Análisis y diseño de algoritmos

2. Eficiencia

Juan Morales García,
Victor M. Sánchez Cartagena,
Jose Luis Verdú Mas

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

13 de febrero de 2025

Objetivos

- Proporcionar la capacidad para analizar con rigor la eficiencia de los algoritmos
  - Distinguir los conceptos de eficiencia en tiempo y en espacio
  - Entender y saber aplicar criterios asintóticos a los conceptos de eficiencia
  - Saber calcular la complejidad temporal o espacial de un algoritmo
  - Saber comparar, en cuanto a su eficiencia, distintas soluciones algorítmicas a un mismo problema

Contenido

Índice

¿Qué es un algoritmo?

Definición (Algoritmo)

Un algoritmo es una serie finita de instrucciones no ambiguas que expresa un método de resolución de un problema

Importante:

- El modelo de computación ha de estar preestablecido (para que los resultados sean comparables entre sí)
  - Máquina RAM (*Random Access Machine*)
    (es el modelo teórico de la **arquitectura "Von Neumann")**
- Los recursos (usualmente tiempo y memoria) necesarios para cada paso elemental estarán acotados
- El algoritmo debe terminar en un número finito de pasos

Noción de complejidad

Definición (Complejidad algorítmica)

Es una medida de los recursos que necesita un algoritmo para resolver un problema

Los recursos mas usuales son:

- Tiempo: Complejidad temporal
- Memoria: Complejidad espacial

Se suele expresar en función de la dificultad *a priori* del problema:

- Tamaño del problema: lo que ocupa su representación
- Parámetro representativo: *i.e.* la dimensión de una matriz

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	
Decir cuál es el mayor de 2 números	
Ordenar un vector de n enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	
Ordenar un vector de n enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de n enteros	
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	tamaño
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

- Usualmente se omite el tamaño de enteros, reales, punteros, etc. si se asume que su tamaño está acotado

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

Problema	**tamaño**
Sumar uno a un entero (binario de 32 bits)	32
Decir cuál es el mayor de 2 números	$2 \cdot 32$
Ordenar un vector de n enteros	$32n$
Multiplicar dos matrices de enteros de $m \times n$ y $n \times \ell$	$32(mn + n\ell)$

- Usualmente se omite el tamaño de enteros, reales, punteros, etc. si se asume que su tamaño está acotado
- ¿Cuántos bits se necesitan para codificar un entero n arbitrariamente grande?

Atención:

La complejidad puede depender de cómo se codifique el problema

Ejemplo

Sumar uno a un entero arbitrariamente grande

- Complejidad constante si el entero se codifica en base uno
- Complejidad lineal si el entero se codifica en base dos

En nuestro modelo de computación, no se permite:

- codificaciones en base uno
- codificaciones no compactas

El tiempo de ejecución

El tiempo de ejecución de un algoritmo depende de:

Factores externos

- La máquina en la que se va a ejecutar
- El compilador
- Los datos de entrada suministrados en cada ejecución

Factores internos

- El número de instrucciones que ejecuta el algoritmo y su duración

¿Cómo estudiamos el tiempo de ejecución?

Definición (Análisis empírico o *a posteriori*)

Ejecutar el algoritmo para distintos valores de entrada y cronometrar el tiempo de ejecución

- ▲ Es una medida del comportamiento del algoritmo en su entorno
- ▼ El resultado depende de los factores externos e internos

Definición (Análisis teórico o *a priori*)

Obtener una función que represente el tiempo de ejecución (en operaciones elementales) del algoritmo para cualquier valor de entrada

- ▲ El resultado depende sólo de los factores internos
- ▲ No es necesario implementar y ejecutar los algoritmos
- ▼ No obtiene una medida real del comportamiento del algoritmo en el entorno de aplicación

Tiempo de ejecución de un algoritmo

Definición (Operaciones elementales)

Son aquellas operaciones que realiza el ordenador en un tiempo acotado por una constante

Ejemplo (Operaciones elementales)

- Operaciones aritméticas básicas
- Asignaciones a variables de tipo predefinido por el compilador
- Los saltos (llamadas a funciones, retorno desde ellos . . .)
- Comparaciones lógicas
- Acceso a estructuras indexadas básicas (vectores o matrices)

Tiempo de ejecución de un algoritmo

Para simplificar, se suele considerar que el coste temporal de las operaciones elementales es unitario

Definición (Tiempo de ejecución de un algoritmo)

Una función ($T(n)$) que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de problema n

Ejemplo 1: Suma del los elementos de un vector

Ejemplo (sintaxis de la STL)

```
int sumar( const vector<int> &v){
  int s = 0;

  for(int i = 0; i < v.size(); i++)
    s += v[i];

  return s;
}
```

Si estudiamos el bucle ($n =$ `v.size()`):

n	asign.	comp.	inc.	total
0	1	1	0	2
1	1	2	1	4
2	1	3	2	6
⋮	⋮	⋮	⋮	⋮
n	1	$n+1$	n	$2+2n$

La complejidad del algoritmo será:

$$T(n) = \underbrace{1}_{\text{primera asignación}} + \underbrace{2+2n}_{\textit{bucle}} + \underbrace{n}_{\text{interior del bucle}} = 3 + 3n$$

Ejemplo 2: Traspuesta de una matriz cuadrada

Traspuesta de una matriz $d \times d$

```
void traspuesta( mat& A){  // supongo que A.n_rows == A.n_cols
  for( int i = 1; i < A.n_rows; i++ )
    for( int j = 0; j < i ; j++ )
      swap( A(i,j), A(j,i) );
}
```

Como la complejidad del bucle interior es: $2 + 3i$ veces

$$T_d(d) = \underbrace{2(d-1)+2}_{\text{bucle exterior}} + \underbrace{\sum_{i=1}^{d-1}(2+3i)}_{\text{interior}} = \cdots = O(d^2)$$

Si queremos la complejidad con respecto al tamaño del problema ($n = d^2$):

$$T_n(n) = T_d(d) = O(d^2) = O(n)$$

Simplificación mediante *cuenta de pasos*

- La cuenta de *pasos de programa* (en lugar de instrucciones) permite obtener expresiones menos farragosas.

Definición (Paso de programa)

Agrupación de instrucciones cuyo tiempo de ejecución es constante con el tamaño del problema

- Las instrucciones no tienen porqué estar consecutivas.
- Se trata de suponer que ese conjunto de instrucciones actúan como si fuera una sola.

Ejercicio

```
int ejemplo( int n ){
  int j = 2;
  for( int i = 0; i < 2000; i++ ) {
    j = j * j;
  }
  int i = 0;
  while( i < n ) {
    i = i + 1;
  }
  while (j>0){
    j--;
  }
  return j;
}
```

Si queremos calcular la complejidad temporal en función de *n* (complejidad paramétrica), ¿cuántos pasos de programa realiza la función `ejemplo`?

Ejercicio

```
int ejemplo( int n ){
  int j = 2;
  for( int i = 0; i < 2000; i++ ) {
    j = j * j;
  }
  int i = 0;
  while( i < n ) {
    i = i + 1;
  }
  while (j>0){
    j--;
  }
  return j;
}
```

Si queremos calcular la complejidad temporal en función de n (complejidad paramétrica), ¿cuántos pasos de programa realiza la función `ejemplo`?
Solución: $n + 1$ pasos

Ejemplo 2: Traspuesta de una matriz cuadrada (II)

Traspuesta de una matriz $d \times d$

```
void traspuesta( mat& A){   // supongo que A.n_rows == A.n_cols
  for( int i = 1; i < A.n_rows; i++ )
    for( int j = 0; j < i ; j++ )
      swap( A(i,j), A(j,i) );
}
```

Volviendo al ejemplo anterior, calculamos la complejidad con respecto a d mediante cuenta de pasos:

$$T_d(d) = \sum_{i=1}^{d-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{d-1} i = (d-1)\frac{1+d-1}{2} \in O(d^2)$$

Ejemplo 3: Producto de dos matrices cuadradas (I)

Producto de dos matrices $d \times d$ (Sintaxis de la librería `armadillo`)

```
1  mat producto( const mat &A, const mat &B ){
2    mat R(A.n_rows, B.n_cols);
3    for( int i = 0; i < A.n_rows; i++ )
4      for( int j = 0; j < B.n_cols; j++ ) {
5        double acc = 0.0;
6        for( int k = 0; k < A.n_cols; k++ )
7          acc += A(i,k) * B(k,j);
8        R(i,j) = acc;
9      }
10   }
11   return R;
12 }
```

- La complejidad de las líneas 6-7 es $O(d)$
- La complejidad de las líneas 4-9 es $O(d) + d \cdot O(d) = O(d^2)$
- La complejidad de las líneas 3-10 es $O(d) + d \cdot O(d^2) = O(d^3)$

La complejidad del algoritmo será: $T_d(d) = O(d^3)$

Ejemplo 3: Producto de dos matrices cuadradas (II)

Producto de dos matrices $d \times d$ (Sintaxis de la librería `armadillo`)

```
mat producto( const mat &A, const mat &B ){
  mat R(A.n_rows, B.n_cols);
  for( int i = 0; i < A.n_rows; i++ )
    for( int j = 0; j < B.n_cols; j++ ) {
      double acc = 0.0;
      for( int k = 0; k < A.n_cols; k++ )
        acc += A(i,k) * B(k,j);
      R(i,j) = acc;
    }
  }
  return R;
}
```

Cuenta de pasos:

$$T_d(d) = \sum_{i=0}^{d-1} \sum_{j=0}^{d-1} (1 + \sum_{k=0}^{d-1} 1) \in O(d^3)$$

Ejemplo 3: Producto de dos matrices cuadradas (III)

¿Cual es la complejidad con respecto al tamaño?

El tamaño del problema es $n = 2d^2$ por lo que $d = \sqrt{n/2}$

$$T_n(n) = T_d(d) = O(d^3) = O\left(\left(\sqrt{n/2}\right)^3\right) = O(n^{3/2})$$

¿Cual es la complejidad espacial?

- En la complejidad espacial no se tiene en cuenta lo que ocupa la codificación del problema.
- Solo se tiene en cuenta lo que es imputable al algoritmo.
  - Con respecto a d: $M_d(d) = O(d^2)$
  - Con respecto al tamaño del problema n:

$$M_n(n) = M_d(d) = O(d^2) = O\left(\left(\sqrt{n/2}\right)^2\right) = O(n)$$

- Dado un vector de enteros v y el entero z
  - Devuelve el primer índice `i` tal que `v[i] == z`
  - Devuelve -1 en caso de no encontrarlo

Búsqueda de un elemento

```
int buscar( const vector<int> &v, int z ){
  for( int i = 0; i < v.size(); i++ )
    if( v[i] == z )
      return i;
  return -1;
}
```

Problema

- No podemos contar el número de pasos porque para diferentes entradas de un mismo tamaño de problema se obtienen diferentes complejidades
- En el ejemplo de la transparencia anterior:

v	z	Instrucciones
$(1,0,2,4)$	1	3
$(1,0,2,4)$	0	6
$(1,0,2,4)$	2	9
$(1,0,2,4)$	4	12
$(1,0,2,4)$	5	14

- ¿Qué podemos hacer?
  - Acotar el coste mediante dos funciones que expresen respectivamente, el coste máximo y el coste mínimo del algoritmo (cotas de complejidad)

Índice

Cotas de complejidad

- Cuando aparecen diferentes casos para una misma talla n, se introducen las siguientes medidas de la complejidad
  - Caso peor: cota superior del algoritmo $\rightarrow C_s(n)$
  - Caso mejor: cota inferior del algoritmo $\rightarrow C_i(n)$
  - Caso promedio: coste promedio $\rightarrow C_m(n)$
- Todas son funciones del tamaño del problema
- El coste promedio es difícil de evaluar a priori
  - Es necesario conocer la distribución de probabilidad de la entrada
  - ¡No es la media de la cota inferior y de la cota superior!

Cotas superior e inferior

Buscar elemento

```
int buscar( const vector<int> &v, int z ) {
  for( int i = 0; i < v.size(); i++ )
    if( v[i] == z )
      return i;
  return -1;
}
```

- En este caso el tamaño del problema es $n = \texttt{v.size()}$

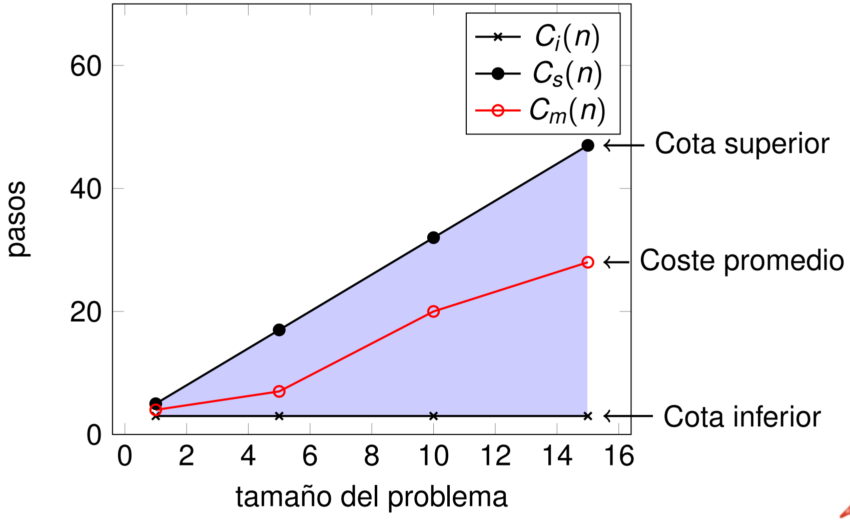
	Mejor caso	Peor caso
	$1+1+1$	$1+3n+1$
Suma	3	$3n+2$

Cotas:

$$C_s(n) = 3n + 2 \quad \in O(n)$$
$$C_i(n) = 3 \quad \in \Omega(1)$$

Cotas superior e inferior

- Coste de la función `buscar`



Análisis asintótico

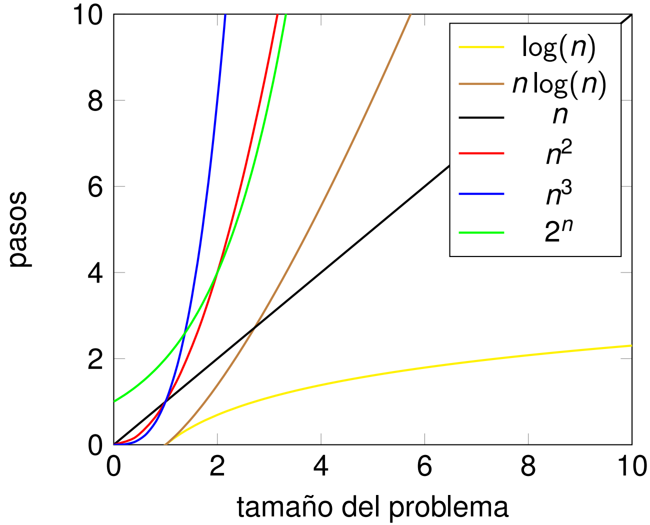
- El estudio de la complejidad resulta realmente interesante para tamaños grandes de problema por varios motivos:
  - Las diferencias “reales” en tiempo de ejecución de algoritmos con diferente coste para tamaños pequeños del problema no suelen ser muy significativas
  - Es lógico invertir tiempo en el desarrollo de un buen algoritmo sólo si se prevé que éste realizará un gran volumen de operaciones
- Al estudio de la complejidad para tamaños grandes de problema se le denomina análisis asintótico
  - Permite clasificar las funciones de complejidad de forma que podamos compararlas entre si fácilmente
  - Para ello, se definen clases de equivalencia que engloban a las funciones que “crecen de la misma forma”.
- Se emplea la notación asintótica

Notación asintótica:

- Notación matemática utilizada para representar la complejidad cuando el tamaño de problema (n) crece ($n \to \infty$)
- Se definen tres tipos de notación:
  - Notación O (ómicron mayúscula o *big omicron*) $\Rightarrow$ cota superior
  - Notación Ω (omega mayúscula o *big omega*) $\Rightarrow$ cota inferior
  - Notación Θ (zeta mayúscula o *big theta*)$\Rightarrow$ coste exacto

¿Para qué sirven?

- Permite agrupar en clases funciones con el mismo crecimiento



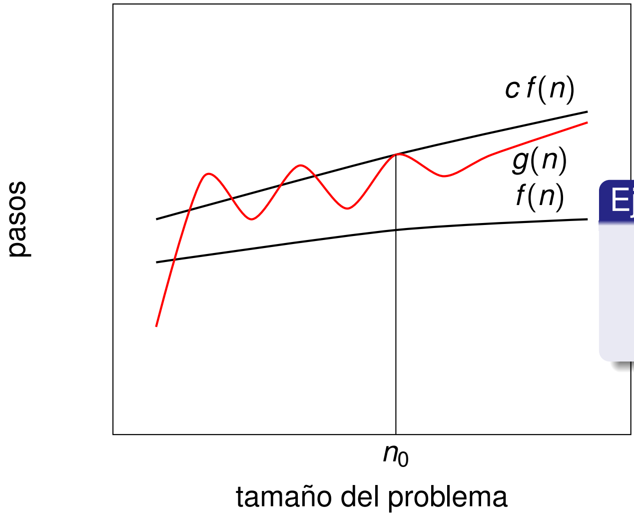
Cota superior. Notación O

- Se usa para describir un límite superior en el crecimiento de cualquier función
  - normalmente se utiliza para $c_s(n)$
- Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+$; se define el conjunto $O(f)$ como el conjunto de funciones acotadas superiormente por un múltiplo de f :

$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N}, \forall n \geqslant n_0, g(n) \leqslant cf(n)\}$$

- Dada una función $t : \mathbb{N} \rightarrow \mathbb{R}^+$ se dice que $t \in O(f)$ si existe un múltiplo de f que es cota superior de t para valores grandes de n
  - es decir, $t(n)$ no crece más rápido que $f(n)$, salvo por un factor constante c

Cota superior. Notación O



Ejemplos:

- ¿$3n + 1 \in O(n)$?
- ¿$3n^2 + 1 \in O(n)$?
- ¿$3n^2 + 2 \in O(n^2)$?

Propiedades (I)

$$
\begin{aligned}
f &\in O(f) \\
f \in O(g) &\Rightarrow O(f) \subseteq O(g) \\
O(f) = O(g) &\Leftrightarrow f \in O(g) \wedge g \in O(f) \\
f \in O(g) \wedge g \in O(h) &\Rightarrow f \in O(h) \\
f \in O(g) \wedge f \in O(h) &\Rightarrow f \in O(\text{mín}\{g, h\}) \\
f_1 \in O(g_1) \wedge f_2 \in O(g_2) &\Rightarrow f_1 + f_2 \in O(\text{máx}\{g_1, g_2\}) \\
f_1 \in O(g_1) \wedge f_2 \in O(g_2) &\Rightarrow f_1 + f_2 \in O(g_1 + g_2) \\
f_1 \in O(g_1) \wedge f_2 \in O(g_2) &\Rightarrow f_1 f_2 \in O(g_1 g_2) \\
f(n) = a_m n^m + \cdots + a_1 n + a_0 \text{ con } a_m > 0 &\Rightarrow f(n) \in O(n^m) \\
O(f) \subset O(g) &\Rightarrow f \in O(g) \wedge g \notin O(f)
\end{aligned}
$$

Ejercicio

- $f(n) = \log_2^2(n) + \log_2(n^3) + \log_3(n^2) + n\log(n) \in O(?)$
- $g(n) = 2^{log_2(n^2)} + 4^{log_2(n)} + 2^{log_2(n)} \in O(?)$
- $f(n) + g(n) \in O(?)$

Propiedades (II)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f \in O(g) \land g \notin O(f)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Leftrightarrow f \notin O(g) \land g \in O(f)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Leftrightarrow f \in O(g) \land g \in O(f)$$

¡ Atención !

$$f \in O(g) \not\Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$g \in O(f) \not\Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$$

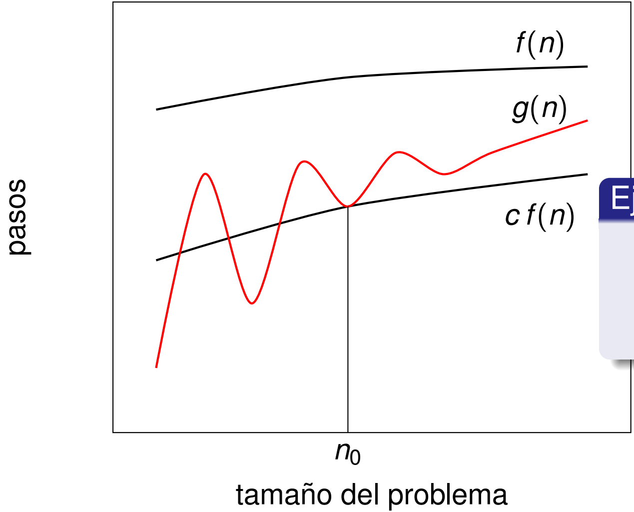
Cota inferior. Notación Ω

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $\Omega(f)$ como el conjunto de funciones acotadas inferiormente por un múltiplo de f:

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ | \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geqslant n_0, g(n) \geqslant cf(n)\}$$

- Dada una función $t : \mathbb{N} \to \mathbb{R}^+$ se dice que $t \in \Omega(f)$ si existe un múltiplo de f que es cota inferior de t para valores grandes de n

Cota inferior. Notación Ω



Ejemplos:

- ¿$3n + 1 \in \Omega(n)$?
- ¿$3n^2 + 1 \in \Omega(n)$?
- ¿$3n^2 + 2 \in \Omega(n^2)$?

Propiedades (I)

$$
\begin{aligned}
f &\in \Omega(f)\\
f \in \Omega(g) &\Rightarrow \Omega(f) \subseteq \Omega(g)\\
\Omega(f) = \Omega(g) &\Leftrightarrow f \in \Omega(g) \wedge g \in \Omega(f)\\
f \in \Omega(g) \wedge g \in \Omega(h) &\Rightarrow f \in \Omega(h)\\
f \in \Omega(g) \wedge f \in \Omega(h) &\Rightarrow f \in \Omega(\text{máx}\{g, h\})\\
f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) &\Rightarrow f_1 + f_2 \in \Omega(\text{máx}(g_1, g_2))\\
f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) &\Rightarrow f_1 + f_2 \in \Omega(g_1 + g_2)\\
f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) &\Rightarrow f_1 f_2 \in \Omega(g_1 g_2)\\
f(n) = a_m n^m + \cdots + a_1 n + a_0 & \text{ con } a_m > 0\\
&\Rightarrow f(n) \in \Omega(n^m)
\end{aligned}
$$

Ejercicio

- $f(n) = \log_2^2(n) + \log_2(n^3) + \log_3(n^2) + n\log(n) \in \Omega(?)$
- $g(n) = 2^{log_2(n^2)} + 4^{log_2(n)} + 2^{log_2(n)} \in \Omega(?)$
- $f(n) + g(n) \in \Omega(?)$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f \notin \Omega(g) \wedge g \in \Omega(f)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Leftrightarrow f \in \Omega(g) \wedge g \notin \Omega(f)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Leftrightarrow f \in \Omega(g) \wedge g \in \Omega(f)$$

¡ Atención !

$$g \in \Omega(n) \nRightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

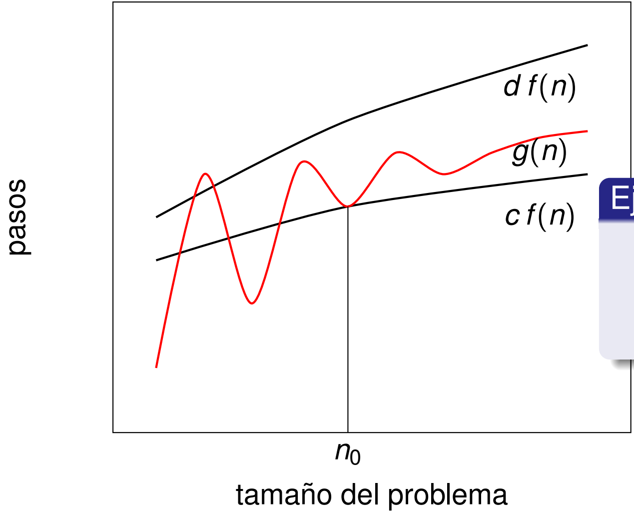
$$f \in \Omega(g) \nRightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$$

Coste exacto / cota promedio. Notación Θ

- Sea $f : \mathbb{N} \to \mathbb{R}^+$; se define el conjunto $\Theta(f)$ como el conjunto de funciones acotadas superior e inferiormente por un múltiplo de f:

$$\Theta(f) = \{g : \mathbb{N} \to R^+ | \exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \\ \forall n \geqslant n_0, cf(n) \leqslant g(n) \leqslant df(n)\}$$

- O lo que es lo mismo: $\Theta(f) = O(f) \cap \Omega(f)$
- Dada una función $t : \mathbb{N} \to \mathbb{R}^+$ se dice que $t \in \Theta(f)$ si existen múltiplos de f que son a la vez cota superior y cota inferior de t para valores grandes de n



Ejemplos:

- ¿$3n + 1 \in \Theta(n)$?
- ¿$3n^2 + 1 \in \Theta(n)$?
- ¿$3n^2 + 2 \in \Theta(n^2)$?

Propiedades (I)

$$
\begin{aligned}
f &\in \Theta(f) \\
f \in \Theta(g) &\Rightarrow \Theta(g) = \Theta(f) \\
\Theta(f) = \Theta(g) &\Leftrightarrow f \in \Theta(g) \wedge g \in \Theta(f) \\
f \in \Theta(g) \wedge g \in \Theta(h) &\Rightarrow f \in \Theta(h) \\
f \in \Theta(g) \wedge f \in \Theta(h) &\Rightarrow f \in \Theta(\text{máx}\{g, h\}) \\
f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) &\Rightarrow f_1 + f_2 \in \Theta(g_1 + g_2) \\
\Theta(f_1 + f_2) &= \Theta(\text{máx}(f_1, f_2)) \\
f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) &\Rightarrow f_1 f_2 \in \Theta(g_1 g_2) \\
f(n) = a_m n^m + \cdots + a_1 n + a_0 &\text{ con } a_m > 0 \\
&\Rightarrow f(n) \in \Theta(n^m)
\end{aligned}
$$

Ejercicio

- $f(n) = \log_2^2(n) + \log_2(n^3) + \log_3(n^2) + n\log(n) \in \Theta(?)$
- $g(n) = 2^{log_2(n^2)} + 4^{log_2(n)} + 2^{log_2(n)} \in \Theta(?)$
- $f(n) + g(n) \in \Theta(?)$

Propiedades (II)

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Leftrightarrow f \in O(g) \land f \notin \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Leftrightarrow f \in \Omega(g) \land f \notin \Theta(g)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Leftrightarrow f \in \Theta(g)$$

Jerarquías de funciones

- Los conjuntos de funciones están incluidos unos en otros generando una ordenación de las diferentes clases de complejidad.

$$\boxed{\boxed{\boxed{O(f_1(n))} \quad O(f_2(n))} \quad O(f_3(n))}$$

- Las clases más utilizadas en la expresión de complejidades son:

$$\underbrace{O(1)}_{\text{constantes}} \subset \underbrace{O(\log\log n)}_{\text{sublogarítmicas}} \subset \underbrace{O(\log n) \subset O(log^{a(>1)} n)}_{\text{logarítmicas}}$$

$$\subset \underbrace{O(\sqrt{n})}_{\text{sublineales}} \subset \underbrace{O(n)}_{\text{lineales}} \subset \underbrace{O(n\log n)}_{\text{lineal-logarítmicas}}$$

$$\subset \underbrace{O(n^2) \subset O(n^{a(>2)})}_{\text{polinómicas}} \subset \underbrace{O(2^n) \subset O(a^n{}_{(a>2)})}_{\text{exponenciales}} \subset \underbrace{O(n!) \subset O(n^n)}_{\text{superexponenciales}}$$

Ejercicios. ¿Verdadero o falso?

1. $4n^3 - 2n^2 + 8 \in O(n^3)$
2. $n^3 \in O(4n^3 - 2n^2 + 8)$
3. $n + n\sqrt{n} \in O(n)$
4. $n + n \log n \in \Theta(n)$
5. $n + n \log n \in \Omega(n)$
6. Si $f \notin O(g)$ y $\exists \operatorname{lím}_{n\to\infty} \frac{f}{g}$: $f \in \Omega(g)$
7. Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in O(g_1 + g_2)$
8. Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in \Omega(g_1 + g_2)$
9. Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in O(g_1 \cdot g_2)$
10. Si $f \in \Theta(g_1)$ y $f \in \Theta(g_2)$: $f \in \Omega(g_1 \cdot g_2)$
11. $O(g_1 + g_2) = O(\operatorname{máx}(g1, g2))$
12. $O(g_1 + g_2) = O(\operatorname{mín}(g1, g2))$
13. $O(2^{\log_2(n)}) \subset O(n^2)$
14. $O(4^{\log_2(n)}) \subset O(n^2)$
15. $\Theta(2^n) = \Theta(3^n)$
16. $\Theta(\log^2(n)) = \Theta(\log^3(n))$
17. $\Theta(\log_2(n)) = \Theta(\log_3(n))$
18. $\Theta(\log_2(n^3)) = \Theta(\log_2(n^2))$
19. $\Theta(\log_a(n)) = \Theta(\log_b(n) = \Theta(\log(n)) \;\; \forall a, b > 1$

Índice

Cálculo de complejidades

- Pasos para obtener las cotas de complejidad
  1. Determinar la talla: variable de la función de complejidad que se pretende encontrar. Puede ser:
     - Tamaño del problema
     - Parámetro de la función.
  2. Determinar los casos mejor y peor (instancias para las que el algoritmo tarda más o menos)
     - Para algunos algoritmos, el caso mejor y el caso peor son el mismo ya que se comportan igualmente para cualquier instancia del mismo tamaño
  3. Obtención de las cotas para cada caso
     - Algoritmos iterativos
     - Algoritmos recursivos

Algoritmos iterativos

Sumar elementos

```
1 int sumar( const vector<int> &v ) {
2   int s = 0;
3   for( int i = 0; i < v.size(); i++ )
4     s += v[i];
5   return s;
6 }
```

Línea	Pasos	C. Asintótica
2	1	$\Theta(1)$
3,4	n	$\Theta(n)$
5	1	$\Theta(1)$
Suma	n+2	$\Theta(n)$

Algoritmos iterativos

Buscar elemento

```
1 int buscar( const vector<int> &v, int z ) {
2   for( int i = 0; i < v.size(); i++ )
3     if( v[i] == z )
4       return i;
5   return -1;
6 }
```

Línea	Cuenta Pasos		C. Asintótica	
	Mejor caso	Peor caso	Mejor caso	Peor caso
2	1	n	$\Omega(1)$	$O(n)$
3	1	n	$\Omega(1)$	$O(n)$
4	1	0	$\Omega(1)$	–
5	0	1	–	$O(1)$
Suma	3	$2n+1$	$\Omega(1)$	$O(n)$

$$C_s(n) = 2n + 1$$
$$C_i(n) = 3$$

$$C_s(n) \in O(n)$$
$$C_i(n) \in \Omega(1)$$

Ejemplo I: Elemento máximo de un vector

Elemento máximo de un vector

```
int maximo( const vector<int> &v ) {
  int max = v[0];
  for( int i = 1; i < v.size(); i++ )
    if( v[i] > max )
      max = v[i];
  return max;
}
```

Elemento máximo de un vector

```
int maximo( const vector<int> &v ) {
  int max = v[0];
  for( int i = 1; i < v.size(); i++ )
    if( v[i] > max )
      max = v[i];
  return max;
}
```

- Cálculo dela complejidad temporal utilizando series matemáticas:
  - Tamaño del problema: $n = \texttt{v.size()}$
  - Los casos mejor y peor se diferencian en una constante
  - Coste exacto:

$$c_e(n) = 1 + \sum_{i=1}^{n-1} 1 = 1 + n - 1 \in \Theta(n)$$

Ejemplo II: Búsqueda en un vector ordenado

Búsqueda binaria

```
int buscar(const vector<int> &v,
            int x
           ) {
  int pri = 0;
  int ult = v.size() - 1;
  while( pri < ult ) {
    int m = ( pri + ult ) / 2;
    if ( v[m] < x )
      pri = m + 1;
    else
      ult = m;
  }
  if( v[pri] == x )
    return pri;
  else
    return -1;
}
```

Ejemplo II: Búsqueda en un vector ordenado

Búsqueda binaria

```
int buscar(const vector<int> &v,
            int x
           ) {
  int pri = 0;
  int ult = v.size() - 1;
  while( pri < ult ) {
    int m = ( pri + ult ) / 2;
    if ( v[m] < x )
      pri = m + 1;
    else
      ult = m;
  }
  if( v[pri] == x )
    return pri;
  else
    return -1;
}
```

- Tamaño del problema: $n = \text{ult} - \text{pri} + 1$
- No existe caso mejor y peor.
- Coste exacto:

iteración	Tamaño	Pasos
1	n	1
2	$n/2$	1
3	$n/4$	1
...	...	...
k	$n/2^{k-1} = 2^*$	1

* Despejar k para obtener el máximo valor que puede tomar.

$$k = \lfloor \log_2 n \rfloor$$

$$c_e(n) = 1 + \sum_{k=1}^{\lfloor \log_2 n \rfloor} 1 \in \Theta(\log n)$$

Ejercicio: obtener la expresión de $C_e(n)$

```
int ejemplo(int n){
  int k=0;
  for( int i = 1; i < n; i*=2 )
    for( int j = 0; j < i; j++ )
      k++;
  return k;
}
```

Errores habituales:

1. $C_e(n) = \frac{1}{2}\sum_{i=1}^{n-1} i \in \Theta(n^2)$
2. $C_e(n) = \sum_{i=1}^{\log_2 n} i \in \Theta(\log^2 n)$
3. $C_e(n) = \sum_{i=1}^{\log_2 n} \frac{i}{2} \in \Theta(i \log n)$
4. $C_e(n) = \sum_{i=1}^{n/2} 2^i \in \Theta(2^n)$
5. $C_e(n) = \sum_{i=1}^{n-1} \log_2 i \in \Theta(n \log n)$
6. $C_e(n) = \sum_{k=1}^{i} \log_2 n \in \Theta(i \log n)$

¿Porqué están todos mal?

Algoritmos recursivos

- Dado un algoritmo recursivo:

Búsqueda binaria

```
int buscar( const vector<int> &v, int pri, int ult, int x){
  if( pri == ult )
    return (v[pri] == x) ? pri : -1;
  int m = ( pri + ult ) / 2;
  if( v[m] < x )
    return buscar( v, m+1, ult, x );
  else
    return buscar( v, pri, m, x );
}
```

- El coste depende de las llamadas recursivas, y, por tanto, debe definirse recursivamente:

$$T(n) \in \begin{cases} \Theta(1) & n = 1 \\ \Theta(1) + T(n/2) & n > 1 \end{cases} \qquad (n = \mathtt{ult} - \mathtt{pri} + 1)$$

Relaciones de recurrencia

- Una relación de recurrencia es una expresión que relaciona el valor de una función f definida para un entero n con uno o más valores de la misma función para valores menores que n

$$f(n) = \begin{cases} a\,f(F(n)) + P(n) & n > n_0 \\ P'(n) & n \leqslant n_0 \end{cases}$$

Donde:

  - $a \in \mathbb{N}$ es una constante
  - $P(n), P'(n)$ son funciones de n
  - $F(n) < n$ (normalmente $n - b$ con $b > 0$, o n/b con $b \geq 1$)

Algoritmos recursivos

- Las relaciones de recurrencia se usan para expresar la complejidad de un algoritmo recursivo aunque también son aplicables a los iterativos
- Si el algoritmo dispone de mejor y peor caso, puede haber una relación de recurrencia para cada caso
- La complejidad de un algoritmo se obtiene en tres pasos:
    1. Determinación de la talla del problema
    2. Obtención de las relaciones de recurrencia del algoritmo
    3. Resolución de las relaciones
- Para resolverlas, usaremos el método de sustitución:
    - Es el método más sencillo
    - Sólo para funciones lineales (sólo una vez en función de sí mismas)
    - Consiste en sustituir cada $f(n)$ por su valor al aplicarle de nuevo la función hasta obtener un término general

Ordenación por selección

- Ejemplo: Ordenar un vector a partir del elemento `pri`:

Ordenación por selección (recursivo)

```
void ordenar( vector<int> &v, int pri) {
  if( pri == v.size() )
    return;
  int m = pri;
  for( int i = pri + 1; i < v.size(); i++ )
    if( v[i] < v[m] )
      m = i;
  swap( v[m], v[pri]);
  ordenar(v,pri + 1);
}
```

- Obtener ecuación de recurrencia a partir del algoritmo:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \Theta(n) + T(n-1) & n > 1 \end{cases}$$

donde $n = \texttt{v.size()} - \texttt{pri}$.

Ecuación de recurrencia

- Resolviendo la recurrencia por sustitución

$$
\begin{aligned}
T(n) &= n + T(n-1) \\
&= n + (n-1) + T(n-2) \\
&= n + (n-1) + (n-2) + T(n-3) \\
&= n + (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + T(1) \\
&= n + (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 \\
&= \sum_{j=1}^{n} j = \frac{n(n+1)}{2}
\end{aligned}
$$

Entonces

$$T(n) \in \Theta(n^2)$$

Algoritmo de ordenación por partición o *Quicksort*

- Elemento pivote: sirve para dividir en dos partes el vector. Su elección define variantes del algoritmo
  - Al azar
  - Primer elemento (Quicksort primer elemento)
  - Elemento central (Quicksort central)
  - Elemento mediana (Quicksort mediana)
- Pasos:
  - Elección del pivote
  - Se divide el vector en dos partes:
    - parte izquierda del pivote (elementos menores)
    - parte derecha del pivote (elementos mayores)
  - Se hacen dos llamadas recursivas. Una con cada parte del vector

Quicksort primer elemento

Quicksort

```
void quicksort( int v[], int pri, int ult ) {
  if( ult <= pri )
    return;
  int p = pri;
  int j = ult;
  while(p < j) {
    if( v[p+1] < v[p] ) {
      swap( v[p+1], v[p] );
      p++;
    } else {
      swap( v[p+1], v[j] );
      j--;
    }
  }
  quicksort(v, pri, p-1);
  quicksort(v, p+1, ult);
}
```

- Tamaño del problema: n
  - Mejor caso: subproblemas $(n/2, n/2)$

$$T(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(n) + T(\frac{n}{2}) + T(\frac{n}{2}) & n > 1 \end{cases}$$

  - Peor caso: subproblemas $(0, n-1)$ o $(n-1, 0)$

$$T(n) \in \begin{cases} \Theta(1) & n \leqslant 1 \\ \Theta(n) + T(0) + T(n-1) & n > 1 \end{cases}$$

Quicksort

- **Mejor caso**:

$$
\begin{aligned}
f(n) &= n + 2T\left(\frac{n}{2}\right) && \text{Rec. 1} \\
&= n + 2\left(\frac{n}{2} + 2f\left(\frac{n}{2^2}\right)\right) = 2n + 2^2 T\left(\frac{n}{2^2}\right) && \text{Rec. 2} \\
&= 2n + 2^2\left(\frac{n}{2^2} + 2f\left(\frac{n}{2^3}\right)\right) = 3n + 2^3 T\left(\frac{n}{2^3}\right) && \text{Rec. 3} \\
&= i\,n + 2^i T\left(\frac{n}{2^i}\right) && \text{Rec. } i
\end{aligned}
$$

La recursion termina cuando $n/2^i = 1$ por lo que habrá $i = \log_2 n$ llamadas recursivas

$$= n \log_2 n + nT(1) = n \log_2 n + n$$

Por tanto,

$$T(n) \in \Omega(n \log n)$$

Quicksort

- Peor caso:

$$\begin{aligned} T(n) &= n + T(n-1) && \text{Rec. } 1 \\ &= n + (n-1) + T(n-2) && \text{Rec. } 2 \\ &= n + (n-1) + (n-2) + T(n-3) && \text{Rec. } 3 \\ &= n + (n-1) + (n-2) + \cdots + T(n-i) && \text{Rec. } i \end{aligned}$$

La recursión termina cuando $n - i = 1$ por lo que habrá $i = n - 1$ llamadas recursivas

$$\begin{aligned} &= n + (n-1) + (n-2) + \cdots + 3 + 2 + T(1) \\ &= \sum_{j=2}^{n} j + 1 = \frac{n(n+1)}{2} \end{aligned}$$

Por tanto,

$$f(n) \in O(n^2)$$

Quicksort

```
             8
          /     \
       4           4
      / \         / \
     2   2       2   2
    / \ / \     / \ / \
    1 1 1 1     1 1 1 1
```

Caso mejor
$\Omega(n \log n)$

```
                  8
                 / \
                7   1
               / \
              6   1
             / \
            5   1
           / \
          4   1
         / \
        3   1
       / \
      2   1
     / \
    1   1
```

Peor caso
$O(n^2)$

Quicksort mediana

- En la versión anterior se cumple que el caso mejor es cuando el elemento seleccionado es la mediana
- En este algoritmo estamos forzando el caso mejor
- Obtener la mediana
  - Coste menor que $O(n \log n)$
  - Se aprovecha el recorrido para reorganizar elementos y para encontrar la mediana en la siguiente subllamada
  - Su complejidad es por tanto de $\Theta(n \log n)$

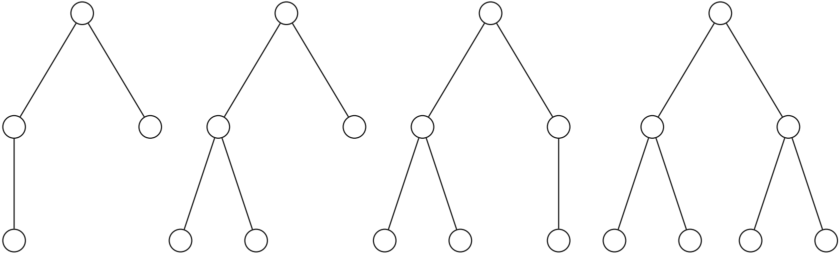
Algoritmo de ordenación por montículo o *Heapsort*

- Está basado en el algoritmo de ordenación por **selección directa**:
  - Seleccionar sucesivamente el máximo y colocarlo en la posición correcta ($\Theta(n^2)$)
- **Mejora la eficiencia de la selección repetida del máximo,** construyendo un *heap* (montículo)
- Realiza la ordenación en dos fases:
  1. Creación del montículo sobre el mismo vector a ordenar
  2. Aprovecha esa estructura para ordenar el vector:
     - Extracción sucesiva de la cima del montículo para colocarla en la posición correcta del vector.

Montículo (*heap*)

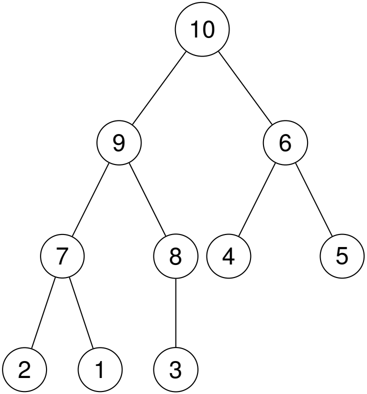
Un montículo es una estructura de datos del tipo *árbol binario* con dos características:

1. Es esencialmente completo (EC):
   - Todas las hojas están desplazadas lo más a la izquierda posible y todos los niveles están completados salvo, en todo caso, el último.
   - Por ejemplo: Estos cuatro árboles binarios son esencialmente completos:



2. Todo nodo es mayor que sus dos hijos (*max heap*)
   - También existe el *min heap*

Ejemplo de montículo máximo I

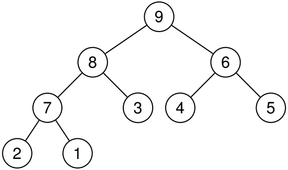


Montículo máximo:

- El valor asociado al nodo raíz es el mayor de todos.
- En cuanto al resto de elementos, no hay orden preestablecido, pero cualquier subárbol cumple lo anterior.

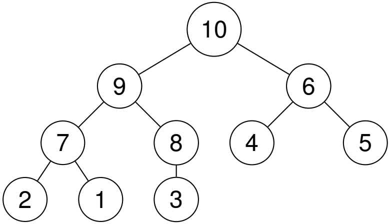
Operaciones asociadas a los montículos:

- **top** (consultar el valor de la cima): Conocer el máximo de los valores. $\Theta(1)$
- **pop** (borrar la cima): Reorganizar para que siga siendo un montículo. $O(\log n)$ y $\Omega(1)$
  Por ejemplo, la operación *pop* sobre el montículo de la izquierda, produciría:



  Se trata de colocar en la cima el último elemento de la estructura (en este caso el nodo con valor 3) y a continuación "hundirlo" (*shink*) hasta colocarlo en su posición correcta.
- **push** (insertar un elemento): Se coloca al final y se reorganiza el árbol. $O(\log n)$.
  En este caso, el elemento se "flota" (*float-up*) hasta colocarlo en su posición correcta.

Ejemplo de montículo máximo II



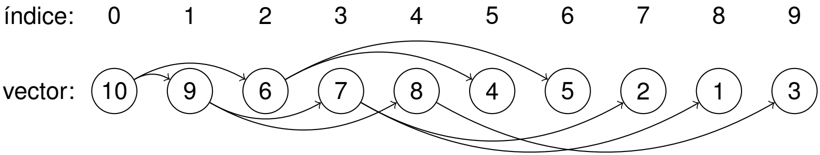
relaciones:

`root()` = 0
`left_child(i)` = 2 * i + 1
`right_child(i)` = 2 * i + 2
`parent(i)` = (i - 1) / 2

Nótese que en un árbol binario EC de tamaño n:

- El número de hojas es $(n + 1)/2$
- Si $(2 * i + 1) > n$ entonces el nodo i es un nodo hoja.

- El tipo de datos que subyace en un montículo suele ser un vector (el árbol binario es en realidad una estructura “imaginaria”)



Aplicaciones de los montículos

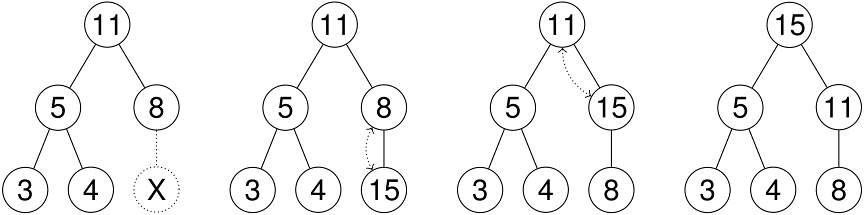
- Sirven para implementar *colas de prioridad* (`priority_queue` en las `STL` de `C++`)
  - Una estructura de datos muy utilizada en Algoritmia.
- Es la manera más eficiente conocida para implementar, de forma conjunta, las operaciones:
  - `top` ($\Theta(1)$),
  - `pop` y `push` (ambas $\Omega(1)$ y $O(\log n)$).

Inserción en un montículo (`push`)

La inserción en un montículo pasa por estas fases:

1. Añadir el elemento al final del vector con el que se representa el montículo.
2. "Flotarlo", es decir, comparar el elemento con su padre:
   1. Si es menor o igual (o es la raíz del árbol) entonces terminar;
   2. si no, intercambiarlo con su padre.
3. Volver a la fase 2.

Ejemplo: inserción de 15



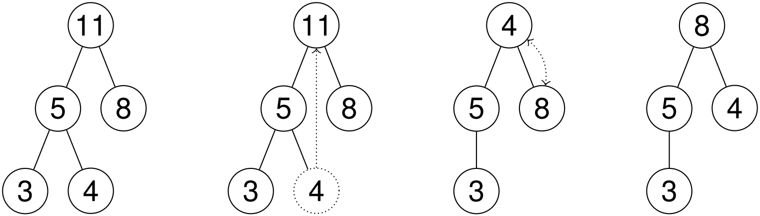
Complejidad: $O(\log(n))$ y $\Omega(1)$

Extracción de la cima -borrado- (pop)

En un montículo solo se puede borrar el nodo raíz. Pasa por estas fases:

1. Reemplazar el nodo raíz del árbol por el nodo que está en último lugar.
2. "Hundir" la nueva raíz, es decir, compararla con el mayor de sus dos hijos:
    1. Si a su vez es mayor o igual (o no tiene hijos) entonces terminar;
    2. si no, intercambiarlo con ese hijo que es mayor.
3. Volver a la fase 2.

Ejemplo: extracción



Construcción de un montículo (*Heapification*)

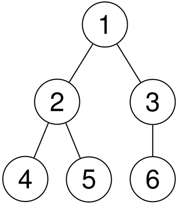
Para construir un montículo a partir de los datos almacenados en un vector v se puede proceder de dos formas:

1. Partir de un montículo vacío v' e ir insertando (`push`) cada uno de los elementos de v.
   - Cada elemento se inserta al final y se flota hasta ocupar su posición correcta.
   - Complejidad $O(n \log n)$ y $\Omega(n)$, **poco eficiente**.

2. Sin utilizar espacio adicional, asumir que el vector v es un árbol binario esencialmente completo (no hay que hacer nada, todo vector lo es).
   - "Hundir", uno a uno, los elementos del vector recorriéndolo de derecha a izquierda desde la posición $(n/2 - 1)$ (último nodo con hojas) hasta la posición 0 (nodo raiz).
     - Notar que no es necesario hundir las hojas.
     - **Complejidad** $\Theta(n)$**, eficiente**.

Construcción de un montículo. Ejemplo (I)

- Construir un montículo máximo a partir del vector $v = (1, 2, 3, 4, 5, 6)$
- Pasos:
  1. Asumir que el vector es un árbol esencialmente completo:

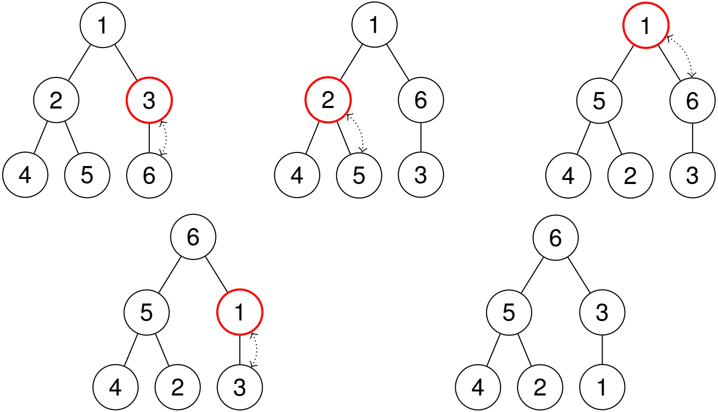


     - En realidad en este paso no hay que hacer nada, es solo una manera de interpretar el vector.
     - Para representar el árbol, se recorre el vector de izquierda a derecha; con cada uno de sus elementos se va dibujando el árbol completando todos los niveles mientras sea posible (solo el último nivel puede quedar incompleto).
  2. Hundir cada nodo del árbol desde el que ocupa la posición $(n/2 - 1)$ (en este caso el nodo con valor 3) hasta la raíz (posición 0).

Construcción de un montículo. Ejemplo (II)

2. (continuación)

```
for (int i = n / 2 - 1; i >= 0 ; i--) sink(v, n, i);
```



Finalmente $v = (6, 5, 3, 4, 2, 1)$ es un montículo máximo.

Construcción de un montículo. Complejidad temporal I

Asumimos un árbol completo con n nodos: $\exists\, p \in \mathbb{N} \mid n = 2^p - 1$.
Este árbol tiene $\frac{n+1}{2} - 1$ nodos internos.

1. Caso más favorable: El vector de entrada ya es un montículo. Ningún elemento se hunde pero se visitan todos los nodos internos. $c_i(n) = \frac{n+1}{2} - 1 \in \Omega(n)$
2. Caso más desfavorable: Todos se hunden hasta las hojas:

Nivel en el árbol	Nodos/nivel	Pasos/nodo	Total pasos en el nivel
0 (raíz)	1	$\log_2 \frac{n+1}{2}$	$1 \times \log_2 \frac{n+1}{2}$
1	2	$\log_2 \frac{n+1}{4}$	$2 \times \log_2 \frac{n+1}{4}$
2	4	$\log_2 \frac{n+1}{8}$	$4 \times \log_2 \frac{n+1}{8}$
...	...	...	...
k (hojas)	2^k	$\log_2 \frac{n+1}{2^{k+1}} = 0^*$	$2^k \times \log_2 \frac{n+1}{2^{k+1}}$

(*) El máximo valor que puede tomar k es $\log_2(n+1) - 1$. Se tiene:

$$c_s(n) = \sum_{k=0}^{\log_2(n+1)-1} 2^k \log_2\left(\frac{n+1}{2^{k+1}}\right) = n - \log_2(n+1) \in O(n)$$

Demostración:

$$c_s(n) = \sum_{k=0}^{\log_2(n+1)-1} 2^k \log_2(\frac{n+1}{2^{k+1}}) = \sum_{k=0}^{\log_2(n+1)-1} 2^k(\log_2(n+1)-(k+1)) =$$

$$= (log_2(n+1)-1) \sum_{k=0}^{\log_2(n+1)-1} 2^k - \sum_{k=0}^{\log_2(n+1)-1} k2^k.$$

Teniendo en cuenta que

$$\sum_{k=0}^{M} 2^k = 2^{M+1} - 1; \text{ y que } \sum_{k=0}^{M} k2^k = 2^{M+1}(M-1)+2,$$

es fácil comprobar, reordenando términos, que

$$c_s(n) = n - \log_2(n+1).$$

Construcción de un montículo. Compl. temporal II

- El caso más desfavorable se puede plantear de otra forma equivalente:

Altura en el árbol[1]	Nodos/nivel	Pasos/nodo	Total pasos en el nivel
0 (hojas)	$\frac{n+1}{2}$	0	$\frac{n+1}{2} \times 0$
1	$\frac{n+1}{4}$	1	$\frac{n+1}{4} \times 1$
2	$\frac{n+1}{8}$	2	$\frac{n+1}{8} \times 2$
...	...	...	...
h (raíz)	$\frac{n+1}{2^{h+1}} = 1^*$	h	$\frac{n+1}{2^{h+1}} \times h$

(*) El máximo valor que puede tomar h es $\log_2(n+1) - 1$.
Al tratarse de un árbol completo, podemos tomar $\lfloor \log_2 n \rfloor = \log_2(n+1) - 1$.
Se tiene:

$$c_s(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{(n+1)h}{2^{h+1}} = \frac{n+1}{2} \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \leq \frac{n+1}{2} \sum_{h=0}^{\infty} \frac{h}{2^h} = n+1 \in O(n).$$

ya que $\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$.

[1] Aquí se numera de hojas a raíz

Algoritmo Sink

Sink (Hundir el nodo de la posición *i* en un heap de *n* elementos)

```
void sink(int *v, size_t n, size_t i) {
  do{
    size_t largest = i;  // Initialize largest as root
    size_t l = 2*i + 1;  // left child
    size_t r = 2*i + 2;  // right child

    if (l < n && v[l] > v[largest]) // Is left child (if exists) larger than root?
      largest = l;

    if (r < n && v[r] > v[largest]) // Is right child larger than largest so far?
      largest = r;

    if (largest == i) break; // If largest is still root then stop

    // If not, swap new largest with current node i and repeat.
    swap(v[i], v[largest]);
    i=largest;
  } while (true);
}
```

Algoritmo de ordenación por montículo o Heapsort

Heapsort

```
void heapSort(int *v, size_t n) {
    // Build a MAX-HEAP with the input array
  for (size_t i = n / 2 - 1; true ; i--){
    sink(v, n, i);
    if (i==0) break; // note that size_t is unsigned type
  }

  for (size_t i=n-1; i>0; i--) {
    swap(v[0], v[i]); // Move current root to the end.
    sink(v, i, 0);
  }
}
```

¿Complejidad temporal?

Algoritmo de ordenación por montículo o `Heapsort`

Heapsort

```
1  void heapSort(int *v, size_t n) {
2      // Build a MAX-HEAP with the input array
3    for (size_t i = n / 2 - 1; true ; i--){
4      sink(v, n, i);
5      if (i==0) break; // note that size_t is unsigned type
6    }
7
8    for (size_t i=n-1; i>0; i--) {
9      swap(v[0], v[i]); // Move current root to the end.
10     sink(v, i, 0);
11   }
12 }
```

¿Complejidad temporal? Líneas 3–6, $\Theta(n)$;

Algoritmo de ordenación por montículo o `Heapsort`

`Heapsort`

```
1  void heapSort(int *v, size_t n) {
2      // Build a MAX-HEAP with the input array
3    for (size_t i = n / 2 - 1; true ; i--){
4      sink(v, n, i);
5      if (i==0) break; // note that size_t is unsigned type
6    }
7
8    for (size_t i=n-1; i>0; i--) {
9      swap(v[0], v[i]); // Move current root to the end.
10     sink(v, i, 0);
11   }
12 }
```

¿Complejidad temporal? Líneas 3–6, $\Theta(n)$; Líneas 8–11, $O(n \log(n))$.

¿Complejidad espacial?

Algoritmo de ordenación por montículo o `Heapsort`

`Heapsort`

```
1  void heapSort(int *v, size_t n) {
2      // Build a MAX-HEAP with the input array
3    for (size_t i = n / 2 - 1; true ; i--){
4      sink(v, n, i);
5      if (i==0) break; // note that size_t is unsigned type
6    }
7
8    for (size_t i=n-1; i>0; i--) {
9      swap(v[0], v[i]); // Move current root to the end.
10     sink(v, i, 0);
11   }
12 }
```

¿Complejidad temporal? Líneas 3–6, $\Theta(n)$; Líneas 8–11, $O(n\log(n))$.

¿Complejidad espacial? $\Theta(1)$