

Algorítmica

Repaso de Conceptos Algorítmicos

Conrado Martínez
U. Politècnica Catalunya

ALG Q1-2022–2023



Temario

- Parte I: Repaso de Conceptos Algorítmicos
- Parte II: Búsqueda y Ordenación Digital
- Parte III: Algoritmos Voraces
- Parte IV: Programación Dinámica
- Parte V: Flujos sobre Redes y Programación Lineal

Parte I

Repaso de Conceptos Algorítmicos

1 Análisis de Algoritmos

2 Divide y vencerás

3 Grafos y Recorridos

Parte I

Repaso de Conceptos Algorítmicos

1 Análisis de Algoritmos

2 Divide y vencerás

3 Grafos y Recorridos

- Eficiencia de un algoritmo = consumo de recursos de cómputo: tiempo de ejecución y espacio de memoria
- Análisis de algoritmos → Propiedades sobre la eficiencia de algoritmos
 - Comparar soluciones algorítmicas alternativas
 - Predecir los recursos que usará un algoritmo o ED
 - Mejorar los algoritmos o EDs existentes y guiar el diseño de nuevos algoritmos

En general, dado un algoritmo A cuyo conjunto de entradas es \mathcal{A} su **eficiencia** o **coste** (en tiempo, en espacio, en número de operaciones de E/S, etc.) es una función T de \mathcal{A} en \mathbb{N} (o \mathbb{Q} o \mathbb{R} , según el caso):

$$\begin{aligned} T : \mathcal{A} &\rightarrow \mathbb{N} \\ \alpha &\rightarrow T(\alpha) \end{aligned}$$

Ahora bien, caracterizar la función T puede ser muy complicado y además proporciona información inmanejable, difícilmente utilizable en la práctica.

Sea \mathcal{A}_n el conjunto de entradas de tamaño n y $T_n : \mathcal{A}_n \rightarrow \mathbb{N}$ la función T restringida a \mathcal{A}_n .

■ *Coste en caso mejor:*

$$T_{\text{mejor}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

■ *Coste en caso peor:*

$$T_{\text{peor}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

■ *Coste promedio:*

$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha) T_n(\alpha) \\ &= \sum_{k \geq 0} k \Pr(T_n = k). \end{aligned}$$

1 Para todo $n \geq 0$ y para cualquier $\alpha \in \mathcal{A}_n$

$$T_{\text{mejor}}(n) \leq T_n(\alpha) \leq T_{\text{peor}}(n).$$

2 Para todo $n \geq 0$

$$T_{\text{mejor}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{peor}}(n).$$

Estudiaremos generalmente sólo el coste en caso peor:

- 1 Proporciona garantías sobre la eficiencia del algoritmo, el coste **nunca** excederá el coste en caso peor
- 2 Es más fácil de calcular que el coste promedio

Una característica esencial del coste (en caso peor, en caso mejor, promedio) es su **tasa de crecimiento**

Ejemplo

- 1 Funciones lineales:

$$f(n) = a \cdot n + b \Rightarrow f(2n) \approx 2 \cdot f(n)$$

- 2 Funciones cuadráticas:

$$q(n) = a \cdot n^2 + b \cdot n + c \Rightarrow q(2n) \approx 4 \cdot q(n)$$

Se dice que las funciones lineales y las cuadráticas tienen tasas de crecimiento distintas. También se dice que son de **órdenes de magnitud** distintos.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	262144
5	32	160	1024	32768	$6,87 \cdot 10^{10}$
6	64	384	4096	262144	$4,72 \cdot 10^{21}$
...					
ℓ	N	L	C	Q	E
$\ell + 1$	$2N$	$2(L + N)$	$4C$	$8Q$	E^2

Los factores constantes y los términos de orden inferior son irrelevantes desde el punto de vista de la tasa de crecimiento: p.e. $30n^2 + \sqrt{n}$ tiene la misma tasa de crecimiento que $2n^2 + 10n \Rightarrow$ **notación asintótica**

Definición

Dada una función $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ la clase $\mathcal{O}(f)$ (O-grande de f) es

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

En palabras, una función g está en $\mathcal{O}(f)$ si existe una constante c tal que $g < c \cdot f$ para toda n a partir de un cierto punto (n_0).

Aunque $\mathcal{O}(f)$ es un conjunto de funciones por tradición se escribe a veces $g = \mathcal{O}(f)$ en vez de $g \in \mathcal{O}(f)$. Sin embargo, $\mathcal{O}(f) = g$ no tiene sentido.

Propiedades básicas de la notación \mathcal{O} :

- 1 Si $\lim_{n \rightarrow \infty} g(n)/f(n) < +\infty$ entonces $g \in \mathcal{O}(f)$
- 2 Es reflexiva: para toda función $f : \mathbb{R}^+ \rightarrow \mathbb{R}^+$, $f \in \mathcal{O}(f)$
- 3 Es transitiva: si $f \in \mathcal{O}(g)$ y $g \in \mathcal{O}(h)$ entonces $f \in \mathcal{O}(h)$
- 4 Para toda constante $c > 0$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

Los factores constantes no son relevantes en la notación asintótica y los omitiremos sistemáticamente: p.e. hablaremos de $\mathcal{O}(n)$ y no de $\mathcal{O}(4 \cdot n)$; no expresaremos la base de logaritmos ($\mathcal{O}(\log n)$), ya que podemos pasar de una base a otra multiplicando por el factor apropiado:

$$\log_c x = \frac{\log_b x}{\log_b c}$$

Otras notaciones asintóticas son Ω (omega), Θ (zita), o (little-oh) y ω (little-omega). La primera define un conjunto de funciones acotada inferiormente por una dada:

$$\Omega(f) = \{g: \mathbb{R}^+ \rightarrow \mathbb{R}^+ \mid \exists n_0 \exists c > 0 \forall n \geq n_0: g(n) \geq c \cdot f(n)\}$$

La notación Ω es reflexiva y transitiva; si $\lim_{n \rightarrow \infty} g(n)/f(n) > 0$ entonces $g \in \Omega(f)$. Por otra parte, si $f \in \mathcal{O}(g)$ entonces $g \in \Omega(f)$ y viceversa.

Se dice que $\mathcal{O}(f)$ es la clase de las funciones que crecen no más rápido que f . Análogamente, $\Omega(f)$ es la clase de las funciones que crecen no más despacio que f .

La notación $o(f)$ designa el conjunto de funciones que crecen estrictamente más despacio que f :

$$g \in o(f) \iff g \in \mathcal{O}(f) \wedge f \notin \mathcal{O}(g).$$

Ej: $n \in o(n^2)$, $n^2 \notin o(n^2)$

Recíprocamente $\omega(f)$ es el conjunto de funciones que crecen estrictamente más rápido que f :

$$g \in \omega(f) \iff g \in \Omega(f) \wedge f \notin \Omega(g).$$

Ej: $n^2 \in \omega(n)$, $n \notin \omega(n)$

- Las notaciones o y ω son transitivas pero **no** reflexivas:
 $f \notin o(f)$, $f \notin \omega(f)$.
- Si $\lim_{n \rightarrow \infty} g(n)/f(n) = 0$ entonces $g \in o(f)$; si
 $\lim_{n \rightarrow \infty} g(n)/f(n) = \infty$ entonces $g \in \omega(f)$.
- La relación entre o y ω es análoga a la de \mathcal{O} y Ω : si
 $f \in o(g)$ entonces $g \in \omega(f)$ y viceversa.

Finalmente,

$$\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$$

es la clase de las funciones con la misma tasa de crecimiento que f .

La notación Θ es reflexiva y transitiva, como las otras. Es además simétrica: $f \in \Theta(g)$ si y sólo si $g \in \Theta(f)$. Si

$\lim_{n \rightarrow \infty} g(n)/f(n) = c$ donde $0 < c < \infty$ entonces $g \in \Theta(f)$.

Propiedades adicionales de las notaciones asintóticas (las inclusiones son estrictas):

- 1 Para cualesquiera constantes $\alpha < \beta$, si f es una función creciente entonces $\mathcal{O}(f^\alpha) \subset \mathcal{O}(f^\beta)$; $f^\alpha \in o(f^\beta)$
- 2 Para cualesquiera constantes a y $b > 0$, si f es creciente, $\mathcal{O}((\log f)^a) \subset \mathcal{O}(f^b)$; $(\log f)^a \in o(f^b)$
- 3 Para cualquier constante $a \geq 0$ y $c > 1$, si f es creciente, $\mathcal{O}(f^a) \subset \mathcal{O}(c^f)$; $f^a \in o(c^f)$

Los operadores convencionales (sumas, restas, divisiones, etc.) sobre clases de funciones definidas mediante una notación asintótica se extienden de la siguiente manera:

$$A \otimes B = \{h \mid \exists f \in A \wedge \exists g \in B : h = f \otimes g\},$$

donde A y B son conjuntos de funciones. Expresiones de la forma $f \otimes A$ donde f es una función se entenderá como $\{f\} \otimes A$.

Este convenio nos permite escribir de manera cómoda expresiones como $n + \mathcal{O}(\log n)$, $n^{\mathcal{O}(1)}$, ó $\Theta(1) + \mathcal{O}(1/n)$.

Regla de las sumas:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\}).$$

Regla de los productos:

$$\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g).$$

Reglas similares se cumplen para las notaciones \mathcal{O} y Ω .

Análisis de algoritmos iterativos

- 1 El coste de una operación elemental es $\Theta(1)$.
- 2 Si el coste de un fragmento S_1 es f y el de S_2 es g entonces el coste de $S_1; S_2$ es $f + g$.
- 3 Si el coste de S_1 es f , el de S_2 es g y el coste de evaluar B es h entonces el coste en caso peor de
 if B **then** S_1
 else S_2
 end if
es $\max\{f + h, g + h\}$.

- 4 Si el coste de S durante la i -ésima iteración es f_i , el coste de evaluar B es h_i y el número de iteraciones es g entonces el coste T de

```
while  $B$  do  
     $S$   
end while
```

es

$$T(n) = \sum_{i=1}^{i=g(n)} f_i(n) + h_i(n).$$

Si $f = \max\{f_i + h_i\}$ entonces $T = \mathcal{O}(f \cdot g)$.

Análisis de algoritmos recursivos

El coste (en caso peor, medio, ...) de un algoritmo recursivo $T(n)$ satisface, dada la naturaleza del algoritmo, una ecuación **recurrente**: esto es, $T(n)$ dependerá del valor de T para tamaños menores. Frecuentemente, la recurrencia adopta una de las dos siguientes formas:

$$T(n) = a \cdot T(n - c) + g(n),$$

$$T(n) = a \cdot T(n/b) + g(n).$$

La primera corresponde a algoritmos que tiene una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con subproblemas de tamaño $n - c$, donde c es una constante. La segunda corresponde a algoritmos que tienen una parte no recursiva con coste $g(n)$ y hacen a llamadas recursivas con subproblemas de tamaño (aproximadamente) n/b , donde $b > 1$.

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n - c) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $c \geq 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$.

Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1. \end{cases}$$

Teorema

Sea $T(n)$ el coste (en caso peor, en caso medio, ...) de un algoritmo recursivo que satisface la recurrencia

$$T(n) = \begin{cases} f(n) & \text{si } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n) & \text{si } n \geq n_0, \end{cases}$$

donde n_0 es una constante, $b > 1$, $f(n)$ es una función arbitraria y $g(n) = \Theta(n^k)$ para una cierta constante $k \geq 0$.

Sea $\alpha = \log_b a$. Entonces

$$T(n) = \begin{cases} \Theta(n^k) & \text{si } \alpha < k \\ \Theta(n^k \log n) & \text{si } \alpha = k \\ \Theta(n^\alpha) & \text{si } \alpha > k. \end{cases}$$

Parte I

Repaso de Conceptos Algorítmicos

1 Análisis de Algoritmos

2 Divide y vencerás

3 Grafos y Recorridos

Introducción

El principio básico de **divide y vencerás** (en inglés, *divide and conquer*; en catalán, *dividir per conquerir*) es muy simple:

- 1 Si el ejemplar (instancia) del problema a resolver es suficientemente simple, se encuentra la solución mediante algún método directo.
- 2 En caso contrario, se *divide* o *fragmenta* el ejemplar dado en subejemplares x_1, \dots, x_k y se resuelve, independiente y recursivamente, el problema para cada uno de ellos.
- 3 Las soluciones obtenidas y_1, \dots, y_k se *combinan* para obtener la solución al ejemplar original.

- El esquema de divide y vencerás expresado en pseudocódigo tiene el siguiente aspecto:

```
procedure DIVIDE_Y_VENCERAS( $x$ )  
  if  $x$  es simple then  
    return SOLUCION_DIRECTA( $x$ )  
  else  
     $\langle x_1, x_2, \dots, x_k \rangle :=$  DIVIDE( $x$ )  
    for  $i := 1$  to  $k$  do  
       $y_i :=$  DIVIDE_Y_VENCERAS( $x_i$ )  
    end for  
    return COMBINA( $y_1, y_2, \dots, y_k$ )  
  end if  
end procedure
```

- Ocasionalmente, si $k = 1$, se habla del esquema de *reducción*.

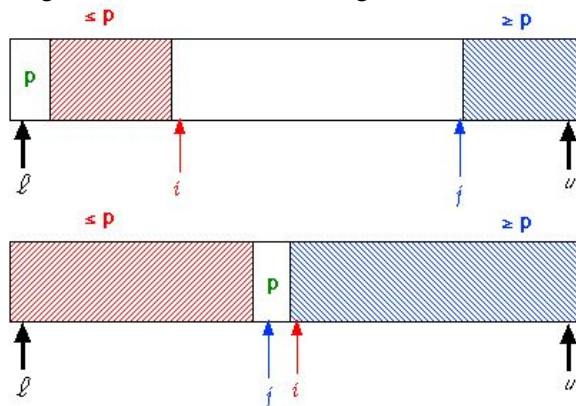
El esquema de divide y vencerás se caracteriza adicionalmente por las siguientes propiedades:

- No se resuelve más de una vez un mismo subproblema
- El tamaño de los subejemplares es, en promedio, una fracción del tamaño del ejemplar original, es decir, si x es de tamaño n , el tamaño esperado de un subejemplar cualquiera x_i es n/c_i donde $c_i > 1$. Con frecuencia, esta condición se cumplirá siempre, no sólo en promedio.

QuickSort

QUICKSORT (Hoare, 1962) es un algoritmo de ordenación que usa el principio de divide y vencerás, pero a diferencia de los ejemplos anteriores, no garantiza que cada subejemplar tendrá un tamaño que es fracción del tamaño original.

La base de *quicksort* es el procedimiento de **PARTICIÓN**: dado un elemento p denominado **pivote**, debe reorganizarse el segmento como ilustra la figura.



El procedimiento de partición sitúa a un elemento, el pivote, en su lugar apropiado. Luego no queda más que ordenar los segmentos que quedan a su izquierda y a su derecha. Mientras que en MERGESORT la división es simple y el trabajo se realiza durante la fase de combinación, en QUICKSORT sucede lo contrario.

Para ordenar el segmento $A[\ell..u]$ el algoritmo queda así

```
procedure QUICKSORT( $A, \ell, u$ )  
Ensure: Ordena el subvector  $A[\ell..u]$   
  if  $u - \ell + 1 \leq M$  then  
    usar un algoritmo de ordenación simple  
  else  
    PARTICION( $A, \ell, u, k$ )  
    ▷  $A[\ell..k - 1] \leq A[k] \leq A[k + 1..u]$   
    QUICKSORT( $(A, \ell, k - 1)$ )  
    QUICKSORT( $A, k + 1, u$ )  
  end if  
end procedure
```


En vez de usar un algoritmo de ordenación simple (p.e. ordenación por inserción) con cada segmento de M o menos elementos, puede ordenarse mediante el algoritmo de inserción al final:

QUICKSORT($A, 1, A.SIZE()$)

INSERTSORT($A, 1, A.SIZE()$)

Puesto que el vector A está quasi-ordenado tras aplicar QUICKSORT, el último paso se hace en tiempo $\Theta(n)$, donde $n = A.SIZE()$.

Se estima que la elección óptima para el umbral o corte de recursión M oscila entre 20 y 25.

Existen muchas formas posibles de realizar la partición. En Bentley & McIlroy (1993) se discute un procedimiento de partición muy eficiente, incluso si hay elementos repetidos. Aquí examinamos un algoritmo básico, pero razonablemente eficaz.

Se mantienen dos índices i y j de tal modo que $A[\ell + 1..i - 1]$ contiene elementos menores o iguales que el pivote p , y $A[j + 1..u]$ contiene elementos mayores o iguales. Los índices barren el segmento (de izquierda a derecha, y de derecha a izquierda, respectivamente), hasta que $A[i] > p$ y $A[j] < p$ o se cruzan ($i = j + 1$).

procedure PARTICION(A, ℓ, u, k)

Require: $\ell \leq u$

Ensure: $A[\ell..k-1] \leq A[k] \leq A[k+1..u]$

$i := \ell + 1; j := u; p := A[\ell]$

while $i < j + 1$ **do**

while $i < j + 1 \wedge A[i] \leq p$ **do**

$i := i + 1$

end while

while $i < j + 1 \wedge A[j] \geq p$ **do**

$j := j - 1$

end while

if $i < j + 1$ **then**

$A[i] := A[j]$

end if

end while

$A[\ell] := A[j]; k := j$

end procedure

El coste de QUICKSORT en caso peor es $\Theta(n^2)$ y por lo tanto poco atractivo en términos prácticos. Esto ocurre si en todos o la gran mayoría de los casos uno de los subsegmentos contiene muy pocos elementos y el otro casi todos, p.e. así sucede si el vector está ordenado creciente o decrecientemente. El coste de la partición es $\Theta(n)$ y entonces tenemos

$$\begin{aligned}Q(n) &= \Theta(n) + Q(n-1) + Q(0) \\&= \Theta(n) + Q(n-1) = \Theta(n) + \Theta(n-1) + Q(n-2) \\&= \dots = \sum_{i=0}^n \Theta(i) = \Theta\left(\sum_{0 \leq i \leq n} i\right) \\&= \Theta(n^2).\end{aligned}$$

Sin embargo, en promedio, el pivote quedará más o menos centrado hacia la mitad del segmento como sería deseable —justificando que *quicksort* sea considerado un algoritmo de divide y vencerás.

Para analizar el comportamiento de QUICKSORT sólo importa el orden relativo de los elementos. También podemos investigar exclusivamente el número de comparaciones entre elementos, ya que el coste total es proporcional a dicho número.

Supongamos que cualquiera de los $n!$ ordenes relativos posibles tiene idéntica probabilidad, y sea q_n el número medio de comparaciones.

$$\begin{aligned} q_n &= \sum_{1 \leq j \leq n} \mathbb{E}[\# \text{ compar.} \mid \text{pivote es } j\text{-ésimo}] \times \Pr\{\text{pivote es } j\text{-ésimo}\} \\ &= \sum_{1 \leq j \leq n} (n - 1 + q_{j-1} + q_{n-j}) \times \frac{1}{n} \\ &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq j \leq n} (q_{j-1} + q_{n-j}) \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq j < n} q_j \end{aligned}$$

Para resolver esta recurrencia emplearemos el denominado *continuous master theorem* (CMT).

El CMT considera recurrencias de divide y vencerás con el siguiente formato

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j, \quad n \geq n_0$$

para un entero positivo n_0 , una función t_n , denominada *función de peaje*, y unos pesos $\omega_{n,j} \geq 0$. Los pesos deben satisfacer dos condiciones adicionales:

- 1 $W_n = \sum_{0 \leq j < n} \omega_{n,j} \geq 1$
- 2 $Z_n = \sum_{0 \leq j < n} \frac{j}{n} \cdot \frac{\omega_{n,j}}{W_n} < 1.$

El paso fundamental es hallar una *función de forma* $\omega(z)$ que aproxima los pesos $\omega_{n,j}$.

Definición

Dado un conjunto de pesos $\omega_{n,j}$, $\omega(z)$ es una función de forma para el conjunto de pesos si

1 $\int_0^1 \omega(z) dz \geq 1$

2 existe una constante $\rho > 0$ tal que

$$\sum_{0 \leq j < n} \left| \omega_{n,j} - \int_{j/n}^{(j+1)/n} \omega(z) dz \right| = \mathcal{O}(n^{-\rho})$$

Un método simple y que funciona usualmente para calcular funciones de forma consiste en sustituir j por $z \cdot n$ en $\omega_{n,j}$, multiplicar por n y tomar el límite para $n \rightarrow \infty$.

$$\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n}$$

Las extensiones a los números reales de muchas funciones discretas son inmediatas, p.e. $j^2 \rightarrow z^2$.

Para los números binomiales se puede emplear la aproximación

$$\binom{z \cdot n}{k} \sim \frac{(z \cdot n)^k}{k!}.$$

La extensión de los factoriales a los reales viene dada por la función gamma de Euler $\Gamma(z)$ y la de los números armónicos es la función $\Psi(z) = \frac{d \ln \Gamma(z)}{dz}$.

Por ejemplo, en quicksort los pesos son todos iguales:

$\omega_{n,j} = \frac{2}{n}$. La función de forma correspondiente es

$\omega(z) = \lim_{n \rightarrow \infty} n \cdot \omega_{n,z \cdot n} = 2$.

Teorema (Roura, 1997)

Sea F_n descrita por la recurrencia

$$F_n = t_n + \sum_{0 \leq j < n} \omega_{n,j} F_j,$$

sea $\omega(z)$ una función de forma correspondiente a los pesos $\omega_{n,j}$, y $t_n = \Theta(n^a(\log n)^b)$, para $a \geq 0$ y $b > -1$ constantes.

*Sea $\mathcal{H} = 1 - \int_0^1 \omega(z) z^a dz$ y $\mathcal{H}' = -(b+1) \int_0^1 \omega(z) z^a \ln z dz$.
Entonces*

$$F_n = \begin{cases} \frac{t_n}{\mathcal{H}} + o(t_n) & \text{si } \mathcal{H} > 0, \\ \frac{t_n}{\mathcal{H}'} \ln n + o(t_n \log n) & \text{si } \mathcal{H} = 0 \text{ y } \mathcal{H}' \neq 0, \\ \Theta(n^a) & \text{si } \mathcal{H} < 0, \end{cases}$$

donde $x = \alpha$ es la única solución real no negativa de la ecuación

$$1 - \int_0^1 \omega(z) z^x dz = 0.$$

Consideremos q_n . Ya hemos visto que los pesos son $\omega_{n,j} = 2/n$ y $t_n = n - 1$. Por tanto $\omega(z) = 2$, $a = 1$ y $b = 0$. Puede comprobarse fácilmente que se dan todas las condiciones necesarias para la aplicación del CMT. Calculamos

$$\mathcal{H} = 1 - \int_0^1 2z \, dz = 1 - z^2 \Big|_{z=0}^{z=1} = 0,$$

por lo que tendremos que aplicar el caso 2 del CMT y calcular \mathcal{H}'

$$\mathcal{H}' = - \int_0^1 2z \ln z \, dz = \frac{z^2}{2} - z^2 \ln z \Big|_{z=0}^{z=1} = \frac{1}{2}.$$

Por lo tanto,

$$\begin{aligned} q_n &= \frac{n \ln n}{1/2} + o(n \log n) = 2n \ln n + o(n \log n) \\ &= 1,386 \dots n \log_2 n + o(n \log n). \end{aligned}$$

QuickSelect

El problema de la selección consiste en hallar el j -ésimo de entre n elementos dados. En concreto, dado un vector A con n elementos y un rango j , $1 \leq j \leq n$, un algoritmo de selección debe hallar el j -ésimo elemento en orden ascendente. Si $j = 1$ entonces hay que encontrar el mínimo, si $j = n$ entonces hay que hallar el máximo, si $j = \lfloor n/2 \rfloor$ entonces debemos hallar la mediana, etc.

Es fácil resolver el problema con coste $\Theta(n \log n)$ ordenando previamente el vector y con coste $\Theta(j \cdot n)$, recorriendo el vector y manteniendo los j elementos menores de entre los ya examinados. Con las estructuras de datos apropiadas puede rebajarse el coste a $\Theta(n \log j)$, lo cual no supone una mejora sobre la primera alternativa si $j = \Theta(n)$.

QUICKSELECT (Hoare, 1962), también llamado FIND y *one-sided* QUICKSORT, es una variante del algoritmo QUICKSORT para la selección del j -ésimo de entre n elementos.

Supongamos que efectuamos una partición de un subvector $A[\ell..u]$, conteniendo los elementos ℓ -ésimo a u -ésimo de A , y tal que $\ell \leq j \leq u$, respecto a un pivote p . Una vez finalizada la partición, supongamos que el pivote acaba en la posición k . Por tanto, en $A[\ell..k-1]$ están los elementos ℓ -ésimo a $(k-1)$ -ésimo de A y en $A[k+1..u]$ están los elementos $(k+1)$ -ésimo a u -ésimo. Si $j = k$ hemos acabado ya que hemos encontrado el elemento solicitado. Si $j < k$ entonces procedemos recursivamente en el subvector de la izquierda $A[\ell..k-1]$ y si $j > k$ entonces encontraremos el elemento buscado en el subvector $A[k+1..u]$.

procedure QUICKSELECT(A, ℓ, j, u)

Ensure: Retorna el $(j + 1 - \ell)$ -ésimo menor elemento de $A[\ell..u]$, $\ell \leq j \leq u$

if $\ell = u$ **then**

return $A[\ell]$

end if

 PARTICION(A, ℓ, u, k)

if $j = k$ **then**

return $A[k]$

end if

if $j < k$ **then**

return QUICKSELECT($A, \ell, j, k - 1$)

else

return QUICKSELECT($A, k + 1, j, u$)

end if

end procedure

Puesto que QUICKSELECT es recursiva final es muy simple obtener una versión iterativa eficiente que no necesita espacio auxiliar.

En caso peor, el coste de QUICKSELECT es $\Theta(n^2)$. Sin embargo, su coste promedio es $\Theta(n)$ donde la constante de proporcionalidad depende del cociente j/n . Knuth (1971) ha demostrado que $C_n^{(j)}$, el número medio de comparaciones necesarias para seleccionar el j -ésimo de entre n es:

$$C_n^{(j)} = 2((n+1)H_n - (n+3-j)H_{n+1-j} - (j+2)H_j + n+3)$$

El valor máximo se alcanza para $j = \lfloor n/2 \rfloor$; entonces $C_n^{(j)} = 2(\ln 2 + 1)n + o(n)$.

Consideremos ahora el análisis del coste promedio C_n suponiendo que j adopta cualquier valor entre 1 y n con igual probabilidad.

$$C_n = n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \mathbb{E}[\text{núm. de comp.} \mid \text{el pivote es el } k\text{-ésimo}] ,$$

puesto que el pivote es el k -ésimo con igual probabilidad para toda k .

La probabilidad de que $j = k$ es $1/n$ y entonces ya habremos acabado. La probabilidad de continuar a la izquierda es $(k - 1)/n$ y entonces se habrán de hacer C_{k-1} comparaciones. Análogamente, con probabilidad $(n - k)/n$ se continuará en el subvector a la derecha y se harán C_{n-k} comparaciones.

$$\begin{aligned} C_n &= n + \mathcal{O}(1) + \frac{1}{n} \sum_{1 \leq k \leq n} \frac{k-1}{n} C_{k-1} + \frac{n-k}{n} C_{n-k} \\ &= n + \mathcal{O}(1) + \frac{2}{n} \sum_{0 \leq k < n} \frac{k}{n} C_k. \end{aligned}$$

Aplicando el CMT con la función de forma

$$\lim_{n \rightarrow \infty} n \cdot \frac{2}{n} \frac{z \cdot n}{n} = 2z$$

obtenemos $\mathcal{H} = 1 - \int_0^1 2z^2 dz = 1/3$ y $C_n = 3n + o(n)$.

Algoritmo de selección de Rivest y Floyd



Robert W. Floyd (1936–2001) Ronald L. Rivest (1947–)

En 1970 Floyd y Rivest diseñaron un algoritmo de selección con coste lineal en caso peor; solo resulta necesario garantizar que el pivote elegido en cada paso de quickselect divide el vector en dos subvectores cada uno de los cuales contiene un fracción del tamaño original n . Dicho pivote debe obtenerse en tiempo lineal. Entonces, en caso peor, el coste del algoritmo es

$$C(n) = \mathcal{O}(n) + C(p \cdot n),$$

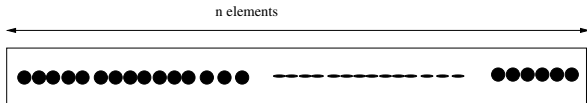
para algún valor $p < 1$. Puesto $\log_{1/p} 1 = 0 < 1$ concluimos que $C(n) = \mathcal{O}(n)$. Por otro lado, obviamente $C(n) = \Omega(n)$, y por lo tanto $C(n) = \Theta(n)$.

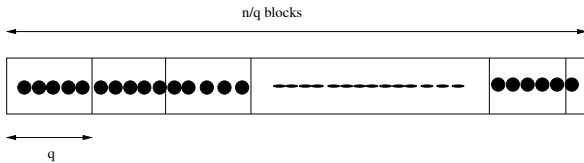
La única diferencia entre el algoritmo QUICKSELECT de Hoare y el algoritmo de Floyd y Rivest reside en la elección del pivote en cada fase recursiva.

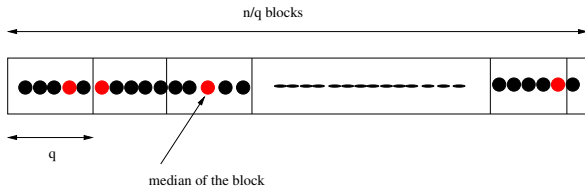
¡Y el algoritmo de Floyd y Rivest obtiene un “buen” pivote usando el propio algoritmo recursivamente!

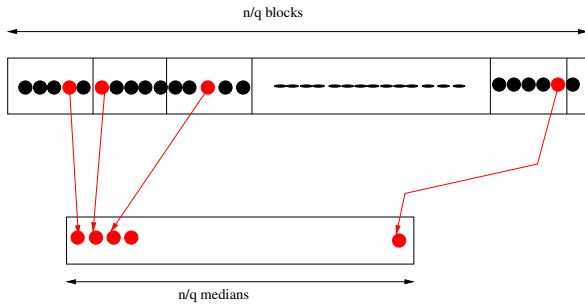
En concreto lo que hace el algoritmo es elegir la denominada **pseudo-mediana** del vector como pivote.

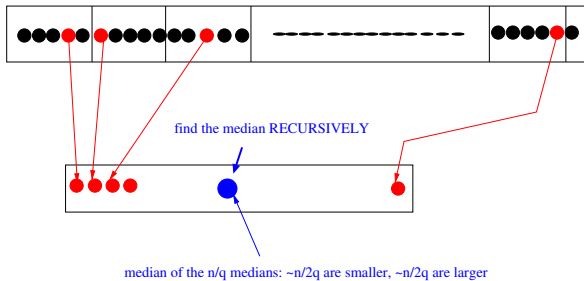
- 1 El subvector $A[\ell..u]$ de tamaño $n = u - \ell + 1$ se subdivide en bloques de q elementos (excepto posiblemente el último bloque, que puede contener $< q$ elementos), para alguna constante impar q . Para cada bloque obtenemos la mediana de sus q elementos.
- 2 Aplicamos el algoritmo recursivamente sobre las $\lceil n/q \rceil$ medianas obtenidas en el paso previo para hallar la mediana de las medianas. Este elemento no es, en general, la mediana del subvector original, por eso se le denomina *pseudo-mediana*.
- 3 La pseudo-mediana se utiliza como pivote para particionar el subvector original, y se hace una llamada recursiva en el subvector apropiado, a la izquierda o a la derecha del pivote, según el rango buscado—exactamente como en quickselect.

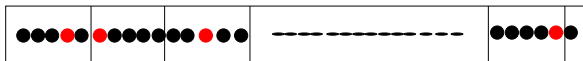













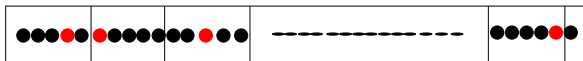





partition the array using 

 $\geq n/2q$ 

 $\geq q/2$ 



partition the array using 



- 1 El coste de la primera fase (calcular las medianas de los bloques) es $\Theta(n)$, ya que hallar la mediana de cada bloque es $\Theta(q) = \Theta(1)$ y hay $\lceil n/q \rceil$ bloques.
- 2 La llamada recursiva para encontrar la mediana de las medianas: $C(n/q)$, la entrada consiste en $\lceil n/q \rceil$ medianas.
- 3 Está garantizado que hay $\approx n/2q \cdot q/2 = n/4$ elementos menores que el pivote; análogamente, hay $\approx n/4$ elementos mayores que el pivote. En caso peor, después de la partición (coste $\Theta(n)$) proseguiremos en un subvector con $\leq 3n/4$ elementos.

Poniendo todo junto, el coste en caso peor satisface

$$C(n) = \mathcal{O}(n) + C(n/q) + C(3n/4),$$

donde el término $\mathcal{O}(n)$ incluye los costes de calcular las medianas de los bloques y de particionar el vector original. La solución de la recurrencia (se puede hacer con un *Discrete Master Theorem* no explicado en ALG) es $C(n) = \mathcal{O}(n)$ si

$$\frac{3}{4} + \frac{1}{q} < 1.$$

Por ejemplo, $q = 3$ no servirá, pero $q = 5$, el valor originalmente propuesto por Floyd y Rivest, sí.

Parte I

Repaso de Conceptos Algorítmicos

1 Análisis de Algoritmos

2 Divide y vencerás

3 Grafos y Recorridos

Definición

Un **grafo (no dirigido)** es un par $G = \langle V, E \rangle$ donde V es un conjunto finito de **vértices** (también llamados **nodos**) y E es un conjunto de **aristas**; cada arista $e \in E$ es un par no ordenado $\{u, v\}$ donde u y v ($u \neq v$) son elementos de V .

Definición

Un **grafo dirigido** o **digrafo** es un par $G = \langle V, E \rangle$ donde V es un conjunto finito de **vértices** o **nodos** y E es un conjunto de **arcos**; cada arco $e \in E$ es un par (u, v) donde u y v ($u \neq v$) son elementos de V .

Si en vez de un conjunto de arcos o aristas tenemos multiconjuntos de arcos o aristas, y se permiten **bucles** (esto es, arcos de la forma (u, u) o aristas $\{u, u\}$) entonces tenemos **multigrafos** (dirigidos y no dirigidos, respectivamente).

Dado un arco $e = (u, v)$ se denomina **origen** a u y **destino** a v . Se dice que u es un **predecesor** de v y que v es un **sucesor** de u . Para una arista $e = \{u, v\}$ se dice que u y v son sus **extremos**, que la arista es **incidente** a u y v , que u y v son **adyacentes**.

Definición

Un **camino** $P = v_0, v_1, v_2, \dots, v_n$ de longitud n en un grafo $G = \langle V, E \rangle$ es una secuencia de vértices de G tal que para toda i , $0 \leq i < n$

$$\{v_i, v_{i+1}\} \in E$$

El vértice v_0 es el origen del camino P y v_n es su destino.

Un camino en un digrafo se define de manera análoga: para cada i , (v_i, v_{i+1}) es un arco del digrafo.

Se dice que un camino es **simple** si no se repite ningún vértice.

Un camino en el que $v_0 = v_n$ es un **ciclo**.

Definición

Un grafo $G = \langle V, E \rangle$ es **conexo** si y sólo si, para todo par de vértices u y v , existe un camino que va de u a v en G .

Para digrafos la definición es idéntica, sólo que entonces se dice que el digrafo es **fuertemente conexo**.

Definición

Dado un grafo $G = \langle V, E \rangle$, el grafo $H = \langle V', E' \rangle$ se dice que es un **subgrafo** de G si y sólo si $V' \subseteq V$, $E' \subseteq E$, y para toda arista $e' = (u', v') \in E'$ se cumple que u' y v' pertenecen a V' .

Si E' contiene todas las aristas de E que son incidentes a dos vértices de V' , se dice que H es el subgrafo **inducido** por V' .

La definición de subgrafo de un grafo dirigido es completamente análoga.

Definición

Una **componente conexa** C de un grafo G es un subgrafo inducido maximal conexo de G . Maximal quiere decir que si se añade cualquier vértice v a $V(C)$ entonces el subgrafo inducido correspondiente no es conexo—en particular, no existe camino entre v y los restantes vértices de C .

En el caso de digrafos, se habla de **componentes fuertemente conexas**: una componente fuertemente conexa del digrafo G es un subgrafo dirigido inducido maximal fuertemente conexo.

Definición

Un grafo conexo y sin ciclos se denomina **árbol (libre)**.

Si G es un grafo conexo, un subgrafo $T = \langle V, E' \rangle$ (es decir que contiene los mismos vértices que G) es un **árbol de expansión** si T es un árbol.

Los árboles libres no tienen raíz, y no existe orden entre los vértices adyacentes a un vértice dado del árbol.

Lema

Si $G = \langle V, E \rangle$ es un árbol entonces $|E| = |V| - 1$.

A menudo trabajaremos con grafos o digrafos con **etiquetas** en las aristas o arcos: el etiquetado de un grafo o digrafo es una función $\phi : E \rightarrow \mathcal{L}$ entre el conjunto de aristas o arcos de G y el conjunto (eventualmente infinito) de etiquetas \mathcal{L} .

Cuando las etiquetas son números (enteros, racionales, reales), se dice que el grafo o digrafo es **ponderado** y a la etiqueta $\phi(e)$ de una arista o arco e se le suele denominar **peso**.

Implementación

Los grafos se implementan habitualmente de una de las dos siguientes formas:

- 1 Mediante **matrices de adyacencia**: la componente $A[i, j]$ de la matriz nos dice si el arco entre los vértices i y j existe o no; eventualmente, $A[i, j]$ puede contener también el peso del arco entre los vértices i y j
- 2 Mediante **listas de adyacencia**: tenemos una tabla T de listas enlazadas; la lista $T[i]$ es la lista de sucesores del vértice i

La implementación mediante matrices de adyacencia es muy costosa en espacio: si $|V| = n$, se requiere espacio $\Theta(n^2)$ para representar el grafo, independientemente del número de aristas/arcos que haya en el grafo. Sólo es interesante si necesitamos poder decidir la existencia (o no) de una arista o arco con máxima eficiencia.

Por regla general, la implementación que se usará es la de listas de adyacencia. El espacio que se requiere es $\Theta(n + m)$, donde $m = |E|$ y $n = |V|$; el espacio utilizado es por lo tanto lineal respecto al tamaño del grafo.

```

typedef int vertex;
typedef pair<vertex, vertex> edge;
typedef list<edge>::iterator edge_iter;

class Graph {
// grafos no dirigidos con V = {0, ..., n-1}
public:
// crea un grafo vacío (sin vértices y sin aristas)
    Graph();

// crea un grafo con n vértices y sin aristas
    Graph(int n);

// añade un vértice al grafo; el nuevo vértice tendrá
// identificador n, donde n era el número de vértices
// del grafo antes de añadir el vértice
    void add_vertex();

// añade la arista (u,v) o e = (u,v) al grafo
    void add_edge(vertex u, vertex v);
    void add_edge(edge e);

// consultoras del numero de vertices y de aristas
    int nr_vertices() const;
    int nr_edges() const;

// devuelve lista de adyacentes a un vertice
    list<edge> adjacent(vertex u) const;
    ...
private:
    int n, m;
    vector<list<edge> > T;
    ...
}

```


Recorridos

Definición

Dado un grafo conexo $G = \langle V, E \rangle$, un **recorrido** del grafo es una secuencia que contiene todos y cada uno de los vértices en $V(G)$ exactamente una vez y tal que para cualquier vértice v en la secuencia se cumple que, o bien v es el primer vértice de la secuencia, o bien existe una arista $e \in E(G)$ que une al vértice v con otro vértice que le precede en la secuencia.

El recorrido de un grafo es una secuencia de los recorridos de las componentes conexas de G . No hay ningún vértice que aparezca en el recorrido sin que se haya completado el recorrido de las componentes conexas correspondientes a los vértices que le preceden en el recorrido.

En el caso de los digrafos, un recorrido que comienza en un cierto vértice v visitará todos y cada uno de los vértices accesibles desde v exactamente una vez; la visita de un vértice $w \neq v$ implica que existe algún otro vértice visitado previamente del cual w es sucesor.

Los recorridos nos permiten visitar todos los vértices de un grafo, con arreglo a la topología del grafo, pues se utilizan las aristas o arcos del grafo para ir haciendo las sucesivas visitas.

Mediante un recorrido (o una combinación de recorridos) podemos resolver eficientemente numerosos problemas sobre grafos. Por ejemplo

- Determinar si un grafo es conexo o no
- Hallar las componentes conexas de un grafo no dirigido
- Hallar las componentes fuertemente conexas de un grafo dirigido
- Determinar si un grafo contiene ciclos o no
- Decidir si un grafo es 2-coloreable, equivalentemente, si es bipartido
- Decidir si un grafo es biconexo o no (la eliminación de un vértice no desconecta al grafo)
- Hallar el camino más corto (con menor número de aristas/arcos) entre dos vértices dados
- etc.

Recorrido en profundidad (DFS)

En el **recorrido en profundidad** (ing: *Depth-First Search*, DFS) de un grafo G , se visita un vértice v y desde éste, recursivamente, se realiza el recorrido de cada uno de los vértices adyacentes/sucesores de v no visitados. Cuando se visita por vez primera un vértice v se dice que se ha **abierto** el vértice; el vértice se **cierra** cuando se completa, recursivamente, el recorrido en profundidad de los vértices adyacentes/sucesores.

La numeración **directa** o **numeración DFS** de un vértice es el número de orden en el que se abre el vértice; así, el vértice en el que se inicia el recorrido tiene numeración directa 1.

La numeración **inversa** de un vértice es el número de orden en el que se cierra el vértice. El primer vértice del recorrido para el cual todos sus vecinos/sucesores ya han sido visitados tiene numeración inversa 1, y así sucesivamente.

▷ *visitado*, *ndfs*, *ninv*, *num_dfs*, *num_inv* son variables globales

```
procedure DFS(G)  
  for  $v \in V(G)$  do  
    visitado[v] := false  
    ndfs[v] := 0; ninv[v] := 0  
  end for  
  num_dfs := 0; num_inv := 0  
  for  $v \in V(G)$  do  
    if  $\neg \text{visitado}[v]$  then  
      DFS-REC(G, v, v)  
    end if  
  end for  
end procedure
```

```

procedure DFS-REC( $G, v, padre$ )
  PRE-VISIT( $v$ )
   $visitado[v] := \text{true}$ 
   $num\_dfs := num\_dfs + 1; ndfs[v] := num\_dfs$ 
  for  $w \in G.ADJACENT(v)$  do
    if  $\neg visitado[w]$  then
      PRE-VISIT-EDGE( $v, w$ )
      DFS-REC( $G, w, v$ )
      POST-VISIT-EDGE( $v, w$ )
    else
       $\triangleright$  si  $w \neq padre$  entonces hemos encontrado un ciclo
    end if
  end for
  POST-VISIT( $v$ )
   $num\_inv := num\_inv + 1; ninu[v] := num\_inv$ 
end procedure

```

```
// Macros para escribir recorridos en un grafo
#define forall(v,G) for(vertex (v) = 0; (v) < (G).nr_vertices(); ++(v))

#define forall_adj(e, u, G) \
    for(edge_iter (e) = (G).adjacent((u)).begin(); \
        (e) != (G).adjacent((u)).end() ; ++(e))

#define target(eit) ((eit) -> second)
#define source(eit) ((eit) -> first)
```



```
void DFS(const Graph& G) {  
    vector<bool> visitado(G.nr_vertices(), false);  
    vector<int> ndfs(G.nr_vertices(), 0);  
    vector<int> ninv(G.nr_vertices(), 0);  
    int num_dfs = 0;  
    int num_inv = 0;  
  
    forall(v, G)  
        if (not visitado[v])  
            DFS(G, v, v, visitado, num_dfs, num_inv, ndfs, ninv);  
}
```

```

void DFS(const Graph& G, vertex v, vertex padre,
        vector<bool>& visitado,
        int& num_dfs, int& num_inv,
        vector<int>& ndfs,
        vector<int>& ninv) {

    PRE-VISIT(v);
    visitado[v] = true;
    ++num_dfs; ndfs[v] = num_dfs;
    forall_adj(e, v, G) {
        vertex w = target(e);
        if (not visitado[w]) {
            PRE-VISIT-EDGE(v,w);
            DFS(G, w, v, visitado, num_dfs, num_inv, ndfs, ninv);
            POST-VISIT-EDGE(v,w);
        } else {
            // si w != padre entonces hemos encontrado un ciclo
        }
    }
    POST-VISIT(v);
    ++num_inv; ninv[v] = num_inv;
}

```

El recorrido en profundidad de una componente conexa induce un árbol de expansión, al que se denomina árbol del recorrido T_{DFS} .

Las aristas de la componente conexa se clasifican entonces en **aristas de árbol** (*tree edges*) y **aristas de retroceso** (*backward edges*). Éstas últimas unen el vértice que se acaba de abrir con un vértice visitado (abierto o cerrado) y por tanto cierran ciclos. El recorrido de todo el grafo da lugar a un bosque de expansión.

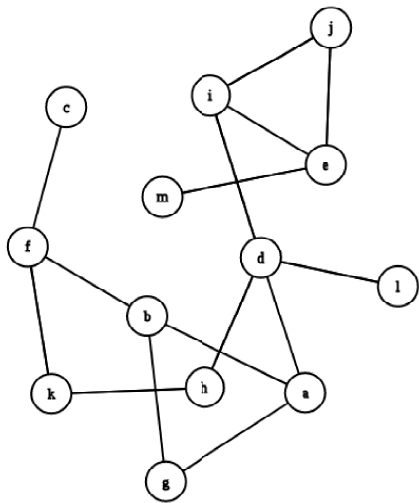
Para cada vértice v , $CC[v]$ será el número de la componente conexa en la que está v . $TreeEdges[i]$ es el conjunto de aristas de árbol en la componente i , y $BackEdges[i]$ el conjunto de aristas de retroceso. El número de componentes conexas viene dado por ncc .

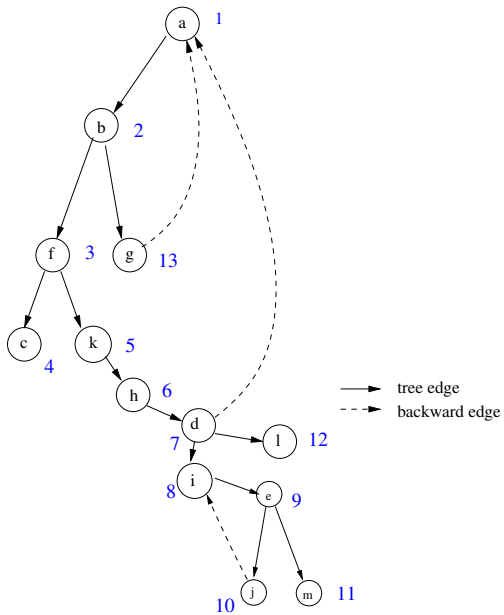
```
procedure DFS( $G$ )  
  for  $v \in V(G)$  do  
     $visitado[v] := \text{false}$   
     $CC[v] := 0$   
  end for  
   $ncc := 0$   
  for  $v \in V(G)$  do  
    if  $\neg visitado[v]$  then  
       $ncc := ncc + 1$   
       $TreeEdges[ncc] := \emptyset$   
       $BackEdges[ncc] := \emptyset$   
      DFS-REC( $G, v, v$ )  
    end if  
  end for  
end procedure
```

```

procedure DFS-REC( $G, v, padre$ )
   $visitado[v] := \mathbf{true}$ 
   $CC[v] := ncc$ 
  for  $w \in G.ADJACENT(v)$  do
    if  $\neg visitado[w]$  then
       $TreeEdges[ncc] := TreeEdges[ncc] \cup \{(v, w)\}$ 
      DFS-REC( $G, w, v$ )
    else if  $w \neq padre$  then
       $BackEdges[ncc] := BackEdges[ncc] \cup \{(v, w)\}$ 
    end if
  end for
end procedure

```

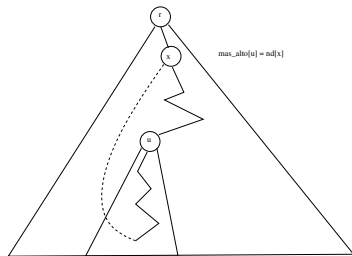




Aplicaciones del DFS: Grafos Biconexos

Un grafo conexo G se dice que es **biconexo** si al eliminar uno cualquiera de sus vértices sigue siendo conexo.

Supongamos que efectuamos un DFS empezando en el vértice r y que para cada vértice u determinamos el número DFS más bajo que es alcanzable siguiendo un camino desde u hasta uno de sus descendientes en el T_{DFS} y a continuación “subimos” con una arista de retroceso. Denominaremos $mas_alto[u]$ a dicho número.



Un vértice u es un **punto de articulación** de G si su eliminación desconecta el grafo. Si G no tiene puntos de articulación entonces G es biconexo. Para ver si un vértice es punto de articulación o no debermos considerar varios casos:

- 1 u es una hoja del T_{DFS} (no tiene descendientes): entonces no es un punto de articulación ya que nunca se desconectará el grafo
- 2 $u = r$ es la raíz del T_{DFS} : es punto de articulación si y sólo si tiene más de un descendiente en el T_{DFS} , su eliminación desconectaría los subárboles, pero si sólo hay uno su eliminación no desconecta a los demás vértices.

- 3 u no es hoja y no es la raíz: entonces es punto de articulación si y sólo si alguno de los vértices adyacentes descendientes w de u cumple $mas_alto[w] \geq ndfs[u]$, es decir, si algún descendiente no puede “escapar” del subárbol enraizado en u sin pasar por u .

Por otro lado para cada vértice u , $mas_alto[u]$ es el mínimo entre: 1) $ndfs[u]$, 2) $mas_alto[v]$ para todo v descendiente directo de u en el T_{DFS} y 3) $ndfs[v]$ para todo v adyacente a u mediante una arista de retroceso.

procedure BICONEXO(G)

Require: G es conexo

for $v \in V(G)$ **do**

$visitado[v] := \text{false}$

$ndfs[v] := 0$

$punto_art[v] := \text{false}$

$num_desc[v] := 0$

end for

$num_dfs := 0$

$es_biconexo := \text{true}$

▷ Basta lanzar un DFS porque G es conexo

$v :=$ un vértice cualquiera de G

 BICONEXO-REC($G, v, v, es_biconexo, ndfs, \dots$)

return $es_biconexo$

end procedure

```

procedure BICONEXO-REC( $G, v, padre, \dots$ )
     $num\_dfs := num\_dfs + 1$ ;  $ndfs[v] := num\_dfs$ 
     $visitado[v] := \mathbf{true}$ 
     $mas\_alto[v] := ndfs[v]$ 
    for  $w \in G.ADJACENT(v)$  do
        if  $\neg visitado[w]$  then
             $num\_desc[v] := num\_desc[v] + 1$ 
            BICONEXO-REC( $G, w, v, \dots$ )
             $mas\_alto[v] := \min(mas\_alto[v], mas\_alto[w])$ 
             $punto\_art[v] := punto\_art[v] \vee$ 
                ( $mas\_alto[w] \geq ndfs[v]$ )
        else if  $padre \neq w$  then
             $mas\_alto[v] := \min(mas\_alto[v], ndfs[w])$ 
        end if
    end for
    if  $v = padre$  then  $\triangleright v$  es la raíz del  $T_{DFS}$ 
         $punto\_art[v] := num\_desc[v] > 1$ 
    end if
     $es\_biconexo := es\_biconexo \wedge \neg punto\_art[v]$ 
end procedure

```

Análisis del DFS

Supongamos que el trabajo que se realiza en cada visita de un vértice (PRE- y POST-VISIT) y cada visita de una arista, sea del árbol o de retroceso, es $\Theta(1)$. Entonces el coste del DFS es

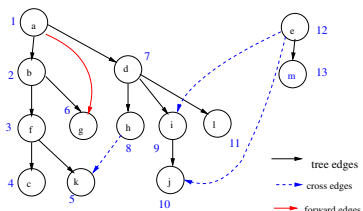
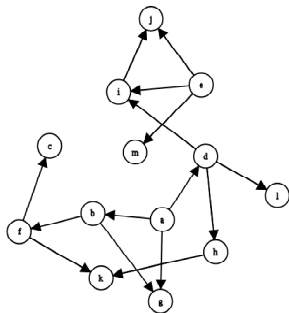
$$\begin{aligned}\sum_{v \in V} \left(\Theta(1) + \Theta(\text{grado}(v)) \right) &= \Theta \left(\sum_{v \in V} \left(1 + \text{grado}(v) \right) \right) = \\ &\Theta \left(\sum_{v \in V} 1 + \sum_{v \in V} \text{grado}(v) \right) = \Theta(n + m)\end{aligned}$$

DFS en Digrafos

El recorrido en profundidad de un digrafo tiene propiedades algo distintas de las del DFS de un grafo no dirigido. Al lanzar un DFS desde un vértice v de un digrafo G se visitarán los vértices (no visitados) de la componente fuertemente conexa de v pero además todos los otros vértices **accesibles** desde v . El DFS de un digrafo induce sucesivos árboles de recorrido, enraizados en los vértices desde los cuales se lanzan los recorridos recursivos. Los conceptos de numeración DFS y de numeración inversa son igual que en el caso de grafos no dirigidos.

El DFS de un digrafo también induce una clasificación de los arcos, pero es algo diferente:

- 1 Arcos de árbol: van del vértice v en curso a un vértice no visitado w
- 2 Arcos de retroceso: van del vértice v en curso a un vértice antecesor w en el árbol T_{DFS} ; se cumple que $ndfs[w] < ndfs[v]$ y que $ndfs[w]$ está abierto
- 3 Arcos de avance (*forward edges*): van del vértice v en curso a un vértice descendiente pero previamente visitado w en el T_{DFS} ; se cumple que $ndfs[v] < ndfs[w]$
- 4 Arcos de cruce (*cross edges*): van del vértice v en curso a un vértice previamente visitado w en el mismo T_{DFS} o en un árbol diferente; se cumple que $ndfs[w] < ndfs[v]$, pero el vértice w ya está cerrado ($ninv[w] \neq 0$)

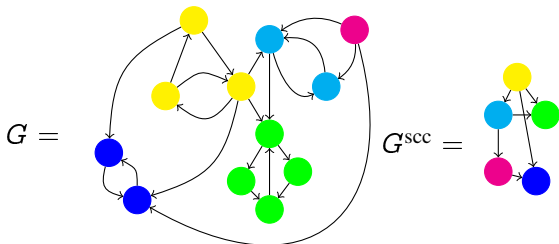


Aplicaciones del DFS: Componentes Fuertemente Conexas

Consideraremos aquí el algoritmo de Kosaraju-Sharir para calcular las componentes fuertemente conexas (SCC, *strongly connected components*) de un digrafo G . Para ello se emplean dos DFS sobre el digrafo G .

El algoritmo se fundamenta en la observación de que G y G^T (el digrafo que G transpuesto, es decir, el que se obtiene invirtiendo la orientación de todos los arcos de G) tienen idénticas SCCs.

Dado un digrafo $G = \langle V, E \rangle$ se denomina grafo de **condensación** o **grafo de SCCs** al digrafo acíclico dirigido $G^{\text{scc}} = \langle V^{\text{scc}}, E^{\text{scc}} \rangle$ en el que hay un vértice $[v]$ por cada SCC de G y existe un arco $([u], [v])$ si existen vértices $x, y \in G$ tales que $(x, y) \in E$ y x pertenece a la SCC asociada a $[u]$ e y pertenece a la SCC asociada a $[v]$.



Por otro lado, es obvio que $(G^{\text{scc}})^T = (G^T)^{\text{scc}}$.

Supongamos que iniciamos un DFS en un vértice u que pertenece a un vértice hoja en $(G^{\text{scc}})^T$. Entonces visitaremos exactamente los vértices de la SCC de u (ej: un vértice amarillo en el ejemplo). Si visitamos los vértices de G^{scc} en orden topológico inverso iremos recobrando todas las SCCs. Por ejemplo, si ya se ha visitado la SCC amarilla y lanzamos un DFS desde un vértice verde visitaremos todos los vértices de la SCC verde (y nada más). Pero en el algoritmo queremos evitar la construcción explícita de $(G^{\text{scc}})^T$.

Lanzamos sucesivos DFS sobre G y vamos guardando los vértices de G en una lista L en orden inverso de cierre (esto es, el primer vértice cerrado en el primer DFS es el último de L y así sucesivamente). Si un vértice x aparece en L antes que un vértice y entonces ambos pertenecen a una misma SCC o bien la SCC de x debe visitarse antes que la SCC de y en un orden topológico inverso de $(G^{\text{SCC}})^T$ porque en G es seguro que desde y no puede alcanzarse a x , salvo que estén en la misma SCC. Por lo tanto en G^T o bien x e y son de la misma SCC o desde x no puede alcanzarse a y .

El algoritmo resuelve el problema entonces mediante dos recorridos en profundidad (tres, si hay que calcular G^T o las listas de predecesores) y como todas las operaciones por vértice y por arco aplicadas son de coste constante, el coste del algoritmo es lineal respecto al tamaño del digrafo $\Theta(|V| + |E|)$.

Primera fase del algoritmo de Kosaraju-Sharir

- ▷ **Post:** Genera una lista L de los vértices de G
- ▷ **en orden descendiente de número de cierre**

procedure DFS(G, L)

for $v \in V(G)$ **do**

$visitado[v] := \text{false}$

end for

$L := \emptyset$

for $v \in V(G)$ **do**

if $\neg visitado[v]$ **then**

 DFS-REC($G, v, L, visitado$)

end if

end for

end procedure

```
procedure DFS-REC( $G, v, L, \textit{visitado}$ )  
   $\textit{visitado}[v] := \textbf{true}$   
  for  $w \in G.\text{SUCCESSOR}(v)$  do  
    if  $\neg \textit{visitado}[w]$  then  
      DFS-REC( $G, w, L, \textit{visitado}$ )  
    end if  
  end for  
   $L := \{v\} + L$   
end procedure
```

Segunda fase del algoritmo de Kosaraju-Sharir

procedure EXTRACTSCC(G)

 EXTRACT-SCCG, L

for $v \in V(G)$ **do**

$SCC[v] := -1$

end for

$n_{scc} := 0$

for $v \in L$ **do**

if $SCC[v] = -1$ **then**

$n_{scc} := n_{scc} + 1$

 ▷ Necesitaremos los predecesores

 ▷ de cada vértice de G

 EXTRACTSCC-REC(G, v, SCC, n_{scc})

end if

end for

end procedure

```
procedure EXTRACTSCC-REC( $G, v, SCC, n_{scc}$ )  
   $SCC[v] := n_{scc}$   
  for  $w \in G.PREDECESSOR(v)$  do  
    if  $SCC[w] = -1$  then  
      EXTRACTSCC-REC( $G, v, SCC, n_{scc}$ )  
    end if  
  end for  
end procedure
```