

Resumen-ADA.pdf



tumorenito69



Análisis y diseño de algoritmos



2º Grado en Ingeniería Informática



Escuela Politécnica Superior
Universidad de Alicante



**Que no te escriban poemas de amor
cuando terminen la carrera** ►►►►►►►

☺
*(a nosotros por
suerte nos pasa)*

WUOLAH

Que no te escriban poemas de amor
cuando terminen la carrera ➡➡➡➡➡



WUOLAH

(a nosotros por suerte nos pasa)

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirme
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah!
Tu que eres tan bonita

T2: EFICIENCIA

Noción de complejidad:

Complejidad algorítmica -> medida de recursos que necesita un algoritmo para resolver un problema.

-**Complejidad temporal:** curva que expresa el tiempo...

La complejidad temporal en el mejor de los casos es una función de la talla que tiene que estar definida para todos los posibles valores de esta.

-**Complejidad espacial:** cantidad de tamaño que necesita un algoritmo. Variables que se definen en el algoritmo.

-**Tamaño del problema:** cantidad de espacio en memoria que se necesita para codificar una instancia de ese problema.

Cálculo de complejidades (fórmulas):

$$\sum_{i=m}^n C \approx C \cdot (n - m + 1)$$

$$S = \sum_{i=0}^n a_i = a_0 \frac{1 - r^{n+1}}{1 - r}$$

$$\sum_{i=1}^n i = \frac{(n+1) \cdot (n-1+l)}{2}$$

$$\sum_{i=0}^n 2^i = \frac{1 - 2^{n+1}}{1 - 2} = 2^{n+1} - 1 \quad 2^0 \cdot \frac{1 - 2^{n+1}}{1 - 2} = 2^{n+1}$$

$$r < 1 \rightarrow \text{cte}$$

Cotas de 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** $\rightarrow C_m(n)$

Análisis asintótico -> estudio de la complejidad para tamaños grandes del problema.

COTA SUPERIOR. NOTACIÓN O

$$f \in O(f)$$

$$f \in O(g) \Rightarrow O(f) \subseteq O(g)$$

$$O(f) = O(g) \Leftrightarrow f \in O(g) \wedge g \in O(f)$$

$$f \in O(g) \wedge g \in O(h) \Rightarrow f \in O(h)$$

$$f \in O(g) \wedge f \in O(h) \Rightarrow f \in O(\min\{g, h\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(\max\{g_1, g_2\})$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(g_1 + g_2)$$

$$f_1 \in O(g_1) \wedge f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f \in O(g)$$

$\rightarrow = 0 \rightarrow g(n) \in O(f(n))$
 $\rightarrow = \infty \rightarrow g \in O(f)$
 $\downarrow = \infty \left\{ \begin{array}{l} g \in O(f) \\ f \in O(g) \end{array} \right.$

$$f(n) = a_m n^m + \dots + a_1 n + a_0 \text{ con } a_m > 0 \Rightarrow f(n) \in O(n^m)$$

$$O(f) \subset O(g) \Rightarrow f \in O(g) \wedge g \notin O(f)$$

- \Rightarrow
 $= 1 \rightarrow \text{mismo orden}$
 $= \infty \rightarrow \text{arriba mayor}$
 $= 0 \rightarrow \text{abajo mayor}$

COTA SUPERIOR. NOTACIÓN Ω

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

COSTE EXACTO. NOTACIÓN Θ

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

Ejemplo:

De las siguientes expresiones, o bien dos son verdaderas y una es falsa o bien al contrario: dos son falsas y una es verdadera. Marca la que en este sentido es distinta a las otras dos.

Seleccione una:

- a. $2n^3 - 10n^2 + 1 \in O(n^3)$ ✓ O $\text{coef } \leq O(-)$
- b. $n + n\sqrt{n} \in \Omega(n)$ ✓ Ω $\text{coef } \geq \Omega(-)$ ✗
- c. $n + n\sqrt{n} \in \Theta(n)$ F Θ mayor coef.

Que no te escriban poemas de amor cuando terminen la carrera

(a nosotros por suerte nos pasa)



Ayer a las 20:20

Oh Wuolah wuolitah
Tu que eres tan bonita

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

No si antes decirte
Lo mucho que te voy a recordar



Envía un mensaje...



WUOLAH

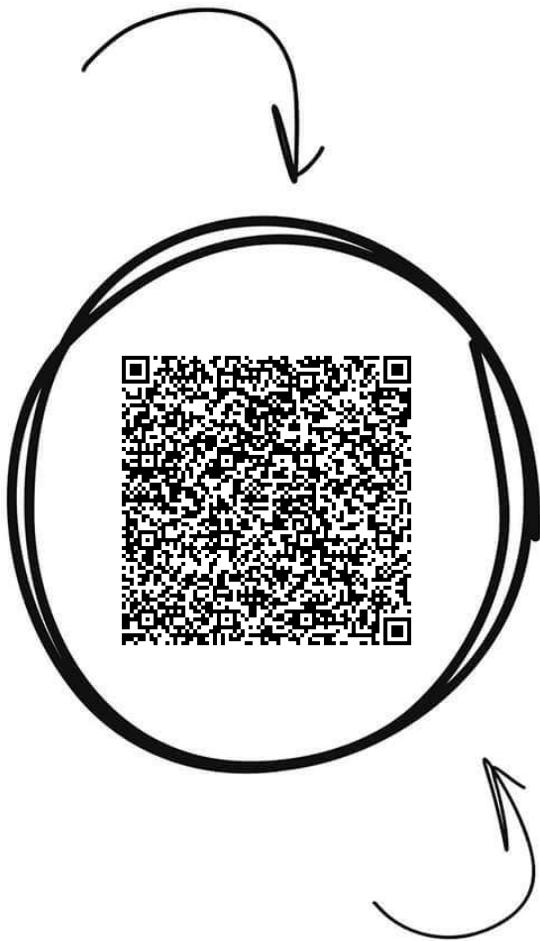


Análisis y diseño de algoritmos



Comparte estos flyers en tu clase y consigue más dinero y recompensas

- 1 Imprime esta hoja
- 2 Recorta por la mitad
- 3 Coloca en un lugar visible para que tus compis puedan escanear y acceder a apuntes
- 4 Llévate dinero por cada descarga de los documentos descargados a través de tu QR



Banco de apuntes de la UA



Jerarquía de funciones:

$$\begin{aligned}
 & O(1) \subset O(\log \log n) \subset O(\log n) \subset O(\log^{a(>1)} n) \\
 & \text{constantes} \quad \text{sublogarítmicas} \quad \text{logarítmicas} \\
 & \subset O(\sqrt{n}) \subset O(n) \subset O(n \log n) \\
 & \text{sublineales} \quad \text{lineales} \quad \text{lineal-logarítmicas} \\
 & \subset O(n^2) \subset O(n^{a(>2)}) \subset O(2^n) \subset O(n!) \subset O(n^n) \\
 & \text{polinómicas} \quad \text{exponenciales} \quad \text{superexponenciales}
 \end{aligned}$$

Cálculo de complejidades:

$$\sum_{i=1}^{n-1} \log n = n \log n$$

Pasos a seguir:

1- Determinar la talla o tamaño del problema.

2- Determinar el caso mejor y peor.

Fijar tamaño n y ver si tarda más o menos para diferentes instancias de ese tamaño.

Si para un tamaño solo hay una instancia -> no tiene caso mejor ni peor.

3- Obtención de las cotas para cada caso

-Algoritmos iterativos

-Algoritmos recursivos

Relación de recurrencia -> 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 mejores que n. Si dispone de mejor y peor caso, puede haber una relación de recurrencia para cada caso.

1- Determinar la talla o tamaño del problema.

2- Obtención de relaciones de recurrencia del algoritmo.

3- Resolución de las relaciones.

QUICKSORT

→ opera directamente sobre los elementos del vector

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

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) & n > 1 \end{cases} \quad \mathcal{O}(n \log_2 n)$$

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

$$T(n) \in \begin{cases} \Theta(1) & n \leq 1 \\ \Theta(n) + T(0) + T(n-1) & n > 1 \end{cases} \quad \mathcal{O}(n^2)$$

Ordenación por selección $\rightarrow \Theta(n^2)$

\nwarrow actúan de la misma forma \rightarrow

para ordenar un vector de n elementos, buscan el máximo de esos n elementos, lo intercambian con el n -ésimo elemento para ponerlo al final, y luego ordenan, usando el mismo algoritmo, el vector de las primeras $n - 1$ componentes.

HEAPSORT \rightarrow opera directamente sobre los elementos del vector

$\Omega(n)$

$O(n \log n)$

- inserción en un montículo $\{O(\log n)\}$
- borrar en un montículo $\{\omega(1)\}$
- construir \rightarrow con montículo auxiliar $\{O(n \log n)\}$
 \nwarrow sin espacio adicional $\{\Theta(n)\}$

Algunas relaciones de recurrencia típicas:

relación	complejidad	ejemplos
$T(n) = T(n/2) + O(1)$	$O(\log n)$	búsqueda binaria
$T(n) = T(n-1) + O(1)$	$O(n)$	búsqueda lineal factorial bucles for, while
$T(n) = 2 T(n/2) + O(1)$	$O(n)$	recorrido de árboles binarios: preorden, en orden, postorden
$T(n) = 2 T(n/2) + O(n)$	$O(n \log n)$	ordenación rápida (quick sort)
$T(n) = T(n-1) + O(n)$	$O(n^2)$	ordenación por selección ordenación por burbuja
$T(n) = 2 T(n-1) + O(1)$	$O(2^n)$	torres de hanoi

$$T(n) = 2T(n/2) + O(1) \quad n^2$$

EXTRAS/TESTS:

- QuickSort pivote mediana = no influye que esté ordenado en la complejidad temporal
No presenta caso mejor ni peor distintos para instancias del mismo tamaño.
Fuerza el mejor caso.
- QuickSort pivote central = mejor cuando está ordenado
Proporciona el mejor resultado.
- QuickSort pivote primero = peor cuando está ordenado
- La complejidad temporal asintótica del QuickSort de dividir el problema en dos subproblemas es $O(n)$, es decir, al dividir el vector, recorremos el nuevo vector para elegir el pivote.

$$\hookrightarrow T(n) = n + 2T(n/2)$$

$$\overbrace{f}^{g=f} = g$$

$$f \in \Theta(g) \rightarrow O(f) = O(g) \vee$$

$g \in O(f) \rightarrow f \not\in O(g) \rightarrow f$ no puede ser que g y f menor que f al mismo tiempo

$$g \leq f$$

$$f \not\in \omega(g) \rightarrow O(f) = \omega(g) \wedge$$

$$f < g$$

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirme
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

T3: DIVIDE Y VENCERÁS → descomponer, resolver y combinar

Consiste en:

- Descomponer el problema en subproblemas de menor tamaño que el original.
- Resolver cada problema de forma individual e independiente.
- Combinar las soluciones.

Aplicable si encontramos:

- Forma de descomponer un problema en subproblemas de talla menor.
- Forma directa de resolver problemas menores de un tamaño determinado.
- Forma de combinar soluciones de los subproblemas que permita obtener la solución del problema original.

La **complejidad temporal resultante** de un problema al que se le aplica Divide y Vencerás no depende únicamente del tamaño resultante de los **subproblemas**, hay que considerar el **coste** de cada recursión, es decir, si nuestro algoritmo recorre un bucle antes de dividirse, ese coste también se tendrá en cuenta.

Esquema Divide y Vencerás (DC)

```
1 Solution DC( Problem p ) {  
2     if( is_simple(p) ) return trivial(p);  
3     ...  
4     list<Solution> s;  
5     for( Problem q : divide(p) ) s.push_back( DC(q) );  
6     return combine(s);  
7 }
```

Eficiencia: costes logarítmicos a exponentiales.

Depende de:

- No de subproblemas (h)
- Tamaño de los subproblemas -> cuanto más parecidos mejor.
- Grado de intersección entre los subproblemas -> si los elementos de una rama lo están también en la otra.

• Ecuación de recurrencia:

- $g(n)$ = tiempo del esquema para tamaño n . (sin llamadas recursivas)
- b = Cte. de división de tamaño de problema

$$T(n) = hT\left(\frac{n}{b}\right) + g(n)$$

- Solución general: suponiendo la existencia de un entero k tal que:
 $g(n) \in \Theta(n^k)$

$$2^{\log_2 n} = n$$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } h < b^k \\ \Theta(n^k \log_b n) & \text{si } h = b^k \\ \Theta(n^{\log_b h}) & \text{si } h > b^k \end{cases}$$

WUOLAH

Teorema de reducción -> los mejores resultados en cuanto a coste se obtienen cuando los subproblemas son aproximadamente del mismo tamaño.

Si se cumple la condición del teorema de reducción ($b = h = a$)

$$T(n) = aT\left(\frac{n}{a}\right) + g(n) \quad g(n) \in \Theta(n^k)$$

$$T(n) = \begin{cases} \Theta(n^k) & k > 1 \\ \Theta(n \log n) & k = 1 \\ \Theta(n) & k < 1 \end{cases}$$

MERGESORT

→ no opera directamente sobre los elementos del vector y necesita almacenamiento adicional

$$T(n) = \begin{cases} 1 & n \leq 1 \\ n + 2T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

Complejidad temporal: $f(n) \in \Theta(n \log n)$

Complejidad espacial -> misma relación de recurrencia

Número de llamadas recursivas que hace -> cambiar n por 1 y tiene coste n

Merge → $\Theta(n)$ → tanto temp como espacial

TORRES DE HANOI

$$T(n) = \begin{cases} 1 & n = 1 \\ 1 + 2T(n-1) & n > 1 \end{cases} \quad T(n) = 2^n - 1 \in \Theta(2^n)$$

-**Selección del k-ésimo mínimo** -> Dado un vector A de n números enteros diferentes,

diseñar un algoritmo que encuentre el k-ésimo valor mínimo. $T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ T(n-1) + n & \text{en otro caso} \end{cases}$

-**Búsqueda binaria o dicotómica** -> Dado un vector X de n elementos ordenado de forma ascendente y un elemento e, diseñar un algoritmo que devuelva la posición del elemento e en el vector X.

Algoritmo	Complejidad temporal		Complejidad espacial
	Mejor	Peor	Peor
Quicksort	$n \log n$	n^2	1
Mergesort	$n \log n$	$n \log n$	n
Hanoi	2^n	2^n	
BubbleSort	n	n^2	1
Ordenación directa	n	n^2	
Inserción binaria	$n \log n$	n^2	
K-ésimo	n	n^2	
Insertion directa	n	n^2	1
Selection directa	n^2	n^2	1
Búsqueda binaria	1	$\log n$	
Heapsort	n	$n \log n$	

EXTRAS/TESTS:

mejor complej para ordenar un vector es $n \log n$

MergeSort y QuickSort tienen en común:

- aplican Divide y Vencerás.
- el número de llamadas recursivas que hacen en el mejor de los casos.
- misma complejidad temporal en el mejor caso.

Si se modifica MergeSort para dividir en tres partes, posteriormente se combinan las soluciones parciales la nueva complejidad temporal es: $n \log(n)$. Y además, la complejidad de la combinación de las soluciones parciales sería Theta(n).

La complejidad temporal asintótica del MergeSort de dividir el problema en dos subproblemas es $O(1)$.

MergeSort tiene menor complejidad que Inserción Directa y que Burbuja.

Los algoritmos de ordenación directos tienen mayor complejidad temporal y ejecuciones mayores que los indirectos.

El mergeSort es asintóticamente más rápido o igual que el QuickSort.

— tiene que cumplir el principio de optimidad

¿Cuál es la complejidad temporal en el caso mejor del algoritmo que, dado un vector, nos dice cuál de sus elementos quedaría en la posición k si lo ordenáramos por orden descendente de valor? $\rightarrow O(n)$

T4: PROGRAMACIÓN DINÁMICA

→ mejora la ejecución, guardando los resultados

Programación dinámica

-Problemas con soluciones lentas debido a que resuelven repetidamente los mismos problemas. La programación dinámica evita esas repeticiones **guardando resultados de subproblemas**, a expensas de aumentar la complejidad espacial.

-Solución alternativa a una solución recursiva ingenua que por un lado se basa en obtener soluciones óptimas a problemas parciales más pequeños y por otro, estos subproblemas se resuelven más de una vez durante el proceso recursivo.

-Esquema más apropiado cuando la descomposición de un problema da lugar a subproblemas de tamaño similar al original, muchos de los cuales se repiten.

-Para valores continuos la programación dinámica recursiva puede resultar mucho más eficiente que la programación dinámica iterativa en cuanto al uso de memoria. Iterativo rellena toda la tabla mientras que en el recursivo solo algunas.

-La mejora de la programación dinámica frente a la solución ingenua es que en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.

Subestructura óptima: si una solución óptima puede construirse eficientemente a partir de las soluciones óptimas de sus subproblemas. Es **una condición necesaria para que se pueda aplicar programación dinámica**. También conocido como **principio de optimalidad**.

Esquema Programación dinámica (DP, recursiva)

```
Solution DP( Problem p ) {
    if( is_solved(p) ) return M[p];
    if( is_simple(p) ) return M[p] = trivial(p); // or simply: return trivial(p)

    list<Solution> s;
    for( Problem q : divide(p) ) s.push_back( DP(q) );
    M[p] = combine(s);
    return M[p];
}
```

- ¿Se puede evitar la recursividad? En este caso sí
 - Resolver los subproblemas de menor a mayor
 - **Almacenar** sus soluciones en una tabla $M[n][r]$ donde

$$M[i][j] = \binom{i}{j}$$

- El almacén de resultados parciales permite evitar repeticiones.
- La tabla se inicializa con la solución a los subproblemas triviales:

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

Esquema Programación dinámica (iterativa)

```
Solution DP( Problem P ) {  
    vector<Solution> M;  
    list<Problem> e = enumeration(P);  
  
    while( !e.empty() ) {  
        Problem p = e.pop_front();  
        if( is_simple(p) )  
            M[p] = trivial(p);  
        else {  
            list<Solution> s;  
            for( Problem q : divide(p) ) s.push_back( M[q] );  
            M[p] = combine(s);  
        }  
    }  
    return M[P];  
}
```

Problema de la mochila

Los pesos son discretos y hay que cogerlos o no.

$\Omega(n)$ -> recorrer todo, ver que no cabe ninguno y acabar.

Solución recursiva:

- Solución recursiva ingenua -> $O(2^n)$
- Solución recursiva con almacén (Memoización) -> Temporal y espacial $O(nW)$

Solución iterativa:

- Solución iterativa con complejidad espacial mejorable(matriz)
-> Temporal y espacial $O(nW)$
- Solución iterativa con complejidad espacial mejorada(dos vectores)
-> Temporal $O(nW)$ y Espacial $O(W)$

Para la mochila discreta tiene la restricción de que los valores tienen que ser enteros positivos.

La secuencia óptima de decisiones tomadas se puede obtener en $\Theta(n)$.

Si W es muy grande entonces la solución obtenida mediante programación dinámica no es buena.

La complejidad espacial de la solución obtenida se puede reducir hasta $\Theta(W)$.

Corte de tubos

Hacer una evaluación exhaustiva “de fuerza bruta” de todas las posibles maneras de cortar el tubo consume un tiempo $\Theta(2^n)$.

Es posible evitar hacer la evaluación exhaustiva “de fuerza bruta” guardando, para cada longitud $j < n$ el precio más elevado posible que se puede obtener dividiendo el tubo correspondiente.

WUOLAH

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ n + \sum_{j=0}^{n-1} T(j) & \text{en otro caso} \end{cases}$$

$$T(n) = 2^n - 1 + 2^n \in O(2^n)$$

-Solución recursiva con almacén (Memoización)

-Temporal: $O(n^2)$

-Espacial: $O(n)$

-Solución iterativa

-Temporal: $O(n^2)$

-Espacial: $O(n)$

La complejidad espacial no se puede reducir

Coefficiente binomial

Se repiten muchos cálculos que se pueden evitar usando programación dinámica.

Fibonacci

EXTRAS/TESTS:

- La mejora que en general aporta la programación dinámica frente a la solución ingenua se consigue gracias al hecho de que en la solución ingenua se resuelve muchas veces un número relativamente pequeño de subproblemas distintos.
- Normalmente se usa programación dinámica para resolver problemas de optimización con dominios discretizados puesto que las tablas se han de indexar con este tipo de valores.
- En algunos casos se puede utilizar para resolver problemas de optimización con dominios continuos pero probablemente pierda su eficacia ya que se puede disminuir drásticamente el número de subproblemas repetidos.
- Por eliminación → el problema del cambio tiene solución eficiente con P.D.
- Complejidad espacial $O(n) \rightarrow 2$ vectores → el almacén no se ha guardado → no se puede hacer paralelo
No podemos determinar donde colocar las puertas de enlace.
- El elemento n -ésimo de la serie tribonacci, $T(n)$, se define como sigue: $T(n) = T(n - 3) + T(n - 2) + T(n - 1)$ para $n \geq 3$; $T(0) = 0$; $T(1) = 1$ y $T(2) = 1$.
 - (a) Una implementación ingenua de la función $T(n)$, la cual llamaría a $T(n-1)$, $T(n-2)$ y $T(n-3)$ tendría una complejidad prohibitiva por la repetición de cálculos que se produciría.
 - (b) Es posible una implementación de programación dinámica iterativa con complejidad $\Theta(n)$.

- Problemas clásicos para los que resulta eficaz la programación dinámica

- El problema de la mochilla 0-1
- Cálculo de los números de Fibonacci
- Problemas con cadenas:
 - La subsecuencia común máxima (*longest common subsequence*) de dos cadenas.
 - La distancia de edición (*edit distance*) entre dos cadenas.
- Problemas sobre grafos:
 - El viajante de comercio (*travelling salesman problem*)
 - Caminos más cortos en un grafo entre un vértice y todos los restantes (alg. de Dijkstra)
 - Existencia de camino entre cualquier par de vértices (alg. de Warshall)
 - Caminos más cortos en un grafo entre cualquier par de vértices (alg. de Floyd)

T5: ALGORITMOS VORACES

→ establecer criterio, ordenar en base a él y coger. No reconsidera.

El problema se reduce a:

- Seleccionar un subconjunto de los objetos disponibles,
- que cumpla las restricciones, y
- que maximice la función objetivo

Se necesita un criterio voraz que decida qué valor tomar en cada momento.

Es habitual preparar los datos para disminuir el coste temporal de la función que determina cuál es la siguiente decisión a tomar.

Algoritmos voraces (Greedy)

Un **algoritmo voraz** es aquel que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción local óptima en cada paso con la esperanza de llegar a una solución general óptima.

-Descomponer el problema en un conjunto de decisiones locales (nos fijamos solo donde estamos) y elegir la más prometedora, es decir, aquella que se considera mejor para optimizar la medida global.

-La eficiencia se basa en el hecho de que nunca se reconsidera las opciones ya tomadas, lo que conduce a algoritmos (muy) eficientes.

Dado un problema de optimización el método voraz garantiza la solución óptima sólo para determinados problemas.

Esquema voraz

```
t_conjuntoElementos VORAZ(t_problema dp)
{
    t_conjuntoElementos y, solucion;
    elemento decision;
    y=prepararDatos(dp);           // preparacion de datos para facilitar seleccion
    while(noVacio(y) || !esSolucion(solucion)) { // quedan datos por seleccionar
        // y aun no se ha llegado a la solucion
        decision=selecciona(y);
        if (esFactible(decision,solucion))
            solucion=anadeElemento(decision,solucion);
        y=quitaElemento(decision,y);   // descartar en cualquier caso
    }
    return solucion;
}
```

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

No si antes decíte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirme
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

Esquema voraz (recursivo)

```
Solution greedy( Problem p ){
    Problem sub_prob;
    Decision decision;
    if( is_simple(p) )
        return trivial(p);
    auto [ sub_prob, decision ] = select( divide(p) );
    return Solution( greedy(sub_prob), decision );
}
```

Esquema voraz iterativo (iterativo)

```
Solucion greedy( Problem p ){
    Problem sub_prob;
    Decision decision;
    Solution solution;
    while( ! is_simple(p) )
        solution += select( divide(p) );
        solution += trivial(p);
    return solution;
}
```

Características:

- Puede que no se encuentre la solución óptima, puede que no exista criterio para encontrarla.
- Puede que no se encuentre ninguna solución.
- Si soluciona el problema quizás sea la mejor solución que existe.

Ejemplos

→ no quiere decir que los pesos sean discretos

- Problema de la mochila discreta (sin fraccionamiento)

El método voraz no resuelve el problema.

El valor que se obtiene es una cota inferior para el valor óptimo que a veces puede ser igual a este.

Obtiene resultado menor/mayor o igual al resultado correcto pero no sabes si es el correcto.

- Problema de la mochila continua

El método voraz resuelve el problema.

Es conveniente la ordenación previa de los objetos para reducir la complejidad temporal en la toma de cada decisión: de $O(n)$ a $O(1)$, donde n es el número de objetos a considerar.

WUOLAH

- El problema del cambio

El método voraz no tiene porqué resolverlo.

- Árboles de recubrimiento de coste mínimo

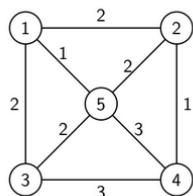
La solución óptima puede construir un único árbol que va creciendo (Prim) o bien construir un bosque de árboles que al final se injertan en un único árbol (Kruskal). Tanto vértice a vértice como arista a arista.

PRIM

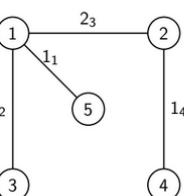
mantener para cada vértice su "padre" más cercano para acelerar

Pasos:

- 1- Se coge un vértice y se añade al conjunto de explorados.
- 2- Se busca el arco de mínimo peso que va de uno explorado a uno sin explorar.
- 3- Se añade el vértice a los explorados y el arco a la solución.



Alg. Prim



Un posible resultado partiendo del vértice 1. La arista (1,3) se ha escogido antes que la (1,2) y esto hace que la (2,4) se escoja en último lugar.

Proceso:

- 1 Parte del vértice 1. Se marca como visitado: $V' = \{1\}$
- 2 Selecciona la arista (1,5). $V' = \{1, 5\}$
- 3 Selecciona la arista (1,3) aunque también podría haber seleccionado la (1,2). $V' = \{1, 3, 5\}$. -la arista (2,4) no puede seleccionarse pues no cumple la condición-
- 4 Selecciona la arista (1,2). $V' = \{1, 2, 3, 5\}$
- 5 Selecciona la arista (2,4). Termina ($V' = V$)



- No hace falta incurrir en el coste cuadrático de recorrer todos los arcos cada vez.
- **Si cambia el mínimo a seleccionar es a causa del último vértice añadido**
 - Hay que guardarse, para cada vértice no visitado, La arista mínima que incide en él desde un vértice visitado:
 - Se hace mediante un vector w de mínimos ya calculados que se actualiza cada vez que se visita un vértice,
 - en w_i está el peso de la mejor arista que conecta un vértice visitado con el vértice i (aún sin visitar),
 - Además, hay que utilizar otro vector f :
 - El vector f es necesario para saber de qué arista se trata,
 - en f_i está el vértice origen de la arista representada por w_i .

KRUSKAL

Pasos:

- 1- Selección de aristas por orden creciente
Ordenación previa de las aristas.
- 2- Se incorpora a la solución si no forma ciclos
- 3- Termina cuando se han seleccionado $n-1$ aristas

- El resultado será una única componente conexa aunque se pueden ir formando distintos subárboles que inicialmente no están conexos, a diferencia de Prim que es una componente conexa durante todo el proceso.

Aplicándolo al algoritmo de Kruskal:

- ① Mantener una partición de los vértices.
 - Se puede hacer con un vector de etiquetas. Una por cada vértice.
 - Inicialmente las n etiquetas son distintas lo que indicaría que cada vértice está en una componente conexa independiente al resto.
- ② Mientras queda más de una componente conexa:
 - **Buscar** la arista de menor peso que une dos componentes distintas,
 - (Esto asegura que no habrá ciclos)
 - Añadir la arista a la solución
 - **Unir** las dos componentes conexas en una única componente conexa (poner a todos los vértices implicados la misma etiqueta)
- ③ El proceso termina cuando todos los vértices están en la misma componente conexa.

Emplea TAD “Union-find” para comprobar si un arco forma ciclos.

COMPARATIVA PRIM VS KRUSKAL

- El algoritmo de Prim tiene una complejidad $O(V^2)$
- El algoritmo del Kruskal tiene una complejidad $O(E \log E)$
 - En el peor caso el número de aristas de un grafo es $E \in O(V^2)$
 - Por lo tanto, el algoritmo de Kruskal es:

$$O(E \log E) = O(V^2 \log V^2) = O(V^2 \log V)$$

¿Cuándo usar cada algoritmo?

- Si el grafo tiende a estar completo el algoritmo de Prim es mejor.
- El algoritmo de Kruskal es mejor cuando el grafo es disperso (pocas aristas).

- El fontanero diligente

El método voraz **resuelve** el problema.

- Asignación de tarea

En el caso de que se incorporasen restricciones que contemplen que ciertas tareas no pueden ser adjudicadas a ciertos trabajadores, la solución factible ya no estaría garantizada, es decir, pudiera ser que el algoritmo no llegue a solución alguna.

EXTRAS/TESTS:

- Puede ser la única alternativa si el dominio de las decisiones es un conjunto infinito.
- Cuando se usa para abordar un problema de optimización por selección discreta (encontrar un subconjunto del conjunto de elementos):
 - Puede que no encuentre solución.
 - Puede que la solución no sea óptima.
 - Imposible reconsiderar la decisión tomada anteriormente.
- El valor que obtiene para el problema de la mochila discreta es una cota inferior para el valor óptimo que a veces puede ser igual a este.
- Garantiza la solución óptima SÓLO para determinados problemas.
- Hay problemas para los cuales se puede obtener siempre la solución óptima, como la mochila continua.
- Hay problemas para los cuales solo obtenía la solución óptima para algunas instancias y un subóptimo para muchas instancias. Mochila discreta, a veces resuelve y otras no.
- Se basan en la idea de que la solución óptima se puede construir añadiendo repetidamente el mejor elemento disponible. Se sigue siempre el mismo criterio de selección.
- Problemas equivalentes en cuanto al tipo de solución:
 - Fontanero diligente y mochila continua -> Obtienen siempre óptimo
 - Mochila discreta y asignación de tareas -> No tienen porque llegar al óptimo
- Voraz continua → cota optimista
- Porque dominios continuos generalmente solo los voraces pueden obtener una solución

Siempre óptimo

- Mochila continua
- Fontanero diligente
- Árboles recubrimiento

A veces óptimo

- Mochila discreta
- Problema del cambio
- Asignación de tareas

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

↳ "listar todo"

T6: BACKTRACKING

→ resolver todas las posibles soluciones teniendo en cuenta restricciones así como posibilidad de descartar

Ejemplo introductorio mochila general

Búsqueda del óptimo -> recorrer todas las soluciones factibles, calcular su valor y quedarse con el mayor de todos.

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

¿Cómo acelerar el programa?

- Evitando explorar ramas que no pueden dar soluciones factibles.
- Calculando los valores a medida que se va rellenando la solución.
- Evitando explorar ramas que no pueden dar soluciones mejores que la que ya tenemos.

Otras mejoras:

También puede ser relevante:

- El orden en el que se exploran los objetos.
- La forma en la que se "despliega el árbol".

Podas más ajustadas:

- Los mecanismos de poda tienen que actuar lo antes posible.
- Una poda más ajustada se puede obtener usando la solución voraz al problema de la mochila.
- La solución al problema de la mochila continua es siempre mayor que la solución al problema de la mochila discreta.
- La efectividad de la poda también puede aumentarse partiendo de una solución factible muy buena.

Vuelta atrás

Proporciona una forma sistemática de generar todas las posibles soluciones de un problema en los que se pretende encontrar soluciones que sean:

- Factibles: que satisfagan unas restricciones.
- Óptimas: optimicen una cierta función objetivo.

Características:

- Número posible de soluciones finito.
- Dominios decisiones tiene que ser discretos o discretizables.
- Puede proporcionar una solución factible, todas las soluciones factibles, solución óptima, las n mejores soluciones factibles al problema.
- Generalmente complejidades prohibitivas.

WUOLAH

Podas para evitar la exploración completa del espacio de soluciones:

- Podar ramas que lleven a soluciones no factibles.
- Podar soluciones que solo llevan soluciones malas usando cotas optimistas y mediante inicialización con cotas pesimistas.
- Cambios en el orden de exploración/expansión para conseguir rápidamente soluciones casi óptimas.

- Cota optimista:
 - estima, a mejor, el mejor valor que podría alcanzarse al expandir el nodo
 - puede que no haya ninguna solución factible que alcance ese valor
 - normalmente se obtienen relajando las restricciones del problema
 - si la cota optimista de un nodo es peor que la solución en curso, se puede podar el nodo
- Cota pesimista: → tiene que ser siempre factible
 - estima, a peor, el mejor valor que podría alcanzarse al expandir el nodo
 - ha de asegurar que existe una solución factible con un valor mejor que la cota
 - normalmente se obtienen mediante soluciones voraces del problema
 - se puede eliminar un nodo si su cota optimista es peor que la mejor cota pesimista
 - permite la poda aún antes de haber encontrado una solución factible
- Cuanto mas ajustadas sean las cotas, mas podas se producirán
 - para actualizar la mejor cota hacia el momento sin llegar hasta sus hojas

```
void backtracking( node n, solution& present_best ){  
  
    if ( is_leaf(n) ) {  
        visited_leaf_nodes++;  
        if( is_best( solution(n), present_best ) )  
            present_best = solution(n);  
        return;  
    }  
  
    for( node a : expand(n) )  
        visited_nodes++;  
        if( is_feasible(a) ) {  
            if( is_promising(a) ) {  
                explored_nodes++;  
                backtracking( a, present_best );  
            } else  
                no_promising_discarded_nodes++;  
        } else  
            no_feasible_discarded_nodes++;  
  
    return;  
}
```

Ejemplos

- Permutaciones → no pueden haber elem. repetidos
- El viajante del comercio

Cota optimista:

- Restricciones:

- el camino ha de pasar por todas las ciudades
- el camino ha de ser continuo

- Relajaciones:

- no pasar por todas las ciudades ⇒ saltar de la última a la primera
- ir saltando de una ciudad a otra ⇒ árbol de recubrimiento mínimo

Poda basada en la mejor solución hasta el momento (cota pesimista)

- buscar una solución subóptima
- algoritmo voraz

La poda optimista que poda mejor el árbol de búsqueda es aquella en la que se ordenan las aristas restantes de menor a mayor distancia y se calcula la suma de las k aristas más cortas, donde k es el número de saltos que nos quedan por dar.

- El problema de las n reinas
- La función compuesta mínima

Ejercicios

- Coloreado de grafos
- Recorrido del caballo de ajedrez
- El laberinto con 4 movimientos
- Asignación de tareas
- Empresa naviera
- Asignación de turnos
- Sudoku

EXTRAS/TESTS:

- Es condición necesaria (aunque no suficiente) que el dominio de las decisiones sea discreto o discretizable y que el número de decisiones a tomar esté acotado.
- Los mecanismos de poda basados en la mejor solución hasta el momento pueden eliminar vectores que representan posibles soluciones factibles.
- Los mecanismos de poda basados en la mejor solución hasta el momento pueden eliminar soluciones parciales que son factibles.
- En ausencia de cotas optimistas y pesimistas, no recorre todo el árbol si hay manera de descartar subárboles que representan conjuntos de soluciones no factibles.
- El orden en el que se van asignando los distintos valores a las componentes del vector que contendrá la solución:
 - Es irrelevante si no se utilizan mecanismos de poda basados en la mejor solución hasta el momento.
 - Puede ser relevante si se utilizan mecanismos de poda basados en estimaciones optimistas.
- Al convertir un algoritmo de vuelta atrás en ramificación y poda, podemos aprovechar mejor las cotas optimistas, si ya que se usa para acceder a los nodos en este orden.
- Puede resolver el problema de la mochila discreta más rápido si se prueba primero a meter cada objeto antes de no meterlo, pero sólo si se usan cotas optimistas para podar el árbol.

- Si hacen uso de cotas optimistas, generan las soluciones posibles al problema mediante un recorrido en profundidad del árbol que representa el espacio de soluciones.
- En el peor caso la complejidad temporal de la solución de la mochila discreta es exponencial.
- Puede ocurrir que el uso de las cotas pesimistas y optimistas sea inútil, incluso perjudicial, puesto que es posible que a pesar de utilizar dichas cotas no se descarte ningún nodo.
- *Recorre un árbol en profundidad al igual que una pila*

35. El esquema de vuelta atrás ...

- (a) Las otras dos opciones son ambas verdaderas.
- (b) Garantiza que encuentra la solución óptima a cualquier problema de selección discreta.

- **¿Cómo obtener la solución óptima?**
 - Programación dinámica: objetos no fragmentables y pesos discretos
 - Algoritmos voraces: objetos fragmentables
- **– No podemos fragmentar los objetos
y los pesos son valores reales –**

En el problema de la mochila

La complejidad espacial: En el caso más desfavorable, el coste espacial asintótico de un algoritmo de ramificación y poda es exponencial, mientras que en vuelta atrás no lo es.

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirme
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

T7: RAMIFICACIÓN Y PODA

→ como vuela atrás pero incorporamos estrategias de búsqueda

Ejemplo introductorio mochila general

Ramificación y poda incorpora estrategias de búsqueda. Que se decida en tiempo de ejecución por qué rama se va a bajar.

Podemos llegar antes a la solución adecuando el orden de exploración del árbol de soluciones según nuestros intereses. Se priorizará para su exploración aquellos nodos más prometedores. → saber si es prometedor o no con cotes optimistas

Cota optimista -> valor resultante de incluir en la solución aquellos objetos pendientes de tratar independientemente de que quepan o no.

¿Puede mejorarse? -> añadiendo cotas pesimistas al expandir el nodo

- La cota pesimista permite mejorar la solución en curso.
- El mejorar la solución en curso permite hacer más podas.
- Esto no era útil en vuelta atrás.

No hace falta llegar a las hojas ya que las cotas pesimistas de los nodos intermedios ya te dan la solución.

La cota pesimista más ajustada al valor óptimo de la mochila discreta es el valor de la mochila discreta que se obtiene usando un algoritmo voraz basado en el valor específico de los objetos.

La cota optimista más ajustada al valor óptimo de la mochila discreta es el valor de la mochila continua correspondiente.

Siempre iterativos

Ramificación y poda

-Variante del diseño de vuelta atrás
Generación de un árbol de expansión.
Uso de cotas para podar ramas que no son de interés.

-**Nodo vivo** -> aquel con posibilidad de ser ramificado.

-Se almacenan en estructuras que faciliten su recorrido y eficiencia de la búsqueda.

- Pila
- Cola
- Cola de prioridad

WUOLAH

- Etapas

- Partimos del nodo inicial del árbol
- Se asigna una **solución pesimista** (subóptima, soluciones voraces)
- Selección
 - Extracción del nodo a expandir del conjunto de nodos vivos
 - La elección depende de la estrategia empleada
 - Se actualiza la mejor solución con las nuevas soluciones encontradas
- Ramificación
 - Se expande el nodo seleccionado en la etapa anterior dando lugar al conjunto de sus nodos hijos
- Poda
 - Se eliminan (podan) nodos que no contribuyen a la solución
 - El resto de nodos se añaden al conjunto de nodos vivos
- El algoritmo finaliza cuando se agota el conjunto de nodos vivos

PROCESO DE PODA

- Cota optimista:

- estima, a mejor, el mejor valor que podría alcanzarse al expandir el nodo
- puede que no haya ninguna solución factible que alcance ese valor
- normalmente se obtienen relajando las restricciones del problema
- si la cota optimista de un nodo es peor que la solución en curso, se puede podar el nodo
- suele ser una solución **no factible**

- Cota pesimista:

- estima, a peor, el mejor valor que podría alcanzarse al expandir el nodo
- ha de asegurar que existe una solución factible con un valor mejor que la cota
- normalmente se obtienen mediante soluciones voraces del problema
- se puede eliminar un nodo si su cota optimista es peor que la mejor cota pesimista
- permite la poda aún antes de haber encontrado una solución factible
- Es una solución **factible**

- Cuanto mas ajustadas sean las cotas, mas podas se producirán

Esquema de Ramificación y Poda

```
solution branch_and_bound( problem p ) {  
  
    node initial = initial_node(p);                      // supposed feasible  
    solution current_best = pessimistic_solution(initial); // pessimistic  
    priority_queue<Node> q.push(initial);  
  
    while( ! q.empty() ) {  
        node n = q.top();  
        q.pop();  
  
        if( is_leaf(n) ) {  
            if( is_better( solution(n), current_best ) )  
                current_best = solution(n);  
            continue;  
        }  
  
        for( node a : expand(n) )  
            if( is_feasible(a) && is_promising( a, current_best ) )  
                q.push(a);  
    }  
  
    return current_best;  
}
```

Características

- La estrategia puede proporcionar:
 - Todas las soluciones factibles
 - Una solución al problema
 - La solución óptima al problema
 - Las n mejores soluciones
- Objetivo de esta técnica
 - Mejorar la eficiencia en la exploración del espacio de soluciones
- Desventajas/Necesidades
 - Encontrar una buena **cota optimista** (problema relajado)
 - Encontrar una buena solución **pesimista** (estrategias voraces)
 - Encontrar una buena estrategia de exploración (cómo ordenar)
 - Mayor requerimiento de memoria que los algoritmos de Vuelta Atrás
 - las complejidades en el peor caso suelen ser muy altas
- Ventajas
 - Suelen ser más rápidos que Vuelta Atrás

pero no tiene pq que

- Muchas veces, para ver si un nodo es prometedor, se hace comparando la mejor solución obtenida con una solución optimista de nodo.

```
1 bool is_promising( const node &n, const solution &current_best) {  
2     return is_better( optimistic_solution(n), current_best );  
3 }
```

se pueden hacer podas **agresivas** cambiándola por:

```
1 bool is_promising( const node &n, const solution &current_best) {  
2     return is_significantly_better(optimistic_solution(n), current_best);  
3 }
```

¡Cuidado! puede que se pierda la solución óptima

Ejemplos

- [Viajante de comercio](#)
- [Función compuesta mínima](#)

EXTRAS/TESTS:

- Cota optimista es necesariamente un valor insuperable, de no ser así se podría podar el nodo que conduce a la solución óptima.
- El orden escogido para priorizar los nodos en la lista de nodos vivos puede influir en el número de nodos que se descartan sin llegar a expandirlos.
- La cota pesimista puede servir para actualizar el valor de la mejor solución hasta el momento.
- El uso de funciones cota puede reducir el número de instancias del problema que pertenecen al caso peor.
- La complejidad en el mejor de los casos puede ser polinómica con el número de decisiones a tomar.
- En general el valor de una cota pesimista es mayor que el valor de una cota optimista si se trata de un problema de minimización, aunque en ocasiones ambos valores pueden coincidir.

Que no te escriban poemas de amor
cuando terminen la carrera ➤➤➤➤➤



WUOLAH

(a nosotros por suerte nos pasa)

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirte
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

- En general el valor de una cota pesimista es menor que el valor de una cota optimista si se trata de un problema de maximización, aunque en ocasiones ambos valores pueden coincidir.
- La diferencia principal entre vuelta atrás y ramificación y poda en el problema de la mochila es el orden de exploración de las soluciones.
- Genera las soluciones posibles al problema mediante un recorrido guiado por estimaciones de las mejores ramas del árbol que representa el espacio de soluciones.
- *Puede que pade más que vuelta atrás pero al final sea más lenta, incluso con implementaciones muy eficientes.*
- Puede haber problemas para los que la complejidad será exponencial o peor; ninguna estrategia de poda puede garantizar que esto no va a ocurrir.

Cola de prioridad -> heap máximo

EFICIENCIA DE PROBLEMAS

Verde = Sí - Rojo = No

Solución voraz trivial y eficiente

- Mochila continua (Pero es mejor implementarla en divide y vencerás)
- Mochila discreta sin limitación en la carga máxima
- Árbol de recubrimiento mínimo para un grado no dirigido con pesos
- Mochila continua o con fraccionamiento
- Árbol de cobertura de coste mínimo de un grado conexo
- El problema del cambio
- Mochila discreta (a secas)
- El problema de la asignación de tareas de coste mínimo en una matriz

Solución dinámica eficiente

- El problema del cambio (eficiente)
- n-ésimo elemento de la serie de Fibonacci (eficiente)
- Mochila discreta (a secas)
- Cortar un tubo de longitud n en segmentos de longitud entera entre 1 y n
- La mochila discreta sin restricciones adicionales
- El problema de la asignación de tareas
- Torres de Hanoi

Ramificación y poda eficiente

- Encontrar subconjunto de tamaño m de suma mínima en un conjunto de n enteros positivos
- Problema del viajante comercio: listar todas las soluciones factibles

Vuelta atrás eficiente

- Obtener todas las permutaciones de una lista compuesta por n elementos

WUOLAH

PROBLEMA ENCAMINAMIENTO ÓPTIMO

- **versión general:** Los nodos y sus interconexiones constituyen un grafo. → no se puede aplicar P.D
- **versión simplificada:** Los nodos se conectan entre sí mediante un único *bus* bidireccional.

● A partir del problema del encaminamiento óptimo (versión simplificada), modificamos la expresión que obtiene el tráfico estimado de la red; ahora es $\sum_{i=0}^{n-1} c_i(d_i - d_{p_i})^2$ (d_i es la distancia de i al primer servidor del bus; p_i es la puerta de acceso más cercana a i). Disponemos de la nueva función $ogw(k, n)$ que obtiene el mínimo tráfico de la subred colocando solo una puerta de enlace en el lugar apropiado entre el servidor k y el $n - 1$ (ambos incluidos). Estamos interesados en una función $met(m, n)$ que calcule el mínimo tráfico de la red colocando, en las mejores ubicaciones, m puertas de enlace. Sabiendo que $met(1, n) = ogw(0, n)$, ¿cómo se podría calcular ahora, de forma recursiva, $met(m, n)$, con $1 \leq m \leq n$?

- se cumple la propiedad de subestructura óptima
- se puede calcular recursivamente

- (a) De la misma manera que en el problema original que se ha trabajado en las sesiones de prácticas.
- Se pretende resolver la versión general del problema del encaminamiento óptimo mediante la técnica "divide y vencerás", ¿se podría obtener la mejor disposición de las puertas?

- (a) No, ya que no se cumple la propiedad "subestructura óptima".

Un subcamino no tiene porque ser solución, entonces los subproblemas no resuelven el problema.

- Se pretende resolver el problema del encaminamiento óptimo (versión general) mediante ramificación y poda. Si un nodo interno del árbol de búsqueda se completa colocando las restantes puertas de enlace en los servidores con más carga de entre los que quedan por analizar. ¿Qué obtenemos?

Una cota pesimista, es decir, una hoja del árbol de búsqueda situada más abajo que ese nodo interno, en la misma rama.

- Teniendo en cuenta que $ogw(k, n)$ obtiene el tráfico estimado si se coloca una sola puerta de enlace entre los nodos k y $n - 1$, ¿cuál de las siguientes recurrencias es la más apropiada para resolver la versión simplificada del problema del encaminamiento óptimo mediante divide y vencerás?

(a)
$$met(m, n) = \begin{cases} ogw(0, n) & \text{si } m = 1 \\ \min_{k \in [m-1, n-1]} (met(m-1, k) + ogw(k, n)) & \text{si } m > 1 \end{cases}$$

COTAS

- la primera es la obvia, la trivial (asumir que los nodos no visitados no aportarán al tráfico) y de hecho, por lo rápido que se calcula, es la que entregué. Las demás eran tan lentas de calcular que no valían mucho la pena y además en algunos casos podaban incluso menos jajajaja
- la siguiente es calcular la distancia mínima en toda la red (se calcula 1 vez al principio). Para cada nodo, su cota optimista sería el tráfico que aportan los nodos visitados que no son puerta de enlace y los demás, aportan su capacidad*distancia mínima. Para nodos no hoja, se ponen las puertas de enlace en los nodos no visitados de mayor capacidad
 - planteadas de esta forma, las dos que he comentado pueden estar mal ya que quizás las puertas de enlace que queden por colocar permitan menores distancias para los nodos que ya se han visitado pero que no son puertas de enlace
 - para evitar eso, puedes usar la distancia mínima en todos los nodos ya visitados que no sean puerta de enlace también en ambas.
- a continuación, igual que la segunda que te acabo de comentar, pero en vez de usar la menor distancia, usas las m menores distancias (calculadas una vez al principio)
 - y las aplicas como prefieras siempre que cumpla que sea cota optimista. Para asegurar que sea cota optimista, puedes aplicar las distancias en orden ascendente (de distancia) sobre los nodos que no sean puertas de enlace en orden descendente (de capacidad) de forma que se puede ordenar al principio los nodos por orden descendente de capacidad e ir aplicando las distancias menores a los nodos que no sean puertas de enlace
- La siguiente cota es como la que acabo de explicar pero con una excepción, en vez de ser las m menores distancias y aplicarlas libremente, te guardas la menor distancia de cada nodo particular y esa es la que usarás para calcular la cota (aquí ya no hay que ordenar tanto)
 - en todos los casos, anteriores si quedan por poner puertas de enlace, puedes ponerlas en los que aporten más al tráfico (de entre los no visitados) según el criterio de la cota que estés usando ya que esta aportación será independiente de dónde te encuentres de la búsqueda.
- La siguiente cota es la más exigente y puede que esté un poco mal, debería revisarla y además, es la que más cálculos puede requerir. Para los nodos visitados que no son puertas de enlace, usar la menor distancia entre dicho nodo y las puertas de enlace o entre dicho nodo y los nodos no visitados. Para los nodos no visitados, se calculará la distancia correspondiente de la misma forma, además, si quedan puertas de enlace por colocar, se colocarán en los nodos que más aporten según este criterio

he de remarcar que esta cota la acabo de adaptar y corregir un poco mientras la explicaba porque al principio la ideé tratando de poner algunas restricciones más, pero seguramente eso hacía que no fuera optimista

la última cota es muy rebuscada, no saldrá en el examen

las anteriores puede, de hecho, alguna salió en el examen de junio

	PROGRAMACIÓN DINÁMICA	VORAZ	BACK-TRACKING	RAMIFICACIÓN Y PODA
MOCHILA DISCRETA	OK	NO		
MOCHILA DISCRETA (sin restricciones adicionales)	NO			
MOCHILA DISCRETA (sin limitación de carga máxima)		OK		
MOCHILA CONTINUA		OK		
CORTE DE TUBOS	OK			
VIAJANTE DE COMERCIO	OK			OK
PROBLEMA DEL CAMBIO	OK	NO		
ASIGNACIÓN DE TAREAS	NO	NO		
FONTANERO DILIGENTE		OK		
TORRES DE HANOI	NO			
FIBONACCI	OK			
PERMUTACIÓN			OK	

Que no te escriban poemas de amor
cuando terminen la carrera ►►►►►



WUOLAH

(a nosotros por suerte nos pasa)

UA | Universitat d'Alacant
Universidad de Alicante

@pepeburgxs

No si antes decirte
Lo mucho que te voy a recordar

Pero me voy a graduar.
Mañana mi diploma y título he de
pagar

Llegó mi momento de despedirme
Tras años en los que has estado mi
lado.

Siempre me has ayudado
Cuando por exámenes me he
agobiado

Oh Wuolah wuolah
Tu que eres tan bonita

	PROGRAMACIÓN DINÁMICA	VORA Z	BACK-TRACKING	RAMIFICACIÓN Y PODA
ÁRBOL DE RECUBRIMIENTO MÍNIMO (de un grafo no dirigido)		OK		
ÁRBOL DE COBERTURA DE COSTE MÍNIMO (de un grafo conexo)		OK		
DIJKSTRA (camino más corto entre un vértice y los restantes)	OK			
WARSHALL (existencia de camino entre cualquier par de vértices)	OK			
FLOYD (caminos más cortos en un grafo entre cualquier par de vértices)	OK			

- Subsecuencia común máxima de dos cadenas.
- Distancia de edición entre dos cadenas.

RESUMEN ADA - PRÁCTICAS

* P. DINÁMICA

- C. ESPACIAL \rightarrow 2 vectores $(2 \cdot n) \in O(n)$
 \hookrightarrow (También con una matriz n·m)

• F. RECURRENCIA: $T(n) = \begin{cases} n^2 & m=1 \\ (n-m)T(m-1) + n^2 & m>1 \end{cases}$

- * R. MATEMÁTICA \rightarrow $m \text{et}(m-1, k) + \text{ogw}(k, n)$

C. TEMPORAL: si usamos P. dinámica $\rightarrow O(n \cdot m)$
(NAIVE)

* RESTRICCIONES:

- Es un vector, cada nodo tiene acceso a su izquierda y derecha.