $N$  siempre tiene que ser menor o igual que  $m$ . De hecho en la práctica  $N \ll m$  y  $\lambda \ll 1$ , una práctica habitual es tomar  $\lambda \lesssim \frac{1}{2} \Rightarrow 2N \lesssim m$ .

Vamos a calcular el rendimiento del hash con direccionamiento abierto.

**Proposición.** Si  $h$  es una función hash uniforme y se usan sondeos aleatorios entonces

$$A_{BHA\_Aleat}^f(N, m) = \frac{1}{1 - \lambda} \quad (4.7)$$

$$A_{BHA\_Aleat}^e(N, m) = \frac{1}{\lambda} \log \left( \frac{1}{1 - \lambda} \right) \quad (4.8)$$

donde  $A_{BHA\_Aleat}^f(N, m)$  es el coste medio de las búsquedas sin éxito (la clave no está en la tabla) en una tabla hash con direccionamiento abierto y sondeos aleatorios y  $A_{BHA\_Aleat}^e(N, m)$  es el coste medio de las búsquedas con éxito.

Podemos observar, además que

$$A_{BHA\_Aleat}^f(N, m) \xrightarrow{\lambda \rightarrow 1} \infty \text{ y } A_{BHA\_Aleat}^e(N, m) \xrightarrow{\lambda \rightarrow 1} \infty.$$

**Demostración.** Para demostrar (4.7) sea  $h$  función hash uniforme; se tiene:

$$\begin{aligned} \text{prob}(T[i] \text{ ocupado}) &= \frac{N}{m} = \lambda \\ \text{prob}(T[i] \text{ libre}) &= 1 - \lambda \end{aligned}$$

con  $i = 1, 2, \dots, m$ .

Por tanto

$$A_{BHA\_Aleat}^f(N, m) = \sum_{k=1}^{\infty} \text{coste de hacer } k \text{ sondeos} \cdot \text{prob}(\text{necesitar } k \text{ sondeos}).$$

Cuando se necesitan  $k$  sondeos para la búsqueda fallida,  $k - 1$  llevan a posiciones ocupadas y el último a una libre. Por tanto,

$$\text{prob}(\text{necesitar } k \text{ sondeos}) = \text{prob}(\text{ocupado})^{k-1} \cdot \text{prob}(\text{libre}) = \lambda^{k-1}(1 - \lambda)$$

y como consecuencia se tiene

$$\begin{aligned} A_{BHA\_Aleat}^f(N, m) &= \sum_{k=1}^{\infty} k \lambda^{k-1} (1 - \lambda) = (1 - \lambda) \sum_{k=1}^{\infty} k \lambda^{k-1} \\ &= (1 - \lambda) \frac{d}{d\lambda} \left( \sum_{k=1}^{\infty} \lambda^k \right) = (1 - \lambda) \frac{d}{d\lambda} \left( \frac{1}{1 - \lambda} \right) \\ &= (1 - \lambda) \frac{1}{(1 - \lambda)^2} = \frac{1}{(1 - \lambda)}. \end{aligned}$$

Para demostrar (4.8) procedemos de la misma forma que en hash con encadenamiento. Enumeramos los datos en  $T$  como  $\{D_1, D_2, \dots, D_N\}$  según el orden de entrada en la tabla  $T$ .

Observamos que el número de sondeos en  $T$  para encontrar  $D_i$ ,  $n_T^e(D_i)$ , no sólo depende de  $h(D_i)$  sino también del estado de la tabla  $T$  en el momento de insertar  $D_i$ , pero no de los elementos que se inserten después de  $D_i$ .

Por tanto tenemos que  $n_T^e(D_i)$  (número de sondeos para encontrar  $D_i$  en  $T$ ) es igual al número de sondeos necesarios para insertar  $D_i$  en  $T_i$ , donde recordamos que  $T_i$  es la tabla hash tras la inserción de los elementos  $D_1, \dots, D_{i-1}$ .

Además el número de sondeos que hacen falta para buscar  $D_i$  es igual al número de sondeos necesario para buscar (con fallo) el elemento  $D_i$  en  $T_i$ ,  $n_{T_i}^f(D_i)$ .

Con estas observaciones tenemos:

$$n_T^e(D_i) = n_{T_i}^f(D_i) \simeq A_{BHA\_Aleat}^f(i - 1, m)$$

y por tanto

$$\begin{aligned} A_{BHA\_Aleat}^e(N, m) &= \frac{1}{N} \sum_{i=1}^N n_T^e(D_i) \simeq \frac{1}{N} \sum_{i=1}^N \frac{1}{1 - \frac{i-1}{m}} \\ &= \frac{1}{N} \sum_{j=0}^{N-1} \frac{1}{1 - \frac{j}{m}} \simeq \frac{1}{N} \int_0^N \frac{1}{1 - \frac{x}{m}} dx \underbrace{= \frac{1}{N} \int_0^{\frac{N}{m}} \frac{1}{1 - u} du}_{u = \frac{x}{m}, x = m \cdot u, dx = m \cdot du} \\ &= \frac{1}{\lambda} \int_0^\lambda \frac{1}{1 - u} du = \frac{1}{\lambda} \log \left( \frac{1}{1 - \lambda} \right) \end{aligned}$$



tal como queríamos demostrar.

Observamos que si llamamos  $f(\lambda) = A_{BHA\_Aleat}^f(N, m) = \frac{1}{1-\lambda}$  se tiene que

$$A_{BHA\_Aleat}^f(N, m) = \frac{1}{\lambda} \int_0^\lambda f(u) du = \frac{1}{\lambda} \log \left( \frac{1}{1-\lambda} \right).$$

En general, si  $\mathcal{S}$  es un tipo de sondeo con direccionamiento abierto y el coste medio de las búsquedas sin éxito es  $A_{BHA\_S}^f(N, m) = f(\lambda)$ , entonces se puede demostrar con los razonamientos anteriores que

$$A_{BHA\_S}^e(N, m) \simeq \frac{1}{\lambda} \int_0^\lambda f(u) du \quad (4.9)$$

Por ejemplo, en el caso de sondeos lineales se tiene que

$$A_{BHA\_SL}^f(N, m) \simeq \frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right) \quad (4.10)$$

y entonces se sigue que

$$A_{BHA\_SL}^e(N, m) = \frac{1}{\lambda} \int_0^\lambda \frac{1}{2} \left( 1 + \frac{1}{(1-u)^2} \right) du = \frac{1}{2} \left( 1 + \frac{1}{1-\lambda} \right). \quad (4.11)$$