

Análisis y diseño de algoritmos

2. Eficiencia

José Luis Verdú Mas, Jose Oncina,
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

26 de enero de 2020



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



Contenido





¿Qué es un algoritmo?

Definición (Algoritmo)

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

Importante:

- la **máquina** sobre la que se define el algoritmo debe estar bien definida
- los **recursos** (usualmente tiempo y memoria) necesarios para cada paso elemental deben estar acotados
- El algoritmo debe **terminar** en un número **finito** de pasos



Noción de complejidad

Definición (Complejidad algorítmica)

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

Los recursos mas usuales son:

Tiempo: Complejidad temporal

Memoria: Complejidad espacial

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

Tamaño del problema: lo que ocupa su representación

Parámetro representativo: *i.e.* la dimensión de una matriz



¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

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

¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

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



¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

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



¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

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



¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

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



¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

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

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



¿Cuál es el tamaño de un problema?

Definición (Tamaño de un problema o instancia)

Número de bits que se necesita para codificar una instancia

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

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



Atención:

La complejidad puede depender de cómo se codifique el problema

Ejemplo

Sumar uno a un entero arbitrariamente grande

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

Normalmente se prohíben:

- codificaciones en base uno
- codificaciones no compactas



El tiempo de ejecución

El tiempo de ejecución de un algoritmo depende de:

Factores externos

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

Factores internos

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



¿Cómo estudiamos el tiempo de ejecución?

Definición (Análisis empírico o *a posteriori*)

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

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

Definición (Análisis teórico o *a priori*)

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

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



Tiempo de ejecución de un algoritmo

Definición (Operaciones elementales)

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

Ejemplo (Operaciones elementales)

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



Tiempo de ejecución de un algoritmo

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

Definición (Tiempo de ejecución de un algoritmo)

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



Ejemplo: Suma de los elementos de un vector

Ejemplo (sintaxis de la STL)

```
1 int sumar( const vector<int> &v){  
2     int s = 0;  
3  
4     for(int i = 0; i < v.size(); i++)  
5         s += v[i];  
6  
7     return s;  
8 }
```

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

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

La complejidad del algoritmo será:

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



Ejercicio: Traspuesta de una matriz cuadrada

Traspuesta de una matriz $d \times d$

(Sintaxis de la librería `armadillo`)

```
1 void traspuesta( mat& A){ // supongo que A.n_rows == A.n_cols
2     for( int i = 1; i < A.n_rows; i++ )
3         for( int j = 0; j < i ; j++ )
4             swap( A(i,j), A(j,i) );
5 }
```

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

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

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

$$T_s(s) = T_d(d) = O(d^2) = O(s)$$



Ejercicio: Producto de dos matrices cuadradas

Producto de dos matrices $d \times d$

(Sintaxis de la librería `armadillo`)

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

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

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



Ejercicio: Producto de dos matrices cuadradas

¿Cual es la complejidad con respecto al tamaño?

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

$$T_s(s) = T_d(d) = O(d^3) = O(\sqrt{s/2}^3) = O(s^{3/2})$$

¿Cual es la complejidad espacial?

- En la complejidad espacial no se tiene en cuenta lo que ocupa la codificación del problema.
- Solo se tiene en cuenta lo que es imputable al algoritmo.

$$T_d(d) = d^2$$

$$T_s(s) = T_d(\sqrt{s/2}) = O(s)$$



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

Búsqueda de un elemento

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



Problema

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

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

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





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

Buscar elemento

```
1 int buscar( const vector<int> &v, int z ) {  
2     for( int i = 0; i < v.size(); i++ )  
3         if( v[i] == z )  
4             return i;  
5     return -1;  
6 }  
7
```

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

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

Cotas:

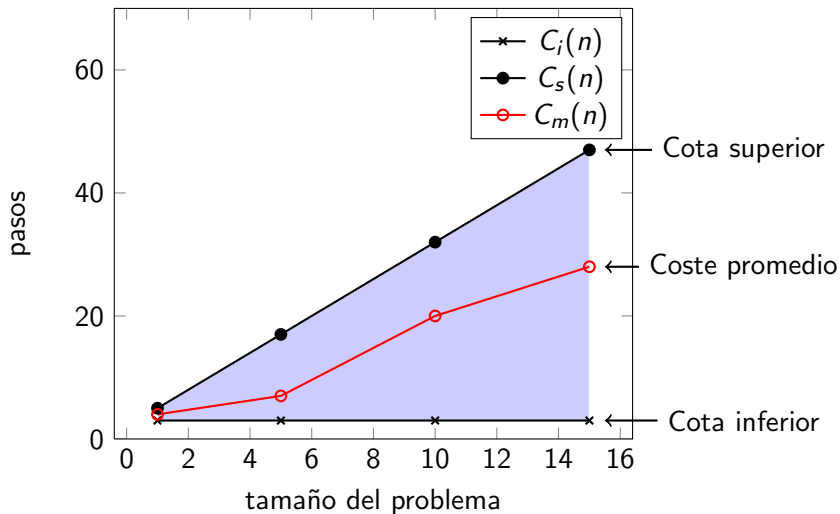
$$C_s(n) = 3n + 2 = O(n)$$

$$C_i(n) = 3 = O(1)$$



Cotas superior e inferior

- Coste de la función buscar



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



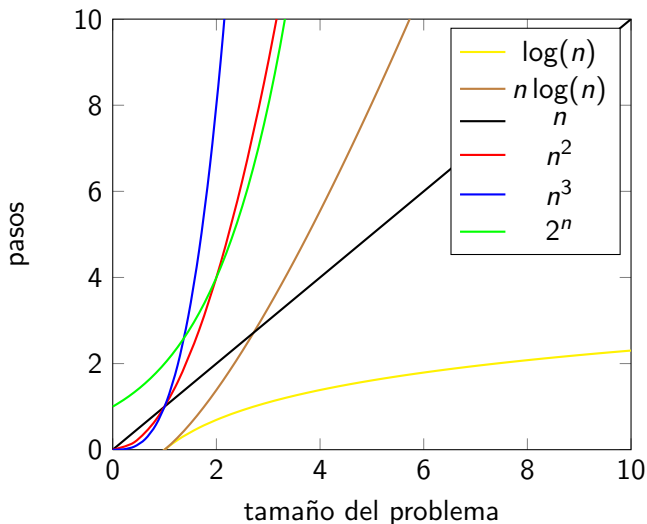
Notación asintótica:

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



¿Para qué sirven?

- Permite agrupar en clases funciones con el mismo crecimiento



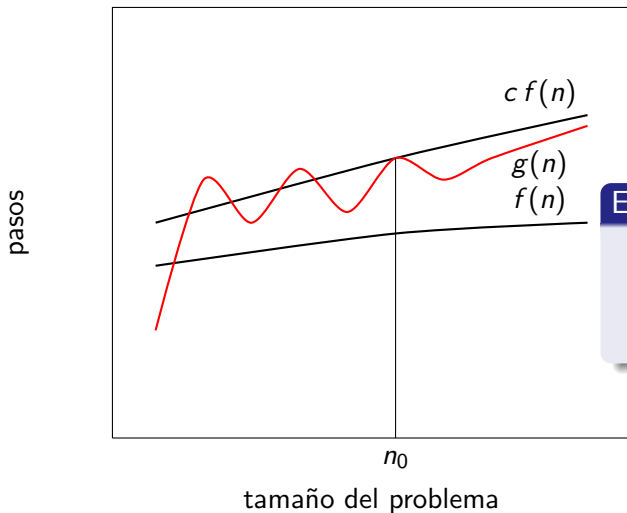
- Sea $f : \mathbb{N} \rightarrow \mathbb{R}^+$; se define el conjunto $O(f)$ como el conjunto de funciones acotadas superiormente por un múltiplo de f :

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

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



Cota superior. Notación O



Ejemplos:

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

$$f \in O(f)$$

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

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

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

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

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

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

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

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

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

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

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

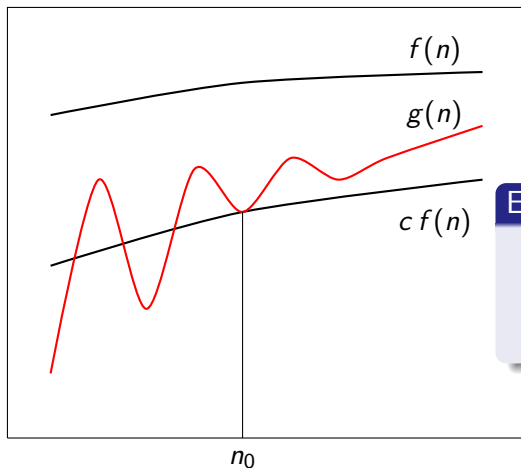
$$\Omega(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0, g(n) \geq cf(n)\}$$

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



Cota inferior. Notación Ω

pasos



tamaño del problema

Ejemplos:

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



$$f \in \Omega(f)$$

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

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

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

$$f \in \Omega(g) \wedge f \in \Omega(h) \Rightarrow f \in \Omega(\max\{g, h\})$$

$$f_1 \in \Omega(g_1) \wedge f_2 \in \Omega(g_2) \Rightarrow f_1 + f_2 \in \Omega(\min(f_1, f_2))$$

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

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

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

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

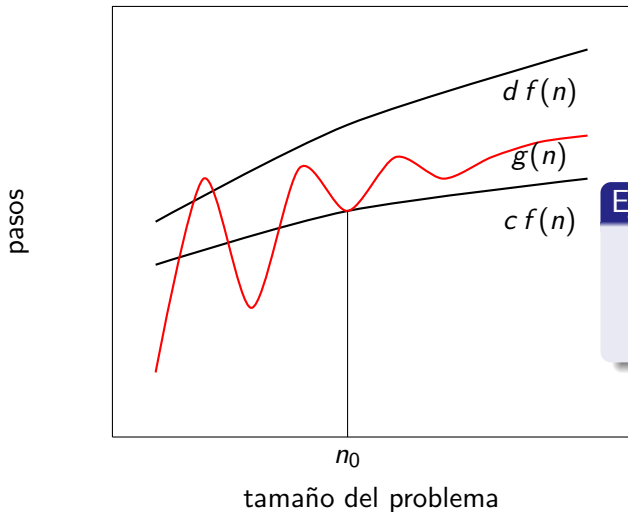
$$\Rightarrow f(n) \in \Omega(n^m)$$

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

$$\Theta(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c, d \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \\ \forall n \geq n_0, cf(n) \leq g(n) \leq df(n)\}$$

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

Coste exacto. Notación Θ



Ejemplos:

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

$$f \in \Theta(f)$$

$$f \in \Theta(g) \Rightarrow \Theta(g) = \Theta(f)$$

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

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

$$f \in \Theta(g) \wedge f \in \Theta(h) \Rightarrow f \in \Theta(\max\{g, h\})$$

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

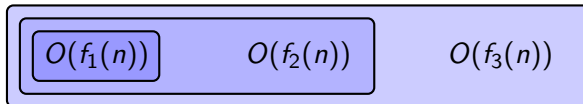
$$f_1 \in \Theta(g_1) \wedge f_2 \in \Theta(g_2) \Rightarrow f_1 f_2 \in \Theta(g_1 g_2)$$

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k, k \neq 0, k \neq \infty \Rightarrow \Theta(f) = \Theta(g)$$

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

Jerarquías de funciones

- Los conjuntos de funciones están incluidos unos en otros generando una ordenación de las diferentes funciones. Por ejemplo, para $O(\cdot)$,



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

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





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

Sumar elementos

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

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



Buscar elemento

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

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

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

$$C_i(n) = 3$$

$$C_s(n) \in O(n)$$

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



Elemento máximo de un vector

```
1 int maximo( const vector<int> &v ) {  
2     int max = v[0];  
3     for( int i = 1; i < v.size(); i++ )  
4         if( v[i] > max )  
5             max = v[i];  
6     return max;  
7 }
```



Búsqueda en un vector ordenado

```
1 int buscar( const vector<int> &v, int x ) {  
2     int pri = 0;  
3     int ult = v.size() - 1;  
4     while( pri < ult ) {  
5         int m = ( pri + ult ) / 2;  
6         if ( v[m] < x )  
7             pri = m + 1;  
8         else  
9             ult = m;  
10    }  
11    if( v[pri] == x )  
12        return pri;  
13    else  
14        return -1;  
15 }
```



- Dado un algoritmo recursivo:

Búsqueda binaria

```
1 int buscar( const vector<int> &v, int pri, int ult, int x){
2     if( pri == ult )
3         return (v[pri] == x) ? pri : -1;
4     int m = ( pri + ult ) / 2;
5     if( v[m] < x )
6         return buscar( v, m+1, ult, x );
7     else
8         return buscar( v, pri, m, x );
9 }
```

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

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



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

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

Donde:

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



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



Ordenación por selección

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

Ordenación por selección (recursivo)

```
1 void ordenar( vector<int> &v, int pri) {  
2     if( pri == v.size() )  
3         return;  
4     int m = pri;  
5     for( int i = pri + 1; i < v.size(); i++ )  
6         if( v[i] < v[m] )  
7             m = i;  
8     swap( v[m], v[pri]);  
9     ordenar(v, pri + 1);  
10 }
```

- Obtener ecuación de recurrencia a partir del algoritmo:

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

donde $n = v.size() - pri$.



- Resolviendo la recurrencia por sustitución

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

Entonces

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

Algoritmo de ordenación por partición o *Quicksort*

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



Quicksort primer elemento

Quicksort

```
1 void quicksort( int v[], int pri, int ult ) {  
2     if( ult <= pri )  
3         return;  
4     int p = pri;  
5     int j = ult;  
6     while( p < j ) {  
7         if( v[p+1] < v[p] ) {  
8             swap( v[p+1], v[p] );  
9             p++;  
10        } else {  
11            swap( v[p+1], v[j] );  
12            j--;  
13        }  
14    }  
15    quicksort(v, pri, p-1);  
16    quicksort(v, p+1, ult);  
17 }
```



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

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

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

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

- Mejor caso:

$$f(n) = n + 2T\left(\frac{n}{2}\right) \quad \text{Rec. 1}$$

$$= n + 2\left(\frac{n}{2} + 2f\left(\frac{n}{2^2}\right)\right) = 2n + 2^2T\left(\frac{n}{2^2}\right) \quad \text{Rec. 2}$$

$$= 2n + 2^2\left(\frac{n}{2^2} + 2f\left(\frac{n}{2^3}\right)\right) = 3n + 2^3T\left(\frac{n}{2^3}\right) \quad \text{Rec. 3}$$

$$= i n + 2^i T\left(\frac{n}{2^i}\right) \quad \text{Rec. } i$$

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

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

Por tanto,

$$T(n) \in \Omega(n \log_2 n)$$

- Peor caso:

$$T(n) = n + T(n-1) \quad \text{Rec. 1}$$

$$= n + (n-1) + T(n-2) \quad \text{Rec. 2}$$

$$= n + (n-1) + (n-2) + T(n-3) \quad \text{Rec. 3}$$

$$= n + (n-1) + (n-2) + \dots + T(n-i) \quad \text{Rec. } i$$

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

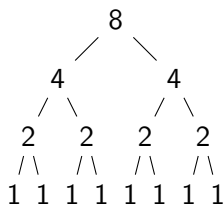
$$= n + (n-1) + (n-2) + \dots + 3 + 2 + T(1)$$

$$= \sum_{j=2}^n j + 1 = \frac{n(n+1)}{2}$$

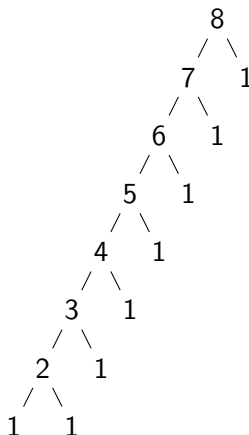
Por tanto,

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

Quicksort



Caso mejor
 $\Omega(n \log_2 n)$



Peor caso
 $O(n^2)$



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



Análisis y diseño de algoritmos

3. El diseño de algoritmos. Divide y vencerás

José Luis Verdú Mas, Jose Oncina,
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

24 de febrero de 2020



1 El diseño de algoritmos

2 Divide y vencerás



El diseño de algoritmos: objetivos

- Dar a conocer las familias más importantes de problemas algorítmicos y estudiar diferentes esquemas o paradigmas de diseño aplicables para resolverlos.
- Aprender a instanciar (particularizar) un esquema genérico para un problema concreto, identificando los datos y operaciones del esquema con las del problema, previa comprobación de que se satisfacen los requisitos necesarios para su aplicación.
- Justificar la elección de un determinado esquema cuando varios de ellos pueden ser aplicables a un mismo problema.



El diseño de algoritmos: definición

- El diseño de algoritmos estudia la aplicación de métodos para resolver problemas en programación.
- La resolución de problemas:
 - Diseño **ad hoc** (frecuentemente “fuerza bruta”)
 - Algoritmos dependientes del problema y no generalizables
 - Dificultad de adecuar cambios en la especificación
 - Esquemas:
 - Cada esquema representa un grupo de algoritmos con características comunes (análogos)
 - Permiten la generalización y reutilización de algoritmos
 - Cada instanciación de un esquema da lugar a un algoritmo diferente



Esquemas algorítmicos más comunes

- Divide y vencerás (*divide and conquer*)
- Programación dinámica (*dynamic programming*)
- Algoritmos voraces (*greedy method*)
- Algoritmos de búsqueda y enumeración
 - Vuelta atrás (*backtracking*)
 - Ramificación y poda (*branch and bound*)
- Algoritmos probabilísticos y heurísticos¹
 - Algoritmos probabilísticos
 - Algoritmos heurísticos
 - Algoritmos genéticos

¹No se tratarán en la asignatura

1 El diseño de algoritmos

2 Divide y vencerás



Ordenación por selección directa

- Ordenar de forma ascendente un vector v de n elementos.

```
1 void selection_sort( Elem v[], int n) {  
2     for( int i=0; i<n-1; i++) {  
3         int m = i;  
4         for( int j = i+1; j<n; j++)  
5             if (v[j] < v[m])  
6                 m = j;  
7         swap(v[i],v[m]);  
8     }  
9 }
```

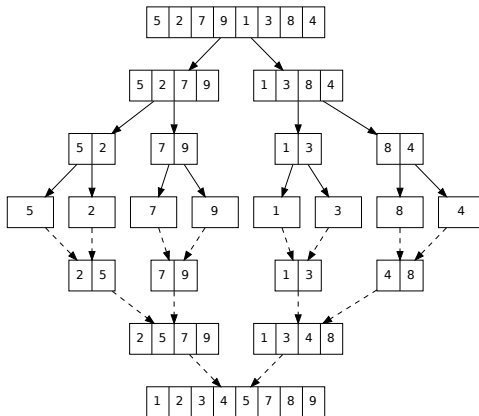
- El bucle de la línea 3 se ejecuta $n - 1$ veces y el bucle de la línea 4 se ejecuta $n - i - 1$ veces con $i \in [0, n - 2]$.
- La línea 5 se ejecuta $\sum_{i=0}^{n-2} (n - i - 1) = \sum_{i=1}^{n-1} i = \frac{1}{2}n(n - 1)$ veces.
- La línea 7 se ejecuta n veces.

Complejidad: $f(n) \in \Theta(n^2)$



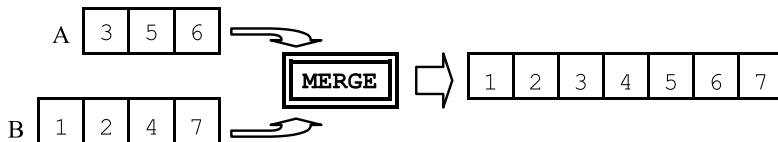
Algoritmo Mergesort: idea

- Ordenar de forma ascendente un vector V de n elementos.
- Solución usando el esquema “divide y vencerás”:



Algoritmo Mergesort: función merge

- El algoritmo mergeSort utiliza la función merge que obtiene un vector ordenado como fusión de dos vectores también ordenados



Mergesort

```
1 void mergeSort(Elem v[], int pi, int pf) {  
2     if ( pi < pf ) { // pi==pf quiere decir 1 elemento  
3         int m = (pi+pf)/2;  
4         mergeSort(v, pi, m);  
5         mergeSort(v, m+1, pf);  
6         merge(v, pi, m, pf);  
7     }  
8 }
```

$\text{merge}(v, pi, m, pf) \in \Theta(n)$ donde $n = pf - pi + 1$.



Algoritmo *Mergesort*: Complejidad

- Talla: n ($n = p_f - p_i + 1$: número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

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



Algoritmo *Mergesort*: Complejidad

- Talla: n ($n = p_f - p_i + 1$: número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

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

¿Cuál es la complejidad espacial?



Algoritmo *Mergesort*: Complejidad

- Talla: n ($n = p_f - p_i + 1$: número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

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

¿Cuál es la complejidad espacial?

- ¿Cuál es la complejidad espacial de merge?



Algoritmo *Mergesort*: Complejidad

- Talla: n ($n = p_f - p_i + 1$: número de elementos del vector)
- Ecuación de recurrencia (coste exacto):

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

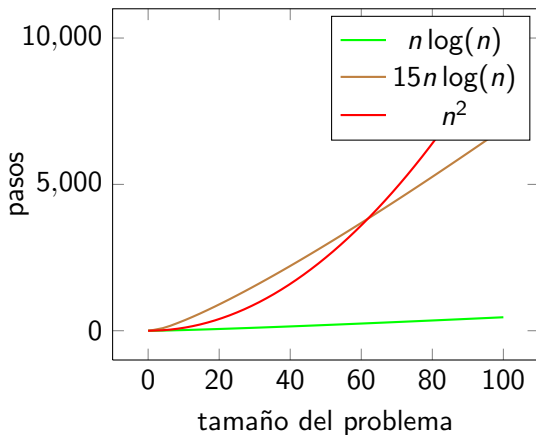
- Complejidad temporal: $f(n) \in \Theta(n \log n)$

¿Cuál es la complejidad espacial?

- ¿Cuál es la complejidad espacial de merge?
- ¡Ojo!: se puede reutilizar la memoria



Mergesort vs. selección



Sólo si el coeficiente de $n \log n$ es mayor que e puede pasar que para ciertos valores pequeños de n , $n \log n$ sea menos ventajoso que n^2 .



Técnica de divide y vencerás

- Técnica de diseño de algoritmos que consiste en:
 - descomponer el problema en subproblemas de menor tamaño que el original
 - resolver cada subproblema de forma individual e independiente
 - combinar las soluciones de los subproblemas para obtener la solución del problema original
- Consideraciones:
 - No siempre un problema de talla menor es más fácil de resolver
 - La solución de los subproblemas no implica necesariamente que la solución del problema original se pueda obtener fácilmente
- Aplicable si encontramos:
 - Forma de descomponer un problema en subproblemas de talla menor
 - Forma directa de resolver problemas menores a un tamaño determinado
 - Forma de combinar las soluciones de los subproblemas que permita obtener la solución del problema original



Esquema de Divide y vencerás

Esquema divide y vencerás

```
1 Solucion DyV( Problema x ) {  
2  
3     if (pequeno(x))  
4         return trivial(x);  
5  
6     list<Solucion> s;  
7     for( Problema q : descomponer(x) )  
8         s.push_back( DyV(q) );  
9     return combinar(s);  
10 }
```



Mergesort como divide y vencerás

Particularización (**instanciación**) del esquema general para el caso de Mergesort:

- **descomponer**: $m = (p_i + p_f)/2$
- **trivial**: retorno sin hacer nada si **pequeño** ($p_i = p_f$)
- **combinar**: `merge(...)`



Quicksort

```
1 void quicksort( Elem v[], int pri, int ult ) {  
2  
3     if( ult <= pri )  
4         return;  
5  
6     int p = pri;    // posicion del pivote  
7     int j = ult;  
8     while(p < j) {  
9         if (v[p+1] < v[p]) {  
10             swap( v[p+1], v[p] );  
11             p++;  
12         } else {  
13             swap( v[p+1], v[j] );  
14             j--;  
15         }  
16     }  
17  
18     quicksort(v, pri, p-1);  
19     quicksort(v, p+1, ult);  
20 }
```

Quicksort como divide y vencerás

Particularización (**instanciación**) del esquema general para el caso de quickSort:

- **descomponer**: cálculo de la posición del elemento pivote
- **trivial**: retorno sin hacer nada si **pequeño** ($ult \leq pri$)
- **combinar**: no es necesario



Análisis de eficiencia (1)

- Eficiencia: costes de logarítmicos a exponenciales.
Depende de:
 - N° de subproblemas (h)
 - Tamaño de los subproblemas
 - Grado de intersección entre los subproblemas
- 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:
 $g(n) \in \Theta(n^k)$

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



- **Teorema de reducción:** los mejores resultados en cuanto a coste se consiguen cuando los subproblemas son aproximadamente del mismo tamaño (y no contienen subproblemas comunes).
- Si se cumple la condición del teorema de reducción ($b = h = a$)

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

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

Las torres de Hanoi: solución

- $\text{hanoi}(n, A \xrightarrow{B} C)$ es la solución del problema: mover los n discos superiores del pivote A al pivote C .
- Supongamos que sabemos mover $n - 1$ discos: sabemos cómo resolver $\text{hanoi}(n - 1, X \xrightarrow{Y} Z)$.
- También sabemos como mover 1 disco del pivote X al Y : $\text{hanoi}(1, X \xrightarrow{Y} Z)$, que es el caso trivial. Lo llamaremos $\text{mover}(X \rightarrow Z)$.
- Resolver $\text{hanoi}(n, A \xrightarrow{B} C)$ equivale a ejecutar:
 - $\text{hanoi}(n - 1, A \xrightarrow{C} B)$
 - $\text{mover}(A \rightarrow C)$
 - $\text{hanoi}(n - 1, B \xrightarrow{A} C)$



Las torres de Hanoi: Complejidad (1)

Nótese que aquí la talla de los subproblemas no es $\frac{n}{a}$ sino $n - 1$:

- No se pueden aplicar las fórmulas generales de las transparencias anteriores
- Divide y vencerás es aquí más una estrategia de solución (un esquema algorítmico) que una manera de conseguir una solución con menor complejidad
 - El problema tiene una complejidad intrínseca peor que las descritas en las transparencias anteriores



Las torres de Hanoi: Complejidad (2)

- Ecuación de recurrencia para el coste exacto (asumiendo coste 1 para todas las operaciones de 1 disco):

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

- Solución:

$$T(n) \stackrel{1}{=} 1 + 2T(n-1)$$

$$\stackrel{2}{=} 1 + 2 + 4T(n-2)$$

$$\stackrel{3}{=} 1 + 2 + 4 + 8T(n-3) = \dots$$

$$\stackrel{k}{=} \sum_{i=1}^k 2^{i-1} + 2^k T(n-k) = 2^k - 1 + 2^k T(n-k)$$

- Paramos cuando $n - k = 1$, osea, en el paso $k = n - 1$,

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

1 Selección del k -ésimo mínimo

- Dado un vector A de n números enteros diferentes, diseñar un algoritmo que encuentre el k -ésimo valor mínimo.

2 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 .

3 Calculo recursivo de la potencia enésima.



Selección del k -ésimo mínimo

Dado un vector A de n enteros, encontrar el k -ésimo valor mínimo.

```
1 Elem quickselect( Elem v[],int pri,int ult,int k ) {
2
3     if( ult == pri )
4         return v[k];
5
6     int p = pri; // pivote
7     int j = ult;
8     while(p < j) {
9         if (v[p+1] < v[p]) {
10             swap( v[p+1], v[p] );
11             p++;
12         } else {
13             swap( v[p+1], v[j] );
14             j--;
15         }
16     }
17
18     if( k == p ) return v[k];
19     if( k < p ) return quickselect(v,pri,p-1,k);
20     return quickselect(v,p+1,ult,k);
21 }
```

Búsqueda binaria o dicotómica

Dado un vector v ordenado de forma ascendente y un elemento e , diseñad un algoritmo que devuelva la posición del elemento en el vector.

Búsqueda binaria

```
1 int bb( Elem v[], int p, int u, Elem e){
2     if( p > u ) return -1 // No hay elementos
3
4     int m = ( p + u ) / 2;
5     if ( e == v[m] ) return m;
6     if ( e < v[m] ) return bb(v,p,m-1,e);
7     return bb(v,m+1,u,e);
8 }
```

- ¿Reconocéis en el algoritmo los componentes del esquema *divide y vencerás*?



- Esta solución se puede ver como un *divide y vencerás* en el que
 - La operación **descomponer** viene representada por $m = (p + u)/2$
 - El problema **pequeno** corresponde a cuando $p > u$
 - Sólo se resuelve uno de los dos subproblemas
 - No es necesaria la combinación



Búsqueda binaria: complejidad

- Ecuación de recurrencia para el caso peor:

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

(agrupamos $n = 0$ en $n = 1$ porque no se produce división).

- Solución:

$$\begin{aligned} T(n) &\stackrel{1}{=} 1 + T(\frac{n}{2}) \\ &\stackrel{2}{=} 2 + T(\frac{n}{2^2}) \\ &\dots \\ &\stackrel{k}{=} k + T(\frac{n}{2^k}) \end{aligned}$$

- Paramos cuando $\frac{n}{2^k} = 1$, o sea en el paso $k = \log(n)$,

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

Cálculo de la potencia enésima

Si asumimos que multiplicar dos elementos de un determinado tipo tiene un coste constante, es posible calcular la enésima potencia x^n de un elemento x de ese tipo en tiempo sublineal usando la siguiente recursión:

$$x^n = \begin{cases} x & n = 1 \\ x^{\frac{n}{2}} x^{\frac{n}{2}} & n \text{ es par} \\ x^{\frac{n-1}{2}} x^{\frac{n-1}{2}} x & n \text{ es impar} \end{cases}$$

Escribid un algoritmo para calcular eficientemente x^n .

- ¿Se puede evitar repetir operaciones?
- ¿Cuál es el coste asintótico del algoritmo resultante?



Análisis y diseño de algoritmos

4. Programación Dinámica

José Luis Verdú Mas, Jose Oncina,
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

8 de marzo de 2020



- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

Problema 1: la mochila 0/1

El problema de la mochila (Knapsack problem):



- Sean n objetos con valores ($v_i \in \mathbb{R}$) y pesos ($w_i \in \mathbb{R}^{>0}$) conocidos
- Sea una mochila con capacidad máxima de carga W
- ¿Cuál es el valor máximo que puede transportar la mochila sin sobrepasar su capacidad?

- Un caso particular: **La mochila 0/1 con pesos discretos**
 - Los objetos no se pueden fraccionar (mochila 0/1 o mochila discreta)
 - La variante más difícil desde el punto de vista computacional
 - Los pesos son cantidades discretas o discretizables
 - Se utilizarán para indexar una tabla
 - Se trata de una versión menos general que suaviza su dificultad



La mochila 0/1. Formalización matemática

- Es un problema de optimización:
 - Secuencia de decisiones: $(x_1, x_2 \dots x_n) : x_i \in \{0, 1\}, 1 \leq i \leq n$
 - En x_i se almacena la decisión sobre el objeto i
 - Si x_i es escogido $x_i = 1$, en caso contrario $x_i = 0$
 - Una secuencia óptima de decisiones es la que maximiza $\sum_{i=1}^n x_i v_i$
sujeto a las restricciones:
 - $\sum_{i=1}^n x_i w_i \leq W$
 - $\forall i : 1 \leq i \leq n, x_i \in \{0, 1\}$
- Representamos mediante $\text{knapsack}(i, C)$ al problema de la mochila con los objetos 1 hasta i y capacidad C
 - El problema inicial es, por tanto, $\text{knapsack}(n, W)$



La mochila 0/1. Subestructura óptima

- Sea $(x_1, x_2 \dots x_n)$ una secuencia óptima de decisiones para el problema $\text{knapsack}(n, W)$
 - Si $x_n = 0$ entonces $(x_1 \dots x_{n-1})$ es una secuencia óptima para el subproblema $\text{knapsack}(n-1, W)$
 - Si $x_n = 1$ entonces $(x_1 \dots x_{n-1})$ es una secuencia óptima para el subproblema $\text{knapsack}(n-1, W - w_n)$

Demostración:

Si existiera una solución mejor $(x'_1 \dots x'_{n-1})$ para cada uno de los subproblemas entonces la secuencia $(x'_1, x'_2 \dots x_n)$ sería mejor que $(x_1, x_2 \dots x_n)$ para el problema original lo que contradice la suposición inicial de que era la óptima.^a

^aEste tipo de demostraciones se denominan “cut and paste”

⇒ La solución al problema presenta una subestructura óptima

- Es decir, la subestructura de los subproblemas puede ser usada para encontrar la solución óptima de el problema completo.



La mochila 0/1. Aproximación matemática

- Se toman decisiones en orden descendente: x_n, x_{n-1}, \dots, x_1
- Ante la decisión x_i hay dos alternativas:
 - Rechazar el objeto i : $x_i = 0$
 - No hay ganancia adicional pero la capacidad de la mochila no se reduce
 - Seleccionar el objeto i : $x_i = 1$
 - La ganancia adicional es v_i , a costa de reducir la capacidad en w_i
- Se selecciona la alternativa que mayor ganancia global resulte
 - No se sabrá mientras no se analicen todas las posibilidades

Solución

$$\{ W \geq 0, n > 0 \}$$

$$\text{knapsack}(0, W) = 0$$

$$\text{knapsack}(n, W) = \max \begin{cases} \text{knapsack}(n-1, W) \\ \text{knapsack}(n-1, W - w_n) + v_n & \text{if } W \geq w_n \end{cases}$$

La mochila 0/1. Solución recursiva ineficiente (ingenua)

Solución recursiva (ineficiente)

```
1 #include <limits>
2
3 double knapsack(
4     const vector<double> &v,      // values
5     const vector<unsigned> &w,    // weights
6     int n,                        // number of objects
7     unsigned W                    // knapsack weight limit
8 ) {
9     if( n == 0 )                  // base case
10         return 0;
11
12     double S1 = knapsack( v, w, n-1, W );    // try not to put it on
13
14     double S2 = numeric_limits<double>::lowest();
15     if( w[n-1] <= W )                // does it fits in the knapsack?
16         S2 = v[n-1] + knapsack( v, w, n-1, W-w[n-1] ); // try to put it on
17
18     return max( S1, S2 );            // choose the best
19 }
```

La mochila 0/1. Coste temporal de la versión ingenua

- Caso mejor: ningún objeto cabe en la mochila: $T(n) \in \Omega(n)$
- Caso peor:

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 + 2T(n-1) & \text{en otro caso} \end{cases}$$

El término general queda como:

$$T(n) = 2^i - 1 + 2^i T(n-i)$$

Que terminará cuando $n-i=0$, o sea:

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

Pero ... ¡solo pueden haber $(n+1)(W+1)$ problemas distintos!

$$\text{mochila}(\{0, 1, \dots, n\}, \{0, 1, \dots, W\})$$



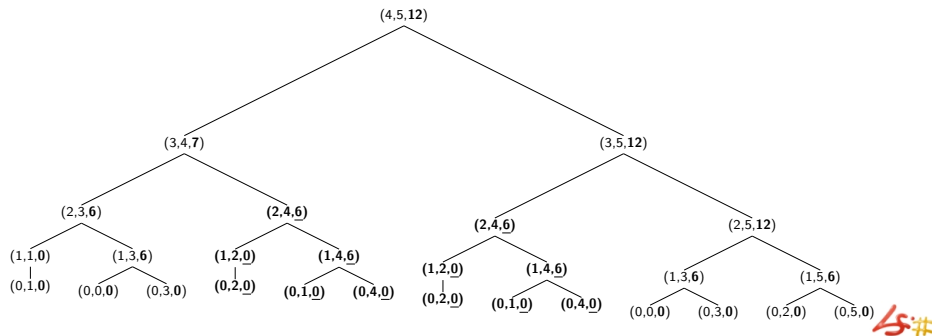
La mochila 0/1. Subproblemas repetidos

En efecto, se resuelven muchos subproblemas iguales:

$$n = 4, \quad W = 5$$

- Ejemplo: $w = (3, 2, 1, 1)$
 $v = (6, 6, 2, 1)$

Nodos: $(i, W, \text{Mochila}(i, W))$; izquierda, $x_i = 1$; derecha, $x_i = 0$.



La mochila 0/1. Solución recursiva con almacén

Solución recursiva eficiente guardando resultados parciales (= memoización)

```
1 const int SENTINEL = -1;
2 double knapsack(
3     vector< vector< double >> &M,                                // Storage
4     const vector<double> &v, const vector<unsigned> &w,        // values & weights
5     int n, unsigned W                                           // num. of objects & Knapsack limit
6 ) {
7     if( M[n][W] != SENTINEL ) return M[n][W];                  // if it is known ...
8     if( n == 0 ) return M[n][W] = 0.0;                         // base case
9
10    double S1 = knapsack( M, v, w, n-1, W );
11    double S2 = numeric_limits<double>::lowest();
12    if (w[n-1] <= W) S2 = v[n-1] + knapsack( M, v, w, n-1, W - w[n-1]);
13    return M[n][W] = max( S1, S2 );                             // store and return the solution
14 }
15 //-----
16 double knapsack(
17     const vector<double> &v, const vector<unsigned> &w, int n, unsigned W
18 ) {
19     vector< vector< double >> M( n+1, vector<double>( W+1, SENTINEL)); // init.
20     return knapsack( M, v, w, n, W );
21 }
```



La mochila 0/1. Memoización

- Ejemplo: Sean $n = 5$ objetos con pesos (w_i) y valores (v_i) indicados en la tabla. Sea $W = 11$ el peso máximo de la mochila.

$M[6][12]$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0		0		0	
$w_1 = 2, v_1 = 1$	0		1	1	1	1	1			1		1
$w_2 = 2, v_2 = 7$	0				8	8	8					8
$w_3 = 5, v_3 = 18$					8	18						26
$w_4 = 6, v_4 = 22$					8							40
$w_5 = 7, v_5 = 28$												40

$$M[i][j] = \max(\underbrace{M[i-1][j]}_{\text{rechazar } i}, \underbrace{M[i-1][j - p_i] + v_i}_{\text{seleccionar } i})$$

El 60 % de las celdas no se usan.

40 Solución al problema
 Contorno o perfil
 Celdas sin usar

$$M[5][11] = \max(\underbrace{M[4][11]}_{\text{Contorno}}, \underbrace{M[4][11 - w_5] + v_5}_{\text{Contorno}}) = \max(40, 36) \quad 5 \text{ no se toma}$$

$$M[4][11] = \max(\underbrace{M[3][11]}_{\text{Contorno}}, \underbrace{M[3][11 - w_4] + v_4}_{\text{Contorno}}) = \max(26, 40) \quad 4 \text{ sí se toma}$$

$$M[3][5] = \max(\underbrace{M[2][5]}_{\text{Contorno}}, \underbrace{M[2][5 - w_3] + v_3}_{\text{Contorno}}) = \max(8, 18) \quad 3 \text{ sí se toma}$$

$$M[2][0] = M[1][0] = 0 \quad 1 \text{ y } 2 \text{ no se toman}$$

La mochila 0/1. Versión iterativa

Una solución iterativa

```
1 double knapsack(  
2     const vector<double> &v, // values  
3     const vector<unsigned> &w, // weights  
4     int last_n, // assessed object  
5     unsigned last_W // Knapsack limit weight  
6 ) {  
7     vector< vector< double >> M( last_n+1, vector<double>(last_W+1));  
8  
9     for( unsigned W = 0; W <= last_W; W++ ) M[0][W] = 0; // no objects  
10    for( unsigned n = 0; n <= last_n; W++ ) M[n][0] = 0; // nothing fits  
11  
12    for( unsigned n = 1; n <= last_n; n++ )  
13        for( unsigned W = 1; W <= last_W; W++ ) {  
14            double S1 = M[n-1][W];  
15            double S2 = numeric_limits<double>::lowest();  
16            if( W >= w[n-1] ) // if it fits ...  
17                S2 = v[n-1] + M[n-1][W-w[n-1]]; // try to put it  
18            M[n][W] = max( S1, S2 ); // store the best  
19        }  
20  
21    return M[last_n][last_W];  
22 }
```



La mochila 0/1. Almacén de la versión iterativa

- Ejemplo: Sean $n = 5$ objetos con pesos (w_i) y valores (v_i) indicados en la tabla. Sea $W = 11$ el peso máximo de la mochila.

$M[6][12]^1$	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 1$	0	0	1	1	1	1	1	1	1	1	1	1
$w_2 = 2, v_2 = 7$	0	0	7	7	8	8	8	8	8	8	8	8
$w_3 = 5, v_3 = 18$	0	0	7	7	8	18	18	25	25	26	26	26
$w_4 = 6, v_4 = 22$	0	0	7	7	8	18	22	25	29	29	30	40
$w_5 = 7, v_5 = 28$	0	0	7	7	8	18	22	28	29	35	35	40

$$M[i][j] = \max(\underbrace{M[i-1][j]}_{\text{rechazar } i}, \underbrace{M[i-1][j - p_i] + v_i}_{\text{seleccionar } i})$$

40

Solución al problema
Contorno o perfil

¡ Se calculan todas las celdas !

$$\begin{aligned}
 M[5][11] &= \max \left(\underline{M[4][11]}, M[4][11 - w_5] + v_5 \right) = \max(40, 36). && 5 \text{ no se toma} \\
 M[4][11] &= \max \left(\underline{M[3][11]}, \underline{M[3][11 - w_4] + v_4} \right) = \max(26, 40). && 4 \text{ sí se toma} \\
 M[3][5] &= \max \left(\underline{M[2][5]}, \underline{M[2][5 - w_3] + v_3} \right) = \max(8, 18). && 3 \text{ sí se toma} \\
 M[2][0] &= M[1][0] = 0. && 1 \text{ y } 2 \text{ no se toman}
 \end{aligned}$$

¹ $M[i][j] \equiv$ Ganancia máxima con los i primeros objetos y con capacidad máxima j . Por tanto, la solución estará en $M[5][11]$

3#

- Complejidad temporal

$$T(n, W) = 1 + \sum_{i=1}^n 1 + \sum_{i=0}^W 1 + \sum_{i=1}^n \sum_{j=1}^W 1 = 1 + n + W + 1 + W(n+1)$$

Por tanto,

$$T(n, W) \in \Theta(nW)$$

- Complejidad espacial

$$S(n, W) \in \Theta(nW)$$

- la complejidad espacial es mejorable ...



La mochila 0/1

- La complejidad temporal de la solución obtenida mediante programación dinámica está en $\Theta(nW)$
 - Un recorrido descendente a través de la tabla permite obtener también, en tiempo $\Theta(n)$, la secuencia óptima de decisiones tomadas.
- Si W es muy grande entonces la solución obtenida mediante programación dinámica no es buena
- Si los pesos w_i o la capacidad W pertenecen a dominios continuos (p.e. los reales) entonces esta solución no sirve
- La complejidad espacial de la solución obtenida se puede reducir hasta $\Theta(W)$
- En este problema, la solución PD-recursiva puede ser más eficiente que la iterativa
 - Al menos, la versión que no realiza cálculos innecesarios es más fácil de obtener en recursivo



Mejoras del algoritmo:

- ¿Se puede reducir la complejidad espacial del algoritmo iterativo propuesto?
 - ¿Cuántos vectores harían falta y de qué tamaño?
 - ¿Se perjudicaría la complejidad temporal?
- Escribe una función para obtener la secuencia de decisiones óptima a partir de la tabla completada para el algoritmo iterativo.
 - ¿Qué complejidad temporal tendría esa función?



La mochila 0/1. Versión iterativa mejorando coste espacial

Solución iterativa economizando memoria

```
1 double knapsack(  
2     const vector<double> &v, const vector<unsigned> &w, // data vectors  
3     int last_n, unsigned last_W      // num. objects & Knapsack limit weight  
4 ) {  
5     vector<double> v0(last_W+1);  
6     vector<double> v1(last_W+1);  
7  
8     for( unsigned W = 0; W <= last_W; W++ ) v0[W] = 0;          // no objects  
9  
10    for( int n = 1; n <= last_n; n++ ) {  
11        for( unsigned W = 1; W <= last_W; W++ ) {  
12            double S1 = v0[W];  
13            double S2 = numeric_limits<double>::lowest();  
14            if( W >= w[n-1] )          // if it fits ...  
15                S2 = v[n-1] + v0[W-w[n-1]]; // try to put it  
16            v1[W] = max( S1, S2 );      // store the best  
17        }  
18        swap(v0,v1);  
19    }  
20    return v0[last_W];  
21 }
```

La mochila 0/1: Extracción de las decisiones /1

Una solución iterativa con almacenamiento de todas las decisiones

```
1 double knapsack(                                // in trace we store the taken decision
2     const vector<double> &v, const vector<unsigned> &w,    // values & weights
3     int last_n, unsigned last_W,                    // assessed object & Knapsack limit
4     vector<vector<bool>> &trace                      // trace (true->store, false->don't)
5 ) {
6     vector< vector< double >> M( last_n+1, vector<double>(last_W+1));
7     trace = vector<vector<bool>>( last_n+1, vector<bool>(last_W+1));
8
9     for( unsigned W = 0; W <= last_W; W++ ) {
10         M[0][W] = 0;                                // no objects
11         trace[0][W] = false;                        // I don't take it
12     }
13
14     for( int n = 1; n <= last_n; n++ )
15         for( unsigned W = 1; W <= last_W; W++ ) {
16             double S1 = M[n-1][W];
17             double S2 = numeric_limits<double>::lowest();
18             if( W >= w[n-1] )                        // if it fits ...
19                 S2 = v[n-1] + M[n-1][W-w[n-1]];    // try to put it
20             M[n][W] = max( S1, S2 );                // store the best
21             trace[n][W] = S2 > S1;                  // if true I take it
22         }
23     return M[last_n][last_W];
24 }
```

La mochila 0/1: Extracción de las decisiones /2

Extracción de las decisiones que interesan

```
1 void parse(  
2     const vector<unsigned> &w,           // weights  
3     const vector<vector<bool>> &trace,    // solutions  
4     vector<bool> &sol  
5 ) {  
6     unsigned last_n = trace.size()-1;  
7     int W = trace[0].size()-1;  
8  
9     for( int n = last_n; n > 0; n-- ) {  
10         if( trace[n][W] ) {  
11             sol[n-1] = true;  
12             W -= w[n-1];  
13         } else {  
14             sol[n-1] = false;  
15         }  
16     }  
17 }
```



La mochila 0/1: Extracción de decisiones (de otra forma)

Extracción de la selección (directamente del almacén)

```
1 void parse(  
2     const vector<vector<double>>> &M,  
3     const vector<double> &v, const vector<unsigned> &w,    // values & weights  
4     int n, unsigned W,                                     // num. of objects & Knapsack limit  
5     vector<bool> &sol  
6 ){  
7     if( n == 0 ) return;  
8  
9     double S1 = M[n-1][W];  
10    double S2 = numeric_limits<double>::lowest();  
11    if (W >= w[n-1] )  
12        S2 = v[n-1] + M[n-1][W-w[n-1]];  
13  
14    if( S1 >= S2 ) {  
15        sol[n-1] = false;  
16        parse( M, v, w, n-1, W, sol );  
17    } else {  
18        sol[n-1] = true;  
19        parse( M, v, w, n-1, W - w[n-1], sol );  
20    }  
21 }
```

- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

Problema 2: Corte de tubos

- Una empresa compra tubos de longitud n y los corta en tubos más cortos, que luego vende
 - El corte le sale gratis
 - El precio de venta de un tubo de longitud i ($i = 1, 2, \dots, n$) es p_i
- Por ejemplo:

longitud i	1	2	3	4	5	6	7	8	9	10
precio p_i	1	5	8	9	10	17	17	20	24	30

- ¿Cual es la forma óptima de cortar un tubo de longitud n para maximizar el precio total?
- Probar todas las formas de cortar es prohibitivo (¡hay $2^{n-1}!$)



Problema 2: Corte de tubos

- Buscamos una descomposición

$$n = i_1 + i_2 + \dots + i_k$$

por la que se obtenga el precio máximo

- El precio es

$$r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$$

- Una forma de resolver el problema recursivamente es:
 - Cortar el tubo de longitud n de las n formas posibles,
 - y buscar el corte que maximiza la suma del precio del trozo cortado p_i y del resto r_{n-i} ,
 - suponiendo que el resto del tubo se ha cortado de forma óptima:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}); \quad r_0 = 0$$



Problema 2: Corte de tubos

Solución recursiva (cálculo de la ganancia máxima)

```
1 int tube_cut(  
2     const vector<int> &p,           // tube length prices  
3     const int l                     // assessed length  
4 ) {  
5  
6     if( l == 0 )  
7         return 0;  
8  
9     int q = numeric_limits<int>::lowest(); //  $q \sim -\infty$   
10    for( int i = 1; i <= l; i++ )  
11        q = max( q, p[i] + tube_cut( p, l-i ) );  
12  
13    return q;  
14 }
```

- Es ineficiente porque hay subproblemas repetidos



Problema 2: Corte de tubos

Solución recursiva (extracción del corte óptimo) /1

```
1 int trace_tube_cut( const vector<int> &p, const int n, vector<int> &trace ) {
2     if( n == 0 ) {      // trace stores for each length which is the optimal cut
3         trace[n] = 0;
4         return 0;
5     }
6     int q = numeric_limits<int>::lowest();
7     for( int i = 1; i <= n; i++ ) {
8         int aux = p[i] + trace_tube_cut( p, n-i, trace);
9         if( aux > q ) {    // Maximum gain
10             q = aux;
11             trace[n] = i;
12         }
13     }
14     return q;
15 }
16 //-----
17 vector<int> trace_tube_cut( const vector<int> &p, const int n ) {
18     vector<int> trace(n+1);
19     trace_tube_cut(p, n, trace);
20     return trace;
21 }
```

Problema 2: Corte de tubos

Solución recursiva (extracción del corte óptimo) /2

```
1 vector<int> parse(  
2     const vector<int> &trace  
3 ) {  
4     vector<int> sol(trace.size(),0);    // How many cuts for each size  
5  
6     int l = trace.size()-1;  
7     while( l != 0 ) {  
8         sol[trace[l]]++;                //where to cut  
9         l = l - trace[l];              // the rest  
10    }  
11    return sol;  
12 }  
13 //-----  
14 ...  
15     vector<int> trace = trace_tube_cut( price, n );  
16     vector<int> sol = parse(trace);  
17     for( unsigned i = 0; i < sol.size(); i++)  
18         if( sol[i] != 0 )  
19             cout << sol[i] << "└cuts└of└length└" << i << endl;  
20 ...
```

Problema 2: Corte de tubos

- Complejidad de la solución recursiva:

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

- Observando que:

$$T(n) = 1 + 2T(n-1)$$

- Tenemos:

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

- Hay 2^{n-1} maneras de cortar el tubo (el árbol de recursión tiene 2^{n-1} hojas).



Problema 2: Corte de tubos

Recursiva con almacén. Ganancia máxima

```
1  const int SENTINEL = -1;
2
3  int tube_cut(
4      vector<int> &M,           // Sub-problem Storage
5      const vector<int> &p, int l
6  ) {
7      if( M[l] != SENTINEL ) return M[l]; // is known?
8
9      if( l == 0 ) return M[0] = 0;
10
11     int q = numeric_limits<int>::lowest();
12     for( int i = 1; i <= l; i++ )
13         q = max( q, p[i] + tube_cut( M, p, l-i));
14
15     return M[l] = q;           // store solution & return
16 }
17
18 int tube_cut( const vector<int> &p, int l ) {
19     vector<int> M(l+1,SENTINEL); // initialization
20     return tube_cut( M, p, l );
21 }
```

- Complejidad espacial: $O(n)$
- Complejidad temporal: $O(n^2)$



Problema 2: Corte de tubos

Recursiva con almacén. Extracción del corte óptimo a partir del almacén

```
1 vector<int> memo_tube_cut(const vector<int> &p, int l){
2     vector<int> M(l+1,SENTINEL); // initialization
3     tube_cut( M, p, l );
4     return M;
5 }
6
7 vector<int> parse( const vector<int> &M, const vector<int> &p ) {
8     vector<int> sol(M.size(), 0);
9
10    int l = M.size() - 1;
11    while( l != 0 ) {
12        for( int i = 1; i <= l; i++ ) {
13            if( M[l] == p[i] + M[l-i] ) {
14                sol[i]++;
15                l -= i;
16                break;
17            }
18        }
19    }
20    return sol;
21 }
```


Problema 2: Corte de tubos

Solución iterativa (directa)

```
1 int tube_cut(  
2     const vector<int> &p,    // tube length prices  
3     int n                    // assessed length  
4 ) {  
5     vector<int> M(n+1);    // Sub-problem storage  
6  
7     for( int l = 0; l <= n; l++ ) {  
8  
9         if( l == 0 ) {          // base case  
10             M[0] = 0;  
11             continue;  
12         }  
13  
14         int q = numeric_limits<int>::lowest();  
15         for( int i = 1; i <= l; i++ )  
16             q = max( q, p[i] + M[l-i] );  
17         M[l] = q;              // Store solution  
18     }  
19     return M[n];  
20 }
```

- Complejidad espacial: $O(n)$
- Complejidad temporal: $O(n^2)$



Problema 2: Corte de tubos

Solución iterativa (mejor)

```
1 int tube_cut(  
2     const vector<int> &p,    // tube length prices  
3     int n                    // assessed length  
4 ) {  
5     vector<int> M(n+1);    // Sub-problem storage  
6  
7     M[0] = 0;              // Base case  
8     for( int l = 1; l <= n; l++ ) {  
9         int q = numeric_limits<int>::lowest();  
10        for( int i = 1; i <= l; i++ )  
11            q = max( q, p[i] + M[l-i] );  
12        M[l] = q;           // Store solution  
13    }  
14  
15    return M[n];  
16 }
```

- Complejidad espacial: $O(n)$
- Complejidad temporal: $O(n^2)$



Problema 2: Corte de tubos

- las dos soluciones con almacén (recursiva descendente e iterativa ascendente) tienen el mismo coste temporal asintótico
- En el iterativo se observa claramente que este coste es $\Theta(n^2)$
- El coste espacial es $\Theta(n)$ (vector p).



- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

¿Qué hemos aprendido con los problemas 1 y 2?

Hay problemas ...

- ... con soluciones recursivas elegantes, compactas e intuitivas
- pero prohibitivamente lentas debido a que resuelven repetidamente los mismos problemas.

Hemos aprendido a:

- **Evitar repeticiones guardando resultados** (*memoización*): usar un almacén para evitar estos cálculos repetidos mejora instantáneamente el coste temporal de las soluciones descendentes (a consta de aumentar la complejidad espacial).
- Aprovechar la **subestructura óptima**: cuando la solución global a un problema incorpora soluciones a problemas parciales más pequeños que se pueden resolver de manera independiente, se puede escribir un algoritmo eficiente.

A esto se le llama **programación dinámica**



¿Qué hemos aprendido con los problemas 1 y 2?

Definición:

Un problema tiene una **subestructura óptima** si una solución óptima puede construirse eficientemente a partir de las soluciones óptimas de sus subproblemas

- Esto también se conoce como **principio de optimalidad**
- Esta es una condición **necesaria** para que se puede aplicar Programación Dinámica



- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial**
- 5 Programación dinámica: Estrategia de diseño

Problema 3: El coeficiente binomial

- **Obtener el valor del coeficiente binomial** $\binom{n}{r}$

Identidad de Pascal: $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$; $\binom{n}{0} = \binom{n}{n} = 1$

(Solución analítica: $\binom{n}{r} = \frac{n!}{r!(n-r)!}$)

Coeficiente binomial

precondición: $\{ n \geq r, n \in \mathbb{N}, r \in \mathbb{N} \}$

```
1 unsigned binomial( unsigned n, unsigned r){
2
3     if ( r == 0 || r == n )
4         return 1;
5
6     return binomial( n-1, r-1 ) + binomial( n-1, r );
7 }
```

- Complejidad temporal (relación de recurrencia múltiple)

$$T(n, r) = \begin{cases} 1 & r = 0 \vee r = n \\ 1 + T(n-1, r-1) + T(n-1, r) & \text{en otro caso} \end{cases}$$

45#

Problema 3: El coeficiente binomial

La solución recursiva es ineficiente.

- Aproximando a una relación de recurrencia lineal:
- Si suponemos:

$$T(n-1, r) \geq T(n-1, r-1)$$

$$T(n, r) \leq g(n, r) = \begin{cases} 1 & n = r \\ 1 + 2g(n-1, r) & \text{en otro caso} \end{cases}$$

$$g(n, r) = 2^k - 1 + 2^k g(n-k, r) \quad \forall k = 1 \dots (n-r)$$

- Por tanto:

$$T(n, r) \sim g(n, r) \in O(2^{n-r})$$



Problema 3: El coeficiente binomial

- Si, en cambio, suponemos:

$$T(n-1, r) \leq T(n-1, r-1)$$

$$T(n, r) \sim g(n, r) = \begin{cases} 1 & r = 0 \\ 1 + 2g(n-1, r-1) & \text{en otro caso} \end{cases}$$

$$g(n, r) = 2^k - 1 + 2^k g(n-k, r-k) \quad \forall k = 1 \dots r$$

- Por tanto:

$$T(n, r) \sim g(n, r) \in O(2^r)$$

Combinando ambos resultados:

$$T(n, r) \sim g(n, r) \in O(2^{\min(r, n-r)})$$

¡Esta solución recursiva no es aceptable!



Problema 3: El coeficiente binomial

Estudio empírico de la complejidad:

$(n, r) = \binom{n}{r}$	Pasos
(40, 0)	1
(40, 1)	79
(40, 2)	1559
(40, 3)	19759
(40, 4)	182779
(40, 5)	1.3×10^6
(40, 7)	3.7×10^7
(40, 9)	5.4×10^8
(40, 11)	4.6×10^9
(40, 15)	8.0×10^{10}
(40, 17)	1.8×10^{11}
(40, 20)	2.8×10^{11}

$(n, r) = \binom{n}{r}$	Pasos
(2, 1)	3
(4, 2)	11
(6, 3)	39
(8, 4)	139
(10, 5)	503
(12, 6)	1847
(14, 7)	6863
(16, 8)	25739
(18, 9)	97239
(20, 10)	369511
(22, 11)	1410863
(24, 12)	5408311

- Caso más costoso: $n = 2r$; crecimiento aprox. 2^n
- los resultados son claramente **prohibitivos**
- Recurrencia aprox.: $f(n) = 1 + 2f(n-1)$; $f(1)=1$



Problema 3: El coeficiente binomial

- ¿Por qué es ineficiente esta solución descendente (*top-down*)?
 - Los problemas se reducen en subproblemas de tamaño similar ($n - 1$).
 - Un problema se divide en dos subproblemas y cada uno de estos en otros dos, y así sucesivamente.

⇒ Esto lleva a complejidades prohibitivas (p.e. exponenciales)
- Pero, ¡el total de subproblemas diferentes no es tan grande!
 - sólo hay nr posibilidades distintas

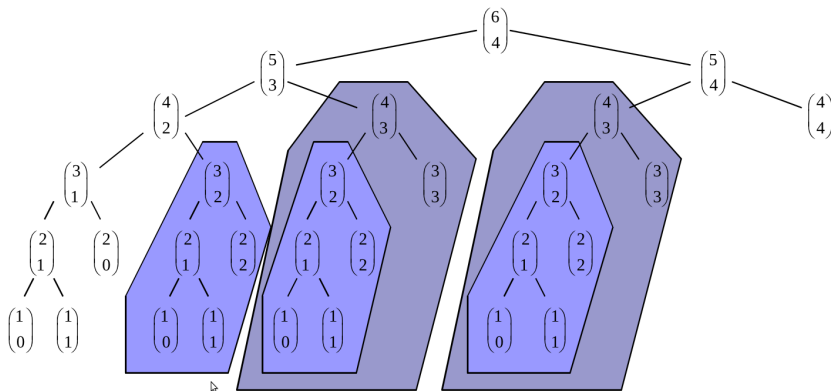
¡La solución recursiva está generando y resolviendo el mismo problema muchas veces!

- ¡Cuidado! la ineficiencia no es debida a la recursividad



Problema 3: El coeficiente binomial

- Solución recursiva: ejemplo para $n = 6$ y $r = 4$



- **INCONVENIENTE:** subproblemas repetidos.
 - Pero sólo hay nr subproblemas diferentes: El problema se puede resolver utilizando almacenes intermedios.



Problema 3: El coeficiente binomial

- Memoización: Almacenamiento de valores ya calculados para no volver a calcularlos.

Una solución recursiva mejorada

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 const unsigned SENTINEL = 0;
2
3 unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {
4
5     if( M[n][r] != SENTINEL )
6         return M[n][r];
7     if( r == 0 || r == n )
8         return 1;
9
10    M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
11
12    return M[n][r];
13 }
14
15 unsigned binomial( unsigned n, unsigned r) {
16     vector<vector<unsigned>> M( n+1, vector<unsigned>(r+1, SENTINEL));
17     return binomial( M, n, r);
18 }
```

Problema 3: El coeficiente binomial

Memoización (para varios problemas)

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial( vector<vector<unsigned>> &M, unsigned n, unsigned r) {
2     if( M[n][r] != 0 ) return M[n][r];
3     if( r == 0 || r == n ) return 1;
4     M[n][r] = binomial(M, n-1, r-1) + binomial(M, n-1, r);
5     return M[n][r];
6 }
7
8 const unsigned MAX_N = 100;
9
10 unsigned binomial( unsigned n, unsigned r) {
11     static vector<vector<unsigned>> M;
12     static bool initialized = false;
13
14     if( !initialized ) {
15         M = vector<vector<unsigned>>(MAX_N, vector<unsigned>(MAX_N, SENTINEL));
16         initialized = true;
17     }
18
19     return binomial( M, n, r);
20 }
```

Problema 3: El coeficiente binomial

Memoización (functors)

$$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$$

```
1 const unsigned SENTINEL = 0;
2 const unsigned MAX_N = 100;
3
4 class Binomial {
5 public:
6     Binomial( unsigned max_n = MAX_N ) : M(
7         vector<vector<unsigned>>(max_n+1, vector<unsigned>(max_n+1, SENTINEL))
8     ){};
9
10    unsigned operator()( unsigned n, unsigned r ) {
11        if( M[n][r] != SENTINEL ) return M[n][r];
12        if( r == 0 || r == n ) return 1;
13        M[n][r] = operator()(n-1, r-1) + operator()(n-1, r);
14        return M[n][r];
15    }
16
17 private:
18     vector<vector<unsigned>> M;
19 };
20
21 Binomial binomial(40); // use: a = binomial(30,20);
```


Problema 3: El coeficiente binomial

- Los resultados mejoran muchísimo cuando se añade un almacén:

$(n, r) = \binom{n}{r}$	Ingenuo	Mem.	$(n, r) = \binom{n}{r}$	Ingenuo	Mem.
(40, 0)	1	1	(2, 1)	3	3
(40, 1)	79	79	(4, 2)	11	8
(40, 2)	1559	116	(6, 3)	20	15
(40, 3)	19759	151	(8, 4)	139	24
(40, 4)	182779	184	(10, 5)	503	35
(40, 5)	1.3×10^{06}	215	(12, 6)	1847	48
(40, 7)	3.7×10^{07}	271	(14, 7)	6863	64
(40, 9)	5.4×10^{08}	319	(16, 8)	25739	80
(40, 11)	4.6×10^{09}	359	(18, 9)	97239	99
(40, 15)	8.0×10^{10}	415	(20, 10)	369511	120
(40, 17)	1.8×10^{11}	432	(22, 11)	1410863	143
(40, 20)	2.8×10^{11}	440	(24, 12)	5408311	168

- En el caso $n = 2r$, el crecimiento es del tipo $(n/2)^2 + n \in \Theta(n^2)$.



Problema 3: El coeficiente binomial

- ¿Se puede evitar la recursividad? En este caso sí
 - Resolver los subproblemas de menor a mayor (recorrido ascendente o *bottom-up*)
 - **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:

$$\begin{aligned} M[i][0] &= 1 & \forall i = 1 \dots (n-r) \\ M[i][i] &= 1 & \forall i = 1 \dots r \end{aligned}$$

Puesto que

$$\binom{m}{0} = \binom{m}{m} = 1, \quad \forall m \in \mathbb{N}$$



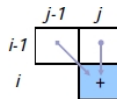
Problema 3: El coeficiente binomial

Recorrido de los subproblemas:

- Los subproblemas se resuelven en sentido ascendente.
- Se almacenan sus soluciones pues harán falta para los siguientes subproblemas.

$$M[i][j] = M[i-1][j-1] + M[i-1][j]$$

$$\forall (i, j) : (1 \leq j \leq r, j+1 \leq i \leq n-r+j)$$



	0	1	2	3	4	$j(r)$
0	1					
1	1	1				
2	1		1			
3				1		
4					1	
5						
6						

$i(n)$







Problema 3: El coeficiente binomial

Una solución iterativa y polinómica (mejorable)

- Ejemplo: Sea $n = 6$ y $r = 4$

	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3		3	3	1	
4			6	4	1
5				10	5
6					15

-  Celdas sin utilizar ¡desperdicio de memoria!
-  Instancias del caso base: perfil o contorno de la matriz
-  Soluciones de los subproblemas. Obtenidos, en este caso, de arriba hacia abajo y de izquierda a derecha
-  Solución del problema inicial. $M[6][4] = \binom{6}{4}$

Solución trivial de programación dinámica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial(unsigned n, unsigned r){
2     unsigned M[n+1][r+1];
3
4     for (unsigned i=0; i <= n-r; i++) M[i][0] = 1;
5     for (unsigned i=1; i <= r; i++) M[i][i] = 1;
6
7     for (unsigned j=1; j<=r; j++)
8         for (unsigned i=j+1; i<=n-r+j; i++)
9             M[i][j] = M[i-1][j-1] + M[i-1][j];
10    return M[n][r];
11 }
```

Problema 3: El coeficiente binomial

Solución trivial de programación dinámica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial(unsigned n, unsigned r){
2     unsigned M[n+1][r+1];
3
4     for (unsigned i=0; i <= n-r; i++) M[i][0] = 1;
5     for (unsigned i=1; i <= r; i++) M[i][i] = 1;
6
7     for (unsigned j=1; j<=r; j++)
8         for (unsigned i=j+1; i<=n-r+j; i++)
9             M[i][j] = M[i-1][j-1] + M[i-1][j];
10    return M[n][r];
11 }
```

- Coste temporal exacto:

$$T(n, r) = 1 + \sum_{i=0}^{n-r} 1 + \sum_{i=1}^r 1 + \sum_{j=1}^r \sum_{i=j+1}^{n-r+j} 1 = rn + n - r^2 + 1 \in \Theta(rn)$$

- Idéntico al descendente con memoización (almacén)



Problema 3: El coeficiente binomial

Solución trivial de programación dinámica

$\{n \geq r, n \in \mathbb{N}, r \in \mathbb{N}\}$

```
1 unsigned binomial(unsigned n, unsigned r){
2     unsigned M[n+1][r+1];
3
4     for (unsigned i=0; i <= n-r; i++) M[i][0]= 1;
5     for (unsigned i=1; i <= r; i++) M[i][i]= 1;
6
7     for (unsigned j=1; j<=r; j++)
8         for (unsigned i=j+1; i<=n-r+j; i++)
9             M[i][j]= M[i-1][j-1] + M[i-1][j];
10    return M[n][r];
11 }
```

- **Complejidad espacial:** $\Theta(rn)$ ¿Se puede mejorar?
- Podéis verlo en <http://v.gd/binCoeff>.



Problema 3: El coeficiente binomial

- Ejercicios propuestos: Reducción de la complejidad espacial:
 - Modificar la función anterior de manera que el almacén no sea más que dos vectores de tamaño $1 + \min(r, n - r)$
 - Modificar la función anterior de manera que el almacén sea un único vector de tamaño $1 + \min(r, n - r)$
 - Con estas modificaciones, ¿queda afectada de alguna manera la complejidad temporal?



- 1 Problema 1: la mochila 0/1
- 2 Problema 2: Corte de tubos
- 3 ¿Qué hemos aprendido con los problemas 1 y 2?
- 4 Problema 3: El coeficiente binomial
- 5 Programación dinámica: Estrategia de diseño

Identificación:

- Diseñar una solución recursiva al problema (top-down)
- Análisis: la complejidad temporal es prohibitiva (p.ex., exponencial)
 - Subproblemas superpuestos
 - Reparto no equitativo de las tallas de los subproblemas
- Si el problema es un problema de optimización, verificar que se puede establecer una subestructura óptima.



Transformación de recursivo a iterativo:

- El siguiente paso es la construcción de la función iterativa (*bottom-up*) a partir de la recursiva (*top-down*)
 - Las llamadas recursivas se sustituyen por accesos a la tabla-almacén
 - Se podría decir, en términos coloquiales, que en el lenguaje de programación se sustituyen paréntesis por corchetes.
 - Sustituir la orden que devuelve el valor en la función recursiva por un almacenamiento en la tabla
 - Utilizar los casos base de la solución recursiva para empezar a rellenar el contorno de la tabla
 - A partir del caso general en la función recursiva, diseñar la estrategia que permita crear los bucles que completen la tabla a partir de los subproblemas resueltos (recorrido ascendente o *bottom-up*)
 - **Es habitual que complejidades exponenciales se transformen en polinómicas**



Programación dinámica recursiva:

- La programación dinámica recursiva consiste en hacer uso del almacén en la versión recursiva
 - La versión recursiva puede ser más eficiente que la iterativa:
 - Evitar los cálculos innecesarios puede ser más fácil en la versión recursiva que en la iterativa
- Por lo tanto, la programación dinámica no implica necesariamente una transformación a iterativo
 - En sus orígenes la transformación a iterativo era un valor añadido pero se debía a que los compiladores no admitían recursividad



Programación dinámica: Estrategia de diseño

Paso de divide y vencerás a programación dinámica

Esquema divide y vencerás

```
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 }
```

Esquema programación dinámica (recursiva)

```
1 Solution DP( Problem p ) {  
2     if( is_solved(p) ) return M[p];  
3     if( is_simple(p) ) return M[p] = trivial(p); // or simply: return trivial(p)  
4  
5     list<Solution> s;  
6     for( Problem q : divide(p) ) s.push_back( DP(q) );  
7     M[p] = combine(s);  
8     return M[p];  
9 }
```

Programación dinámica: Estrategia de diseño

Paso a programación dinámica iterativa:

Esquema programación dinámica (iterativa)

```
1 Solution DP( Problem P) {  
2     vector<Solution> M;  
3     list<Problem> e = enumeration(P);  
4  
5     while( !e.empty() ) {  
6         Problem p = e.pop_front();  
7         if( is_simple(p) )  
8             M[p] = trivial(p);  
9         else {  
10            list<Solution> s;  
11            for( Problem q : divide(p) ) s.push_back( M[q] );  
12            M[p] = combine(s);  
13        }  
14    }  
15    return M[P];  
16 }
```

Le enumeración ha de cumplir:

- todo problema en `divide(p)` aparece antes que `p`
- el problema `P` es el último de la enumeración.



Programación dinámica: casos de aplicación

- 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)



Análisis y diseño de algoritmos

5. Algoritmos voraces

José Luis Verdú Mas, Jose Oncina,
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

23 de marzo de 2020



- 1 Ejemplo introductorio: problema de la mochila continuo
- 2 Algoritmos voraces (Greedy)
- 3 Ejemplos
 - El problema de la mochila discreta
 - El problema del cambio
 - Árboles de recubrimiento de coste mínimo: Prim y Kruskal
 - El fontanero diligente
 - Asignación de tareas



Problema de la Mochila continuo

- Sean n objetos con valores v_i y pesos w_i y una mochila con capacidad máxima de transporte de peso W .
- Seleccionar un conjunto de objetos de forma que:
 - no sobrepase el peso W (restricción)
 - el valor transportado sea máximo (función objetivo)
 - se permite fraccionar los objetos
- El problema se reduce a:
 - Seleccionar un subconjunto de (fracciones de) los objetos disponibles,
 - que cumpla las restricciones, y
 - que maximice la función objetivo.
- ¿Cómo resolverlo mediante un algoritmo voraz?
 - Se necesita un criterio de **selección voraz** que decida qué objeto tomar en cada momento.



- Supongamos el siguiente ejemplo:

$$W = 12 \quad w = (6, 5, 2) \quad v = (48, 35, 20) \quad v/w = (8, 7, 10)$$

Criterios	Solución	Peso W	Valor v
valor decreciente	$(1, 1, \frac{1}{2})$	12	93
peso creciente	$(\frac{5}{6}, 1, 1)$	12	95
valor específico decreciente $(\frac{v_i}{w_i})$	$(1, \frac{4}{5}, 1)$	12	96

- Solución: $X = (x_1, x_2, \dots, x_n)$, $x_i \in [0, 1]$
 - $x_i = 0$: no se selecciona el objeto i
 - $0 < x_i < 1$: fracción seleccionada del objeto i
 - $x_i = 1$: se selecciona el objeto i completo
- Función objetivo:

$$\text{máx} \left(\sum_{i=1}^n x_i v_i \right) \quad (\text{valor transportado})$$

- Restricción:

$$\sum_{i=1}^n x_i w_i \leq W$$

algoritmo voraz (valor óptimo. v1)

```
1 double knapsack(  
2     const vector<double> &v, // values  
3     const vector<double> &w, // weights  
4     double W // knapsack weight limit  
5 ){  
6     vector<unsigned> idx(w.size()); // objects sorted by value density  
7     for( unsigned i = 0; i < idx.size(); i++) idx[i] = i;  
8  
9     sort( idx.begin(), idx.end(), // sort by value density  
10         [&v,&w]( unsigned x, unsigned y ){  
11             return v[x]/w[x] > v[y]/w[y];  
12         }  
13     );  
14     double acc_v = 0.0;  
15     for( unsigned i = 0; i < idx.size(); i++ ) {  
16         if( w[ idx[i] ] > W ) {  
17             acc_v += W/w[ idx[i] ] * v[ idx[i] ];  
18             break;  
19         }  
20         acc_v += v[ idx[i] ];  
21         W -= w[ idx[i] ];  
22     }  
23     return acc_v;  
24 }
```

algoritmo voraz (valor óptimo. v2)

```
1 double knapsack(  
2     const vector<double> &v, // values  
3     const vector<double> &w, // weights  
4     double W // knapsack weight limit  
5 ){  
6     vector<size_t> idx(w.size());  
7     for( size_t i = 0; i < idx.size(); i++) idx[i] = i;  
8  
9     sort( idx.begin(), idx.end(), // sort by value density  
10         [&v,&w]( size_t x, size_t y ) { return v[x]/w[x] > v[y]/w[y]; } );  
11  
12     double acc_v = 0.0;  
13     for( auto i : idx ) {  
14         if( w[i] > W ) {  
15             acc_v += W/w[i] * v[i];  
16             break;  
17         }  
18         acc_v += v[i];  
19         W -= w[i];  
20     }  
21     return acc_v;  
22 }
```

- Complejidad: $\Theta(n \log(n))$



algoritmo voraz (vector óptimo)

```
1 vector<double> knapsack_W(  
2     const vector<double> &v, // values  
3     const vector<double> &w, // weights  
4     double W                // knapsack weight limit  
5 ){  
6     vector<size_t> idx(w.size());  
7     for( size_t i = 0; i < idx.size(); i++) idx[i] = i;  
8     sort( idx.begin(), idx.end(), [&v,&w]( size_t x, size_t y ){  
9         return v[x]/w[x] > v[y]/w[y]; } );  
10  
11     vector<double> x(w.size(),0);  
12     double acc_v = 0.0;  
13     for( auto i : idx ) {  
14         if( w[i] > W ) {  
15             acc_v += W/w[i] * v[i];  
16             x[i] = W/w[i];  
17             break;  
18         }  
19         acc_v += v[i];  
20         W -= w[i];  
21         x[i] = 1.0;  
22     }  
23     return x;  
24 }
```

Teorema: El algoritmo encuentra la solución óptima

Sea $X = (x_1, x_2, \dots, x_n)$ la solución del algoritmo ($\sum_{i=1}^n x_i w_i = W$)

Sea $Y = (y_1, y_2, \dots, y_n)$ otra solución factible ($\sum_{i=1}^n y_i w_i = Q \leq W$)

- De la hipótesis se desprende: $W - Q = \sum_{i=1}^n (x_i - y_i) w_i \geq 0$
 - Hay que demostrar: $V(X) - V(Y) \geq 0$, sabiendo que:
 - $V(X) - V(Y) = \sum_{i=1}^n x_i v_i - \sum_{i=1}^n y_i v_i = \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i}$
 - Sea j la posición del (único) objeto que puede estar fraccionado en X , es decir:
 - Sea $j : x_i = 1 \ \forall i < j$ y $x_i = 0 \ \forall i > j$
 - Si podemos demostrar que $(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j} \ \forall i$,
- concluiremos:

$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) w_i \frac{v_i}{w_i} \geq \frac{v_j}{w_j} \sum_{i=1}^n (x_i - y_i) w_i = \frac{v_j}{w_j} (W - Q) \geq 0$$

Corrección del algoritmo /2

Es cierto que $(x_i - y_i) \frac{v_i}{w_i} \geq (x_i - y_i) \frac{v_j}{w_j} \forall i$?

$i < j$: (parte completamente cargada en X)

- $x_i = 1, y_i \leq 1 \implies x_i - y_i \geq 0$
- $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$
- Se cumple! (por la forma en la que hemos escogido j)

$i = j$: (El elemento que puede estar fraccionado)

- $\frac{v_i}{w_i} = \frac{v_j}{w_j}$
- Se cumple! (ambas partes de la inecuación son iguales).

$i > j$: (parte completamente vacía en X)

- $x_i = 0, y_i \geq 0 \implies x_i - y_i \leq 0$
- $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$
- Se cumple! (puesto que el primer factor es negativo)



- 1 Ejemplo introductorio: problema de la mochila continuo
- 2 Algoritmos voraces (Greedy)
- 3 Ejemplos
 - El problema de la mochila discreta
 - El problema del cambio
 - Árboles de recubrimiento de coste mínimo: Prim y Kruskal
 - El fontanero diligente
 - Asignación de tareas



Algoritmos voraces (*Greedy*)

Definición:

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

Dicho de otra forma:

- Descompone el problema en un conjunto de decisiones *locales*...
- ...y elige la más prometedora
 - Es decir, aquella que se considera mejor para optimizar la medida global.
 - Pero esa decisión puede no conducir a solución óptima (depende del problema).
- Nunca reconsidera las decisiones ya tomadas.
 - Lo que conduce a algoritmos (muy) eficientes.



Esquema voraz

```
1 t_conjuntoElementos VORAZ(t_problema dp)
2 {
3     t_conjuntoElementos y, solucion;
4     elemento decision;
5     y=prepararDatos(dp);           // preparacion de datos para facilitar seleccion
6     while(noVacio(y) || !esSolucion(solucion)) { // quedan datos por seleccionar
7                                           // y aun no se ha llegado a la solucion
8         decision=selecciona(y);
9         if (esFactible(decision,solucion))
10             solucion=anadeElemento(decision,solucion);
11         y=quitaElemento(decision,y); // descartar en cualquier caso
12     }
13     return solucion;
14 }
15
```



Algoritmos voraces: esquema

Esquema voraz (recursivo)

```
1 Solution greedy( Problem p ){
2     Problem sub_prob;
3     Decision decision;
4     if( is_simple(p) )
5         return trivial(p);
6     tie(sub_prob, decision) = select( divide(p) );
7     return Solution( greedy(sub_prob), decision );
8 }
```

Esquema voraz iterativo (iterativo)

```
1 Solucion greedy( Problem p ){
2     Problem sub_prob;
3     Decision decision;
4     Solution solution;
5     while( ! is_simple(p) )
6         solution += select( divide(p) );
7     solution += trivial(p);
8     return solution;
9 }
```

- Son algoritmos eficientes y fáciles de implementar.
- Dependiendo del problema que se esté resolviendo:
 - Puede que no se encuentre la solución óptima;
 - Incluso puede que no se encuentre ninguna solución;
 - Pero si soluciona el problema entonces quizá sea la mejor solución que exista.
- Es necesario un buen criterio de selección para tener garantías de éxito.
- Aún en el caso de que no garantice solución óptima, se aplican mucho:
 - En problemas con muy alta complejidad computacional,
 - Cuando es suficiente una solución aproximada.



- 1 Ejemplo introductorio: problema de la mochila continuo
- 2 Algoritmos voraces (Greedy)
- 3 Ejemplos
 - El problema de la mochila discreta
 - El problema del cambio
 - Árboles de recubrimiento de coste mínimo: Prim y Kruskal
 - El fontanero diligente
 - Asignación de tareas



Problema de la mochila discreta (sin fraccionamiento)

- Sean n objetos con valores v_i y pesos w_i y una mochila con capacidad máxima de transporte peso W . Seleccionar un conjunto de objetos de forma que:
 - no sobrepase el peso W
 - el valor transportado sea máximo
- Formulación del problema:
 - Expresaremos la solución mediante un vector (x_1, x_2, \dots, x_n) donde x_i representa la decisión tomada con respecto al elemento i .
 - Función objetivo:

$$\text{máx} \left(\sum_{i=1}^n x_i v_i \right) \quad (\text{valor transportado})$$

- Restricciones

$$\sum_{i=1}^n x_i w_i \leq W \quad x_i \in \{0, 1\} \begin{cases} x_i = 0 & \text{no se selecciona el objeto } i \\ x_i = 1 & \text{sí se selecciona el objeto } i \end{cases}$$



problema de la mochila discreta (sin fraccionamiento)

- En este caso el método voraz no resuelve el problema.
- Ejemplo:

$$W = 120 \quad w = (60, 60, 20) \quad v = (300, 300, 200) \quad v/w = (5, 5, 10)$$

- solución voraz: $(0, 1, 1) \rightarrow \text{valor total} = 500$
- solución óptima: $(1, 1, 0) \rightarrow \text{valor total} = 600$

⇒ La selección por valor específico no conduce al óptimo.

- No se conoce ningún criterio de selección (voraz o no) que conduzca al óptimo ante cualquier instancia de este problema.
- Aún así, esta solución se utiliza mucho como aproximación al óptimo (en esto consisten las heurísticas voraces).



Heurística voraz para resolver 'la mochila discreta'

```
1 double knapsack_d(  
2     const vector<double> &v,  
3     const vector<double> &w,  
4     double W  
5 ) {  
6     vector<size_t> idx( w.size() );  
7     for( size_t i = 0; i < idx.size(); i++) idx[i] = i;  
8  
9     sort( idx.begin(), idx.end(), [&w,&v]( size_t x, size_t y ){  
10         return v[x]/w[x] > v[y]/w[y];  
11     } );  
12  
13     double acc_v = 0.0;  
14  
15     for( auto i : idx ) {  
16  
17         if( w[i] < W ) {  
18             acc_v += v[i];  
19             W -= w[i];  
20         }  
21     }  
22  
23     return acc_v;  
24 }
```

El problema del cambio

- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C :

- Una solución es una secuencia de decisiones

$$S = (s_1, s_2, \dots, s_n)$$

- La función objetivo es

$$\text{mín } |S|$$

- La restricción es

$$\sum_{i=1}^n \text{valor}(s_i) = M$$

- La solución voraz es tomar en cada momento la moneda de mayor valor posible.



Ejemplo

- Consiste en formar una suma M con el número mínimo de monedas tomadas (con repetición) de un conjunto C :
- Sea $M = 65$

C	S	n	Solución
$\{1, 5, 25, 50\}$	$(50, 5, 5, 5)$	4	óptima
$\{1, 5, 7, 25, 50\}$	$(50, 7, 7, 1)$	4	óptima
	$(50, 5, 5, 5)$	4	no voraz
$\{1, 5, 11, 25, 50\}$	$(50, 11, 1, 1, 1, 1)$	6	factible pero no óptima
$\{5, 11, 25, 50\}$	$(50, 11, ?)$???	no encuentra solución



Árbol de recubrimiento de coste mínimo

- Partimos de un grafo $g = (V, A)$:
 - conexo
 - ponderado
 - no dirigido
 - con arcos positivos
- Queremos el árbol de recubrimiento de g de coste mínimo:
 - subgrafo de g
 - con todos los vértices (recubrimiento)
 - sin ciclos (árbol)
 - conexo (árbol)
 - coste mínimo



Algoritmos de Prim y Kruskal

- Existen al menos dos algoritmos voraces que resuelven este problema,
 - algoritmo de Prim
 - algoritmo de Kruskal
- En ambos se van añadiendo arcos de uno en uno a la solución
- la diferencia está en la forma de elegir los arcos a añadir



Algoritmo de Prim básico

- 1: Datos: $G = (V, A)$
- 2: Resultado: $Sol \subseteq A$
- 3: Auxiliar: $V' \subseteq V$
- 4: Auxiliar: $(v, v') \in A$
- 5: $Sol \leftarrow \emptyset$
- 6: $v \leftarrow \text{elementoAleatorio}(V)$
- 7: $V' \leftarrow \{v\}$
- 8: **while** $V' \neq V$ **do**
 - 9:
 - ▷ Selección: Arista de menor peso con un vértice visitado y el otro no
 - 10: $(v, v') \leftarrow \text{aristaMenorPeso}(A)$ **con** $v \in V - V'$ **y** $v' \in V'$
 - 11: $Sol \leftarrow Sol \cup \{(v, v')\}$
 - ▷ Se añade esa arista a la solución
 - 12: $V' \leftarrow V' \cup \{v\}$
 - ▷ Se visita el vértice v
- 13: **end while**
 - ▷ Grafo no dirigido, conexo y ponderado
 - ▷ Árbol de expansión mínimo
 - ▷ Conjunto de vértices visitados
 - ▷ Arista seleccionada
 - ▷ Inicialmente el árbol está vacío
 - ▷ Partimos de cualquier vértice de V
 - ▷ Se visita ese vértice
 - ▷ Mientras no se hayan visitado todos los vértices

Algoritmo de Prim

- Se mantiene un conjunto de vértices explorados
- Se coge un vértice al azar y se añade al conjunto de explorados
- en cada paso:
 - buscar el arco de mínimo peso que va de un vértice explorado a uno que no lo está
 - añadir el arco a la solución y el vértice a los explorados



Algoritmo de Prim

Algoritmo de Prim (ineficiente)

```
1 list<edge> prim( const Graph &g ) {
2     int n = g.size();
3     vector<bool> visited( n, false);
4     list<edge> r;
5
6     edge e{-1,0};
7     for( int i = 0; i < n-1; i++ ) {
8         visited[e.d] = true;
9
10        e = min_edge( g, visited );
11
12        r.push_back(e);
13    }
14    return r;
15 }
16 }
```

Estructuras de datos

```
1 typedef vector<vector<int>>> Graph;
2
3 struct edge {
4     int s;
5     int d;
6 };
7
8 // Graph instantiation example
9 // (999 == \infty)
10
11 Graph g{
12     { 999, 3, 1, 6, 999, 999 },
13     { 3, 999, 5, 999, 3, 999 },
14     { 1, 5, 999, 5, 6, 4 },
15     { 6, 999, 5, 999, 999, 2 },
16     { 999, 3, 6, 999, 999, 6 },
17     { 999, 999, 4, 2, 6, 999 }
18 };
```


Algoritmo de Prim (ineficiente)

```
1 edge min_edge(  
2     const Graph& g,  
3     const vector<bool> &visited  
4  
5 ) {  
6     int n = g.size();  
7     int min = numeric_limits<int>::max();  
8     edge e;  
9     for( int i = 0; i < n; i++ )  
10         for( int j = 0; j < n; j++ )  
11             if( visited[i] && !visited[j] )  
12                 if( g[i][j] < min ) {  
13                     min = g[i][j];  
14                     e.s = i;  
15                     e.d = j;  
16                 }  
17  
18     return e;  
19 }
```

- complejidad
min-edge: $O(V^2)$
- complejidad de prim:
 $O(V^3)$
- ¿se puede mejorar?



Mejora:

- No hace falta recorrer todos los arcos cada vez
- Si cambia el mínimo es a causa del último vértice añadido
- Hay que guardarse, para cada vértice, el último mínimo
 - 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, en f_i está el vértice origen de esa arista (representada por w_i)



Algoritmo de Prim

Algoritmo de Prim (con indices)

```
1 list<edge> prim( const Graph &g ) {
2     int n = g.size();
3     vector< bool > visited( n, false);           // visited vertex
4     vector<int> w(n, numeric_limits<int>::max() ); // previous min's
5     vector<int> f(n);                             // father of the min's
6     list<edge> r;
7
8     edge e{-1,0};
9     for( int i = 0; i < n-1; i++ ) {
10
11         visited[e.d] = true;
12         update_idx( g, w, f, e.d );              // update index
13
14         e = min_edge( g, w, f, visited );
15
16         r.push_back(e);
17     }
18     return r;
19 };
```

Algoritmo de Prim

Actualizar índices

```
1 void update_idx(  
2     const Graph &g,  
3     vector<int> &w,  
4     vector<int> &f,  
5     int nv  
6 ) {  
7  
8     int n = g.size();  
9     for( int j = 0; j < n; j++ )  
10         if( w[j] > g[nv][j] ) {  
11             w[j] = g[nv][j];  
12             f[j] = nv;  
13         }  
14 }
```

Buscar mejor arco

```
1 edge min_edge(  
2     const Graph &g,  
3     const vector<int> &w,  
4     vector<int> &f,  
5     const vector<bool> &visited  
6 ) {  
7     int n = g.size();  
8  
9     int min = numeric_limits<int>::max();  
10    edge e;  
11    for( int j = 0; j < n; j++ ) {  
12        if( !visited[j] && w[j] < min ) {  
13            min = w[j];  
14            e.s = f[j];  
15            e.d = j;  
16        }  
17    }  
18    return e;  
19 }
```

Algoritmo de Kruskal

Algoritmo de Kruskal básico

```
1: Datos:  $G = (V, A)$ 
2: Resultado:  $Sol \subseteq A$ 
3: Auxiliar:  $A' \subseteq A$ 
4: Auxiliar:  $(u, v) \in A$ 
5:  $Sol \leftarrow \emptyset$ 
6:  $A' \leftarrow \text{ordenarPesosCreciente}(A)$ 
7: while  $|Sol| < |V| - 1 \wedge A' \neq \emptyset$  do
8:    $(u, v) \leftarrow \text{primero}(A')$ 
9:   if  $\text{noCreaCiclo}((u, v), Sol)$  then
10:     $Sol \leftarrow Sol \cup \{(u, v)\}$ 
11:   end if
12:    $A' \leftarrow A' - \{(u, v)\}$ 
13: end while
14: if  $|Sol| = |V| - 1$  then
15:   return  $Sol$ 
16: else
17:   return  $\emptyset$ 
18: end if
```

▷ Grafo no dirigido, conexo y ponderado
▷ Árbol de expansión mínimo
▷ Conjunto ordenado de aristas
▷ Arista seleccionada
▷ Comenzamos con un bosque vacío
▷ Preparar conjunto de aristas
▷ Selección: Arista de mínimo peso aún sin considerar
▷ Ha de conectar dos árboles existentes,
▷ o bien, ser un nuevo árbol (nueva componente conexa)
▷ Se añade a la solución
▷ En cualquier caso se descarta la arista
▷ Grafo no conexo: no existe árbol de recubrimiento

¿Cómo implementar la función **noCreaCiclo**(...)?

- Una posibilidad que no sirve:
 - 1 Marcar cada vértice que se selecciona
 - 2 Una arista forma parte de la solución si sus dos vértices no están ya marcados
 - Sirve para descartar ciclos, pero ...
 - El problema es que descarta aristas que conectan dos componentes conexas
- Solución:
 - Las estructuras de conjuntos disjuntos (*Disjoint-set*)



Conjuntos disjuntos:

- También llamado TAD unión-búsqueda (*union-find*) por las operaciones que comprende.
- Tenemos una partición de un conjunto de datos y queremos:
 - Inicializar la partición: cada elemento en un bloque distinto
 - Poder unir dos bloques de la partición (*union*)
 - Saber a qué bloque pertenece un elemento (*find*)



Aplicándolo al algoritmo de Kruskal:

- mantener una partición de los vértices
- mientras queda más de un bloque
 - buscar el arco de menor peso que una dos bloques distintos
 - (Esto asegura que no habrán ciclos)
 - añadir el arco a la partición
 - unir los dos bloques



Algoritmo de Kruskal. Conjuntos disjuntos (*Disjoint-set*)

Una forma de abordarlo:

- Mediante un vector de etiquetas.
- Asignamos una etiqueta distinta a cada partición,
- **union**: reetiquetamos los elementos de uno de los bloques,
 - De manera que ambos bloques tengan la misma etiqueta (unión)
 - Complejidad $O(n)$
- **find**: consultar la etiqueta.
 - Complejidad: $O(1)$

Esta implementación básica produce una complejidad temporal del alg. de Kruskal $O(A \log A + V^2)$

- $O(A \log A)$ por la ordenación de las aristas
- $O(V^2)$ por la operación **union** para las $|V| - 1$ aristas seleccionadas
- Otras implementaciones más eficientes de conjuntos disjuntos llevan a una complejidad: $O(A \log A)$



Algoritmo de Kruskal

```
1 list<edge> kruskal( const Graph &g ) {
2     struct node { int w; edge e; };
3
4     int n = g.size();
5     list<edge> r;
6     disjoint_set s(n);
7
8     vector<node> v;
9     for( int i = 1; i < n; i++ )
10         for( int j = 0; j < i; j++ )
11             v.push_back({ g[i][j], {i, j} });
12
13     sort( v.begin(), v.end(), []( const node &n1, const node &n2 ) {
14         return n1.w < n2.w;
15     });
16
17     for( auto n : v ) {
18         if( s.find(n.e.s) != s.find(n.e.d) ) {
19             r.push_back(n.e);
20             s.merge( n.e.s, n.e.d );
21         }
22     }
23     return r;
24 }
```

El fontanero diligente

- Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i -ésima tardará t_i minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes.
- En otras palabras, si llamamos E_i a lo que espera el cliente i -ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E(n) = \sum_{i=1}^n E_i$$



La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i .
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij} , donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j , y $x_{ij} = 1$ indica que sí.
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador.
- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^n \sum_{j=1}^n x_{ij} b_{ij}$$

- Diremos que una asignación es óptima si es de mínimo coste.



Análisis y diseño de algoritmos

6. Vuelta atrás

José Luis Verdú Mas, Jose Oncina,
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

30 de marzo de 2020

- 1 Ejemplo introductorio: El problema de la mochila (general)
- 2 Vuelta atrás
- 3 Ejercicios
 - Permutaciones
 - El viajante de comercio
 - El problema de las n reinas
 - La función compuesta mínima
- 4 Ejercicios propuestos

1 Ejemplo introductorio: El problema de la mochila (general)

2 Vuelta atrás

3 Ejercicios

- Permutaciones
- El viajante de comercio
- El problema de las n reinas
- La función compuesta mínima

4 Ejercicios propuestos

El problema de la mochila (general)

Dados:

- n objetos con valores v_i y pesos w_i
- una mochila que solo aguanta un peso máximo W

Seleccionar un conjunto de objetos de forma que:

- no se sobrepase el peso límite W (restricción)
- el valor transportado sea máximo (función objetivo)

• **¿Cómo obtener la solución óptima?**

- Programación dinámica: objetos no fragmentables y pesos discretos
- Algoritmos voraces: objetos fragmentables

• **¿Cómo lo resolvemos si no podemos fragmentar los objetos y los pesos son valores reales?**

Formalización del problema

- Solución: $X = (x_1, x_2, \dots, x_n)$ $x_i \in \{0, 1\}$
- Restricciones:
 - Implícitas:

$$x_i \in \begin{cases} 0 & \text{no se selecciona el objeto } i \\ 1 & \text{se selecciona el objeto } i \end{cases}$$

- Explícitas:

$$\sum_{i=1}^n x_i w_i \leq W$$

- Función objetivo:

$$\text{máx} \sum_{i=1}^n x_i v_i$$

Tipos de soluciones

- Supongamos el siguiente ejemplo:

$$W = 16$$

$$w = (2, 8, 7)$$

$$v = (20, 40, 49)$$

- Combinaciones posibles (espacio de soluciones):

Solución **Peso** **Valor**

(0, 0, 0)	0	0
(0, 0, 1)	7	49
(0, 1, 0)	8	40
(0, 1, 1)	15	89
(1, 0, 0)	2	20
(1, 0, 1)	9	69
(1, 1, 0)	10	60
(1, 1, 1)	17	109

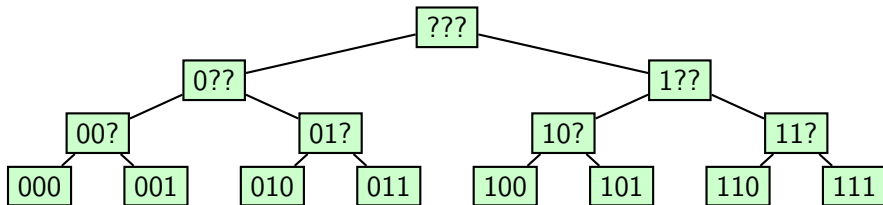
Soluciones factibles

Solución óptima

Solución voraz

Solución NO factible

Generación de todas las combinaciones



Recorrer todas las combinaciones.

Llamada inicial: combinaciones(0,x)

```
1 void combinaciones(unsigned k, vector <short> &x){
2     if( k == x.size() ) {                // It is a leaf
3         cout << x << endl;
4         return;
5     }                                    // It is not a leaf
6     for (unsigned j=0; j<2; j++) {
7         x[k]=j;                          // New alternative
8         combinaciones(k+1, x);           // expand
9     }
10 }
```

Complejidad temporal: $\Theta(n2^n)$

¿Cómo generar solo soluciones factibles?

- Solo imprimimos la soluciones (hojas) que cumplen:

$$\sum_{i=1}^n x_i w_i \leq W$$

Generación de todas las soluciones factibles

Generar soluciones factibles

Llamada inicial: feasible(w, W, 0, x)

```
1 double weight( const vector<double> &w, const vector<short> &x){
2     double acc_w = 0.0;
3     for (unsigned i = 0; i < w.size(); i++ ) acc_w += x[i] * w[i];
4     return acc_w;
5 }
6 //-----
7 void feasible( const vector<double> &w, double W, unsigned k, vector<short> &x){
8     if( k == x.size() ) {                                // It is a leaf
9         if( weight( w, x ) <= W )
10             cout << x << endl;
11         return;
12     }                                                     // It is not a leaf
13     for (unsigned j=0; j<2; j++) {
14         x[k]=j;
15         feasible(w,W,k+1,x);                             // expand
16     }
17 }
```

Complejidad temporal: $\Theta(n2^n)$

¿Podemos acelerar el programa?

- Sí, evitando explorar ramas que no pueden dar soluciones factibles
- Por ejemplo:
 - hemos construido una solución hasta el elemento k :

$$(x_1, x_2, \dots, x_k, ?, \dots, ?)$$

- si tenemos que:

$$\sum_{i=1}^k x_i w_i \geq W$$

⇒ Ninguna expansión de esta solución puede ser factible

```

1 double weight(const vector<double>&w, unsigned k, const vector<short>&x ){
2     double acc_w = 0;
3     for ( unsigned i = 0; i < k; i++ ) acc_w += x[i] * w[i];
4     return acc_w;
5 }
6 //-----
7 void feasible(
8     const vector<double> &w, double W, unsigned k, vector<short> &x
9 ){
10     if( k == x.size() ) {                                // if it is a leaf
11         cout << x << endl;
12         return;
13     }                                                    // if it's not a leaf
14     for (unsigned j = 0; j < 2; j++ ) {
15         x[k]=j;
16         if ( weight(w, k+1, x ) <= W )                    // if it is feasible
17             feasible( w, W, k+1, x );                    // Expand
18     }
19 }

```

- **Peor caso** Todos los objetos caben: $O(n2^n)$
- **Mejor caso** Ningún objeto cabe: $\Omega(n^2)$

¿se puede mejorar?

- Nótese que el peso se puede ir calculando a medida que se rellena la solución

Eliminando ramas por peso (incremental)

feasible(w, W, 0, x, 0)

```
1 void feasible(  
2     const vector<double> &w, double W, unsigned k,  
3     vector<short> &x, double acc_w  
4 ){  
5     if( k == x.size() ) { // if it is a leaf  
6         cout << x << endl;  
7         return;  
8     }  
9 // if it is not a leaf  
10    for (unsigned j = 0; j < 2; j++ ) {  
11        x[k]=j;  
12        double current_w = acc_w + x[k] * w[k]; // update weight  
13        if ( current_w <= W ) // if it is feasible  
14            feasible(w, W, k+1, x, current_w); // Expand  
15    }  
16 }
```

- **Peor caso** Todos los objetos caben: $O(n 2^n)$
- **Mejor caso** Ningún objeto cabe: $\Omega(n)$

Calculando la solución óptima

- Para calcular la solución óptima hay que:
 - recorrer todas las soluciones factibles
 - calcular su valor y ...
 - ... quedarse con la mejor de todas

```

1 void knapsack(
2     const vector<double> &v, const vector<double> &w,
3     double W, unsigned k, vector<short> &x,
4     double acc_w, double &best_v
5 ){
6     if( k == x.size() ) {                                // if it is a leaf
7         best_v = max( best_v, value(v,x));
8         return;
9     }
10                                     // it is not a leaf
11     for (unsigned j = 0; j < 2; j++ ) {
12         x[k]=j;
13         double current_w = acc_w + x[k] * w[k];          // update weight
14         if ( current_w <= W )                             // it is feasible
15             knapsack(v, w, W, k+1, x, current_w, best_v); // expand
16     }
17 }
    
```

- **Peor caso** Todos los objetos caben: $O(2^n)$
- **Mejor caso** Ningún objeto cabe: $\Omega(n^2)$

¿Se puede mejorar?

- El valor se puede ir calculando a medida que se rellena la solución

```
1 void knapsack(  
2     const vector<double> &v, const vector<double> &w,  
3     double W, unsigned k, vector<short> &x,  
4     double acc_w, double acc_v, double &best_v  
5 ){  
6     if( k == x.size() ) { // if it is a leaf  
7         best_v = max( best_v, acc_v);  
8         return;  
9     }  
10 // it is not a leaf  
11     for (unsigned j = 0; j < 2; j++ ) {  
12         x[k]=j;  
13         double current_w = acc_w + x[k] * w[k]; // update weight  
14         double current_v = acc_v + x[k] * v[k]; // update value  
15         if ( current_w <= W ) // if it is feasible  
16             knapsack( v, w, W, k+1, x, current_w, current_v, best_v);  
17     }  
18 }
```

- **Peor caso** Todos los objetos caben: $O(2^n)$
- **Mejor caso** Ningún objeto cabe: $O(n)$

¿Podemos acelerar el programa?

- Sí, evitando explorar ramas que no pueden dar soluciones mejores la que ya tenemos.
- Por ejemplo:
 - hemos construido una solución hasta el elemento k :

$$(x_1, x_2, \dots, x_k, ?, \dots, ?)$$

- ya hemos encontrado una solución factible de valor v_b
- si tenemos que:

$$\sum_{i=1}^k x_i v_i + \sum_{i=k+1}^n v_i \leq v_b$$

⇒ Ninguna expansión de esta solución puede dar un valor mayor que v_b

```

1 void knapsack(
2     const vector<double> &v, const vector<double> &w,
3     double W, unsigned k, vector<short> &x,
4     double acc_w, double acc_v, double &best_v
5 ){
6     if( k == x.size() ) {                                // if it is a leaf
7         best_v = acc_v;
8         return;
9     }
10
11                                     // it is not a leaf
12     for (unsigned j = 0; j < 2; j++ ) {
13         x[k]=j;
14         float current_w = acc_w + x[k] * w[k];           // update weight
15         float current_v = acc_v + x[k] * v[k];           // update value
16         if( current_w <= W &&                               // if is feasible ...
17             current_v + add_rest(v, k+1) > best_v         // ... and is promising
18         )
19             knapsack(v, w, W, k+1, x, current_w, current_v, best_v);
20     }
21 }

```

- **Peor caso** Todos los objetos caben: $O(n2^n)$ (se puede mejorar)
- **Mejor caso** Ningún objeto cabe: $O(n)$

Podas mas ajustadas

- Interesa que los mecanismos de poda “actúen” lo antes posible
 - Una poda mas **ajustada** se puede obtener usando la solución voraz al problema de la **mochila continua**
 - la solución al problema de la mochila continua es siempre **mayor** que la solución al problema de la mochila discreto
- ⇒ El mejor valor que se puede obtener para una solución incompleta:

$$(x_1, x_2, \dots, x_k, ?, \dots, ?)$$

será menor que:

$$\sum_{i=1}^k x_i v_i + \text{knapsack}_c \left(\{x_{k+1}, \dots, x_n\}, W - \sum_{i=1}^k x_i w_i \right)$$

donde $\text{knapsack}_c(X, P)$ es la solución de la mochila continua


```

1 void knapsack(
2     const vector<double> &v, const vector<double> &w,
3     double W, unsigned k, vector<short> &x,
4     double acc_w, double acc_v, double &best_v
5 ){
6     if( k == x.size() ) {                                // if it is a leaf
7         best_v = acc_v;
8         return;
9     }
10                                                                    // it is not a leaf
11     for (unsigned j = 0; j < 2; j++ ) {
12         x[k]=j;
13         double current_w = acc_w + x[k] * w[k];          // update weight
14         double current_v = acc_v + x[k] * v[k];          // update value
15         if( current_w <= W &&                               // if it is promising
16             current_v + knapsack_c( v, w, k+1, W - current_w) > best_v
17         )
18             knapsack( v, w, W, k+1, x, current_w, current_v, best_v);
19     }
20 }
    
```

- **Peor caso** Todos los objetos caben: $O(2^n n \log n)$ (se puede mejorar)
- **Mejor caso** Ningún objeto cabe: $O(n)$

Partiendo de una solución cuasi-óptima

- La efectividad de la poda también puede aumentarse partiendo de una solución factible muy “buena”
- Una posibilidad es usar la solución voraz para la mochila discreta:

Solución óptima partiendo de un subóptimo.

```
1 double best_v = greedy_discrete_knapsack(v,w,W);  
2 knapsack(v, w, W, 0, x, best_v );
```

- También puede ser relevante la forma en la que se “despliega el árbol”:
 - i.e.: completar la tupla primero con los unos y después con los ceros

- 25 muestras aleatorias de tamaño $n = 30$

Tipo de poda	Partiendo de un subóptimo voraz	Llamadas recursivas realizadas (promedio)	Tiempo medio (segundos)
Ninguna	–	1054.8×10^6	875.65
Completando con todos los objetos restantes	No	925.5×10^3	0.112
	Si	389.0×10^3	0.072
Completando según la sol. voraz mochila continua	No	2.3×10^3	0.034
	Si	18	0.002

Encontrar las combinaciones

```
1 void combination(unsigned n){
2     vector<short> x(n);
3
4     int k = 0;
5     x[0] = -1;
6     while( k > -1 ) {
7         while( x[k] < 1 ) {
8             x[k]++;
9             if( k == int(n - 1) ) { // if it is a leaf
10                 cout << x << endl;
11             } else { // if it is not a leaf
12                 k++;
13                 x[k] = -1;
14             }
15         }
16         k--;
17     }
18 }
```

Encontrar las soluciones factibles

```
1 void feasible( const vector<double> &w, double W){
2     vector<short> x(w.size());
3
4     int k = 0;
5     x[0] = -1;
6     while( k > -1 ) {
7         while( x[k] < 1 ) {
8             x[k]++;
9             if( k == int(w.size() - 1) ) {    // if it is a leaf
10                 if( weight(w, x) <= W ) {    // if it is feasible
11                     cout << x << endl;
12                 } else {                      // if it is not a leaf go deeper
13                     k++;
14                     x[k] = -1;
15                 }
16             }
17         }
18         k--;
19     }
20 }
```

Solución iterativa

```
1 double knapsack(const vector<double> &v, const vector<double> &w, double W) {
2     vector<short> x(w.size());
3     double best_v = -1;
4     int k = 0;
5     x[0] = -1;
6     while( k > -1 ){
7         while( x[k] < 1 ){
8             x[k]++;
9             if( weight(w,k+1,x) <= W && value(v,k+1,x) // if it is promising
10                + knapsack_c(v,w,k+1,W-weight(w,k+1,x)) > best_v ){
11                 if( k == int(x.size() - 1) ) { // if it is a leaf
12                     best_v = value(v,x);
13                 } else { // expand
14                     k++;
15                     x[k] = -1;
16                 }
17             }
18             k--;
19         }
20     }
21     return best_v;
22 }
```

1 Ejemplo introductorio: El problema de la mochila (general)

2 Vuelta atrás

3 Ejercicios

- Permutaciones
- El viajante de comercio
- El problema de las n reinas
- La función compuesta mínima

4 Ejercicios propuestos

Vuelta atrás: definición y ámbito de aplicación

- Algunos problemas sólo se pueden resolver mediante el estudio exhaustivo del conjunto de posibles soluciones al problema
- De entre todas ellas, se podrá seleccionar un subconjunto o bien, aquella que consideremos la mejor (la solución óptima)
- *Vuelta atrás* proporciona una forma sistemática de generar todas las posibles soluciones a un problema
- Generalmente se emplea en la resolución de problemas de selección u optimización en los que el conjunto de soluciones posibles es finito
- En los que se pretende encontrar una o varias soluciones que sean:
 - Factibles: que satisfagan unas restricciones y/o
 - Óptimas: optimicen una cierta función objetivo

Vuelta atrás: características I

- La solución debe poder expresarse mediante una tupla de decisiones:
 $(x_1, x_2, \dots, x_n) \quad x_i \in D_i$
 - Las decisiones pueden pertenecer a dominios diferentes entre sí pero estos dominios siempre serán discretos o discretizables
- Es posible que se tenga que explorar todo el espacio de soluciones
 - Los mecanismos de poda van dirigidos a disminuir la probabilidad de que esto ocurra.
- La estrategia puede proporcionar:
 - una solución factible
 - todas las soluciones factibles
 - la solución óptima al problema
 - las n mejores soluciones factibles al problema
- A costa, en la mayoría de los casos, de complejidades prohibitivas

Vuelta atrás: características II

- La generación y búsqueda de la solución se realiza mediante un sistema de prueba y error:
 - Sea $(x_1, x_2, \dots, x_i, \dots)$ una tupla por completar
 - Decidimos sobre la componente x_i :
 - Si la decisión cumple las restricciones se añade x_i a la solución y se avanza a la siguiente componente x_{i+1}
 - Si no cumple las restricciones se prueba otra posibilidad para x_i
 - También se prueba otra posibilidad para x_i cuando regresa de x_{i+1}
 - Si no hay más posibilidades para x_i se retrocede a x_{i-1} para probar otra posibilidad con esa componente (por lo que el proceso comienza de nuevo)
 - Al final, y si ningún mecanismo de poda lo impide, se habrá explorado todo el espacio de soluciones
- Se trata de un recorrido sobre una estructura arbórea imaginaria

Vuelta atrás: Esquema general recursivo

Esquema recursivo de *backtracking* (optimización)

```
1 solution backtracking( problem P ){
2     solution the_best = feasible_solution(n);
3     backtracking( initial_node(P), the_best);
4     return the_best;
5 }
6
7 void backtracking( node n, solution& the_best ){
8
9     if ( is_leaf(n) ) {
10         if( is_best( solution(n), the_best ) )
11             the_best = solution(n);
12         return;
13     }
14
15     for( node a : expand(n) )
16         if( is_feasible(a) && is_promising(a) )
17             backtracking( a, the_best );
18
19     return;
20 }
```

1 Ejemplo introductorio: El problema de la mochila (general)

2 Vuelta atrás

3 Ejercicios

- Permutaciones
- El viajante de comercio
- El problema de las n reinas
- La función compuesta mínima

4 Ejercicios propuestos

Dado un entero positivo n , escribir un algoritmo que muestre todas las permutaciones de la secuencia $(0, \dots, n-1)$

- Solución:

- sea $X = (x_0, x_1, \dots, x_{n-1})$ $x_i \in \{0, 1, \dots, n-1\}$
- cada permutación será cada una de las reordenaciones de X
- restricción: X no puede tener elementos repetidos
- no hay función objetivo: se buscan todas las combinaciones factibles

```
1 bool is_used( vector<short> &x, unsigned k, short e ) {
2     for( unsigned i = 0; i < k; i++ )
3         if( x[i] == e )
4             return true;
5     return false;
6 }
7
8 void permutations( unsigned k, vector<short> &x ) {
9     if( k == x.size() ) {
10         cout << x << endl;
11         return;
12     }
13
14     for( unsigned c = 0; c < x.size(); c++ )
15         if( !is_used( x, k, c ) ) {
16             x[k] = c;
17             permutations( k+1, x );
18         }
19 }
20
21 void permutations( unsigned k ) {
22     vector<short> x(k);
23     permutations( 0, x );
24 }
```

```
1 void permutations( unsigned k, vector<short> &x, vector<bool>& is_used ){
2     if( k == x.size() ) {
3         cout << x << endl;
4         return;
5     }
6
7     for( unsigned c = 0; c < x.size(); c++ ) {
8
9         if( !is_used[c] ) {
10             x[k] = c;
11             is_used[c] = true;
12             permutations( k+1, x, is_used );
13             is_used[c] = false;
14         }
15
16     }
17 }
18
19 void permutations( unsigned k ) {
20     vector<bool> is_used(k, false);
21     vector<short> x(k);
22     permutations( 0, x, is_used );
23 }
```

Restricción (con swap)

```
1 void permutations( unsigned k, vector<short> &x ) {
2
3     if( k == x.size() ) {
4         cout << x << endl;
5         return;
6     }
7
8     for( unsigned c = k; c < x.size(); c++ ) {
9         swap(x[k],x[c]);
10        permutations( k+1, x );
11        swap(x[k],x[c]);
12    }
13 }
14
15 void permutations( unsigned k ) {
16     vector<short> x(k);
17
18     for( unsigned i = 0; i < k; i++ )
19         x[i] = i;
20
21     permutations( 0, x );
22 }
```


El viajante de comercio

Dado un grafo ponderado $g = (V, E)$ con pesos no negativos, el problema consiste en encontrar un *ciclo hamiltoniano* de mínimo coste

- Un *ciclo hamiltoniano* es un recorrido en el grafo que recorre todos los vértices sólo una vez y regresa al de partida
- El coste de un ciclo viene dado por la suma de los pesos de las aristas que lo componen

El viajante de comercio

- Expresamos la solución mediante una tupla $X = (x_1, x_2, \dots, x_n)$ donde $x_i \in \{1, 2, \dots, n\}$ es el vértice visitado en i -ésimo lugar
 - Asumimos que los vértices están numerados,
 $V = \{1, 2, \dots, n\}$, $n = |V|$
 - Fijamos el vértice de partida (para evitar rotaciones):
 - $x_1 = 1$; $x_i \in \{2, 3, \dots, n\} \quad \forall i : 2 \leq i \leq n$
- Restricciones
 - No se puede visitar dos veces el mismo vértice:
 $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
 - Existencia de arista: $\forall i : 1 \leq i < n, \text{weight}(g, x_i, x_{i+1}) \neq \infty$
 - Existencia de arista que cierra el camino: $\text{weight}(g, x_n, x_1) \neq \infty$
- Función objetivo:

$$\min \sum_{i=1}^{n-1} \text{weight}(g, x_i, x_{i+1}) + \text{weight}(g, x_n, x_1)$$

El viajante de comercio

```
1 unsigned round( const graph &g, const vector<short> &x ) {
2     unsigned d = 0;
3     for( unsigned i = 0; i < x.size() - 1; i++ )
4         d += g.dist( x[i], x[i+1] );
5     d += g.dist( x[x.size()-1], x[0] );
6     return d;
7 }
8 void solve(
9     const graph &g, unsigned k, vector<short> &x,
10    vector<bool>& is_used, unsigned &shortest
11 ){
12     if( k == x.size() ) {
13         shortest = min( shortest, round(g,x) );
14         return;
15     }
16     for( unsigned c = 0; c < x.size(); c++ ) {
17         if( !is_used[c] ) {
18             x[k] = c;
19             is_used[c] = true;
20             solve( g, k+1, x, is_used, shortest );
21             is_used[c] = false;
22         }
23     }
24 }
```

El viajante de comercio (2ª parte)

```
1 void solve(const graph &g, unsigned k, vector<short> &x,  
2           vector<bool>& is_used, unsigned &shortest){  
3     if( k == x.size() ) {  
4         shortest = min( shortest, round(g,x));  
5         return;  
6     }  
7     for( unsigned c = 0; c < x.size(); c++ ) {  
8         if( !is_used[c] ) {  
9             x[k] = c;  
10            is_used[c] = true;  
11            solve( g, k+1, x, is_used, shortest );  
12            is_used[c] = false;  
13        }  
14    }  
15 }  
16  
17 unsigned solve( const graph &g ) {  
18     unsigned shortest = numeric_limits<unsigned>::max();  
19     vector<bool> used(g.num_cities(), false);  
20     vector<short> x(g.num_cities());  
21     x[0] = 0; // fix the first city  
22     solve( g, 1, x, used, shortest );  
23     return shortest;  
24 }
```

El viajante de comercio (con swap)

```
1 void solve(  
2     const graph g, unsigned k, vector<short> &x,  
3     unsigned &shortest  
4 ) {  
5     if( k == x.size() ) {  
6         shortest = min( shortest, round(g,x));  
7         return;  
8     }  
9  
10    for( unsigned c = k; c < x.size(); c++ ) {  
11        swap( x[k], x[c] );  
12        solve( g, k+1, x, shortest );  
13        swap( x[k], x[c] );  
14    }  
15 }  
16  
17 unsigned solve( const graph &g ) {  
18     unsigned shortest = numeric_limits<unsigned>::max();  
19     vector<short> x(g.num_cities());  
20     for( unsigned i = 0; i < x.size(); i++ )  
21         x[i] = i;  
22     solve( g, 1, x, shortest ); // Primera ciudad fija  
23     return shortest;  
24 }
```

Ejercicio:

- La solución algorítmica propuesta resulta inviable por su prohibitiva complejidad: $O(n^n)$
- Por ello, y para acelerar la búsqueda, se pide:
 - Aplicar algún mecanismo de poda basado en la mejor solución hasta el momento (por ejemplo, empezar con la solución voraz)
 - Diseñar algún heurístico voraz que permita cumplir el objetivo
 - Sugerencia: Utilizar las ideas de los algoritmos de Prim o de Kruskal (relajación de las restricciones)

Las n mejores soluciones

```
1 void solve_nbest(
2     const graph g,
3     unsigned k,
4     vector<short> &x,
5     priority_queue<unsigned> &pq,
6     unsigned n
7 ){
8     if( k == x.size() ) {
9         unsigned len = round(g,x);
10        if( pq.top() > len ) {
11            if( pq.size() == n )
12                pq.pop();
13            pq.push(len);
14        }
15        return;
16    }
17
18    for( unsigned c = k; c < x.size(); c++ ) {
19        swap(x[k],x[c]);
20        solve_nbest( g, k+1, x, pq, n );
21        swap(x[k],x[c]);
22    }
23 }
```









Las n mejores soluciones

```
1 priority_queue<unsigned> solve_nbest(  
2     const graph &g,  
3     unsigned n  
4 ){  
5     vector<short> x(g.num_cities());  
6     for( unsigned i = 0; i < x.size(); i++ )  
7         x[i] = i;  
8  
9     priority_queue<unsigned> pq;  
10    pq.push(numeric_limits<unsigned>::max());  
11  
12    solve_nbest( g, 1, x, pq, n );  
13  
14    return pq;  
15 }
```


El problema de las n reinas

En un tablero de “ajedrez” de $n \times n$ obtener todas las formas de colocar n reinas de forma que no se ataquen mutuamente (ni en la misma fila, ni columna, ni diagonal).

- Ejemplo:

	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5								
6								
7								
8								

El problema de las n reinas

Solución:

- No puede haber dos reinas en la misma fila,
 - la reina i se colocará en la fila i .
 - El problema es determinar en qué columna se colocará.
- Sea $X = (x_1, x_2, \dots, x_n)$,
 - $x_i \in \{1, 2, \dots, n\}$: columna en la que se coloca la reina de la fila i
- Restricciones:
 - 1 No puede haber dos reinas en la misma fila:
 - implícito en la forma de representar la solución.
 - 2 No puede haber dos reinas en la misma columna
 - X no puede tener elementos repetidos.
 - 3 No puede haber dos reinas en la misma diagonal:
 $\Rightarrow |i - j| \neq |x_i - x_j|$

El problema de las n reinas (todas las soluciones)

```
1 bool feasible( vector<unsigned> &x, unsigned k ) {
2     for( unsigned i = 0; i < k; i++ ) {
3         if( x[i] == x[k] ) return false;
4         if( x[i] < x[k] && x[k] - x[i] == k - i ) return false;
5         if( x[i] > x[k] && x[i] - x[k] == k - i ) return false;
6     }
7     return true;
8 }
9
10 void n_queens( unsigned k, vector<unsigned> &x ) {
11     if( k == x.size() ) {
12         cout << x << endl;
13         return;
14     }
15     for( unsigned i = 0; i < x.size(); i++ ) {
16         x[k] = i;
17         if( factible(x, k) ) n_queens( k+1, x );
18     }
19 }
20
21 void n_queens( unsigned n ) {
22     vector<unsigned> x(n);
23     n_queens(0,x);
24 }
```

El problema de las n reinas (sólo una solución)

```
1 void n_queens( unsigned k, vector<unsigned> &x, bool &found ) {
2
3     if( k == x.size() ) {
4         cout << x << endl;
5         found = true;
6         return;
7     }
8
9     for( unsigned i = 0; i < x.size(); i++ ) {
10         x[k] = i;
11         if( factible(x, k) )
12             n_queens( k+1, x, found );
13         if( found ) break;
14     }
15 }
16
17
18 void n_queens( unsigned n ) {
19     vector<unsigned> x(n);
20     bool found = false;
21
22     n_queens( 0, x, found );
23 }
```

El problema de las n reinas (sólo una solución)

```
1 void n_queens(  
2     unsigned k, vector<unsigned> &x, vector<unsigned> &sol, bool &found  
3 ) {  
4     if( k == x.size() ) {  
5         sol = x;  
6         found = true;  
7         return;  
8     }  
9     for( unsigned i = 0; i < x.size(); i++ ) {  
10         x[k] = i;  
11         if( factible(x, k) )  
12             n_queens( k+1, x, sol, found );  
13         if( found )  
14             break;  
15     }  
16 }  
17  
18 vector<unsigned> n_queens( unsigned n ) {  
19     vector<unsigned> x(n);  
20     vector<unsigned> sol(n);  
21     bool found = false;  
22     n_queens( 0, x, sol, found );  
23     return sol;  
24 };
```

La función compuesta mínima

Dadas dos funciones $f(x)$ y $g(x)$ y dados dos números cualesquiera x e y , encontrar la función compuesta mínima que obtiene el valor y a partir de x tras aplicaciones sucesivas e indistintas de $f(x)$ y $g(x)$

- Ejemplo: Sean $f(x) = 3x$, $g(x) = \lfloor x/2 \rfloor$, y sean $x = 3$, $y = 6$
 - Una transformación de 3 en 6 con operaciones f y g es:

$$(g \circ f \circ g \circ f \circ f \circ g)(3) = 6 \quad (5 \text{ composiciones})$$

- La mínima (aunque no única) es:

$$(f \circ g \circ g \circ f)(3) = 6 \quad (3 \text{ composiciones})$$

La función compuesta mínima

Solución:

- $X = (x_1, x_2, \dots, x_k)$ $x_i \in \{0, 1\}$ $\begin{cases} 0 \equiv \text{se aplica } f(x) \\ 1 \equiv \text{se aplica } g(x) \end{cases}$
 - $(x_1, x_2, \dots, x_k) \equiv (x_k \circ \dots \circ x_2 \circ x_1)$
 - El tamaño de la tupla no se conoce a priori
 - Se pretende minimizar el tamaño de la tupla solución (función objetivo)
 - asumiremos un máximo de M composiciones (evitar ramas infinitas)
- Llamamos $F(X, k, x)$ al resultado de aplicar al valor x la composición representada en la tupla X hasta su posición k

$$F(X, k, x) = (x_k \circ \dots \circ x_2 \circ x_1)(x) = x_k(\dots x_2(x_1(x)) \dots)$$

- Restricciones:
 - $F(X, k, x) \neq F(X, i, x) \ \forall i < k$, para evitar recálculos
 - $k < M$, para evitar búsquedas infinitas
 - siempre se puede calcular $F(X, k, x)$
 - $k < v_b$, (v_b es la mejor solución actual) tupla “prometedora”

La función compuesta mínima (sin poda)

```
1 int F( const vector<short> &x, unsigned k, int init) {
2     for( unsigned i = 0; i < k; i++ )
3         if( x[i] == 0 ) init = f(init);
4         else      init = g(init);
5     return init;
6 }
7 void composition(
8     unsigned k, vector<short> &x, unsigned M,
9     int first, int last, unsigned &best
10 ) {
11     if( F(x, k, first) == last && k < best )
12         best = k;                                // all the nodes are a feasible solution
13     if( k == M ) return;                          // base case
14     for( short i = 0; i < 2; i++ ) {
15         x[k] = i;
16         composition(k+1, x, M, first, last, best);
17     }
18 }
19 int composition(unsigned M, int first, int last) {
20     vector<short> x(M);
21     unsigned best = numeric_limits<unsigned>::max(); // unsigned max
22     composition( 0, x, M, first, last, best);
23     return best;
24 }
```


La función compuesta mínima (incremental)

```
1 int F1( unsigned i, int init) {
2     if( i == 0 )
3         return f(init);
4     else
5         return g(init);
6 }
7
8 void composition(
9     unsigned k, unsigned M, int first, int last, unsigned &best
10 ) {
11     if( first == last && k < best ) best = k;
12     if( k == M ) return;
13     for( short i = 0; i < 2; i++ ) {
14         int next = F1( i, first );
15         composition( k+1, M, next, last, best);
16     }
17 }
18
19 int composition( unsigned M, int first, int last ) {
20     unsigned best = numeric_limits<unsigned>::max(); // unsigned max
21     composition( 0, M, first, last, best );
22     return best;
23 }
```

La función compuesta mínima (incremental con poda)

```
1 void composition( unsigned k, unsigned M, int first, int last,
2   unsigned &best
3 ) {
4   if( k >= best ) return;
5
6   if( first == last ) { // k < best
7     best = k;
8     return;
9   }
10  if( k == M ) return;
11
12  for( short i = 0; i < 2; i++ ) {
13    int next = F1(i,first);
14    composition(k+1, M, next, last, best);
15  }
16 }
17
18 int composition(unsigned M, int first, int last) {
19   unsigned best = numeric_limits<unsigned>::max();
20   composition( 0, M, first, last, best);
21   return best;
22 }
```

La función compuesta mínima (con poda y memoria)

```
1 void composition(  
2     unsigned k, unsigned M, int first, int last,  
3     unordered_map<int, Default<unsigned, u_max>> &best  
4 ) {  
5     if( k > best[last] ) return;    // if new solutions can't be better ... prune  
6  
7     if( best[first] <= k ) // if current node is reachable in fewer steps prune  
8         return;           // what happens if best[first] does not exists?  
9     best[first] = k;  
10  
11     if( k == M ) return;  
12  
13     for( short i = 0; i < 2; i++ ) {  
14         int next = F1(i,first);  
15         composition(k+1, M, next, last, best);  
16     }  
17 }  
18  
19 int composition(unsigned M, int first, int last) {  
20     unordered_map<int, Default<unsigned, u_max>> best;  
21     composition( 0, M, first, last, best);  
22  
23     return best[last];  
24 }
```

Cambio del valor por defecto del operador []

```
1 template<typename T, T X>
2 class Default {
3 private:
4     T val;
5 public:
6     Default () : val(T(X)) {}
7     Default (T const & _val) : val(_val) {}
8     operator T & () { return val; }
9     operator T const & () const { return val; }
10 };
11
12 unsigned const u_max = numeric_limits<unsigned>::max();
```

Potencia de las podas

Número de llamadas recursivas para encontrar el mínimo número de composiciones (12) para llegar de 1 a 11

Tipo poda	$M = 15$	$M = 20$	$M = 25$
Básico	65 535	2 097 151	67 108 863
mejor en curso	9 075	40 323	1 040 259
memorización	1 055	7 243	50 669
las dos	565	2 541	16 469

Sería mejor hacer una búsqueda por niveles...

1 Ejemplo introductorio: El problema de la mochila (general)

2 Vuelta atrás

3 Ejercicios

- Permutaciones
- El viajante de comercio
- El problema de las n reinas
- La función compuesta mínima

4 Ejercicios propuestos

El coloreado de grafos

Dado un grafo G , encontrar el menor número de colores con el que se pueden colorear sus vértices de forma que no haya dos vértices adyacentes con el mismo color

El recorrido del caballo de ajedrez

Encontrar una secuencia de movimientos “legales” de un caballo de ajedrez de forma que éste pueda visitar las 64 casillas de un tablero sin repetir ninguna

El laberinto con cuatro movimientos

- Se dispone de una cuadrícula $n \times m$ de valores $\{0, 1\}$ que representa un laberinto. Un valor 0 en una casilla cualquiera de la cuadrícula indica una posición inaccesible; por el contrario, con el valor 1 se simbolizan las casillas accesibles.
- Encontrar un camino que permita ir de la posición $(1, 1)$ a la posición (n, m) con cuatro tipos de movimiento (arriba, abajo, derecha, izquierda)

La asignación de tareas

- Supongamos que disponemos de n trabajadores y n tareas. Sea $b_{ij} > 0$ el coste de asignarle el trabajo j al trabajador i .
- Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables x_{ij} , donde $x_{ij} = 0$ significa que al trabajador i no le han asignado la tarea j , y $x_{ij} = 1$ indica que sí
- Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador
- Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^n \sum_{j=1}^n x_{ij} b_{ij}$$

- Encontrar una asignación óptima, es decir, de mínimo coste

- Supongamos una empresa naviera que dispone de una flota de N buques cada uno de los cuales transporta mercancías de un valor v_i que tardan en descargarse un tiempo t_i . Solo hay un muelle de descarga y su máximo tiempo de utilización es T
- Diseñar un algoritmo que determine el orden de descarga de los buques de forma que el valor descargado sea máximo sin sobrepasar el tiempo de descarga T . (Si se elige un buque para descargarlo, es necesario que se descargue en su totalidad)

La asignación de turnos

- Estamos al comienzo del curso y los alumnos deben distribuirse en turnos de prácticas
- Para solucionar este problema se propone que valoren los turnos de práctica disponibles a los que desean ir en función de sus preferencias
- El número de alumnos es N y el de turnos disponibles es T
- Se dispone una matriz de preferencias P , $N \times T$, en la que cada alumno escribe, en su fila correspondiente, un número entero (entre 0 y T) que indica la preferencia del alumno por cada turno (0 indica la imposibilidad de asistir a ese turno; M indica máxima preferencia)
- Se dispone también de un vector C con T elementos que contiene la capacidad máxima de alumnos en cada turno
- Se pretende encontrar una solución para satisfacer el número máximo de alumnos según su orden de preferencia sin exceder la capacidad de los turnos

- El famoso juego del **Sudoku** consiste en rellenar una rejilla de 9×9 celdas dispuestas en 9 subgrupos de 3×3 celdas, con números del 1 al 9, atendiendo a la restricción de que no se debe repetir el mismo número en la misma fila, columna o subgrupo 3×3
- Además, varias celdas disponen de un valor inicial, de modo que debemos empezar a resolver el problema a partir de esta solución parcial sin modificar ninguna de las celdas iniciales

Análisis y diseño de algoritmos

7. Ramificación y Poda

José Luis Verdú Mas, Jose Oncina,
Mikel L. Forcada

Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

1 de mayo de 2020

- 1 Ejemplo introductorio
- 2 Esquema de ramificación y poda
- 3 Ejemplos resueltos
 - El viajante de comercio
 - La función compuesta mínima
- 4 Ejercicios

El problema de la mochila (general)

Dados:

- n objetos con valores v_i y pesos w_i
- una mochila que solo aguanta un peso máximo W

Seleccionar un conjunto de objetos de forma que:

- no se sobrepase el peso límite W (restricción)
- el valor transportado sea máximo (función objetivo)

- Solución: $X = (x_1, x_2, \dots, x_n)$ $x_i \in \{0, 1\}$
- Restricciones:

- Implícitas:

$$x_i \in \begin{cases} 0 & \text{no se selecciona el objeto } i \\ 1 & \text{se selecciona el objeto } i \end{cases}$$

- Explícitas:

$$\sum_{i=1}^n x_i w_i \leq W$$

- Función objetivo:

$$\text{máx} \sum_{i=1}^n x_i v_i$$

Tipos de soluciones

- Supongamos el siguiente ejemplo:

$$W = 16$$

$$w = (2, 8, 7)$$

$$v = (20, 40, 49)$$

- Combinaciones posibles (espacio de soluciones):

Solución **Peso** **Valor**

(0, 0, 0)	0	0
(0, 0, 1)	7	49
(0, 1, 0)	8	40
(0, 1, 1)	15	89
(1, 0, 0)	2	20
(1, 0, 1)	9	69
(1, 1, 0)	10	60
(1, 1, 1)	17	109

Soluciones factibles

Solucion óptima

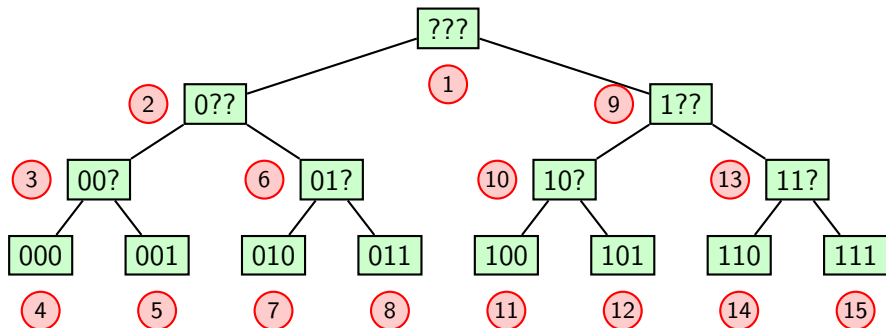
Solución voraz

Solución NO factible

Solución usando vuelta atrás

- Combinaciones posibles:

- Supongamos el ejemplo: $W = 16$ $w = (7, 8, 2)$ $v = (49, 40, 20)$
- Espacio de soluciones
 - Generación ordenada mediante vuelta atrás
 - Nodos generados: 15
 - Nodos expandidos: 7



¿Es el recorrido importante?

- ¿Podríamos llegar antes a la solución óptima con otro recorrido?
- ¿Cómo?
 - Adecuando el **orden de exploración** del árbol de soluciones según nuestros intereses. Se priorizará para su exploración aquellos nodo mas prometedores
- ... sin olvidar las podas

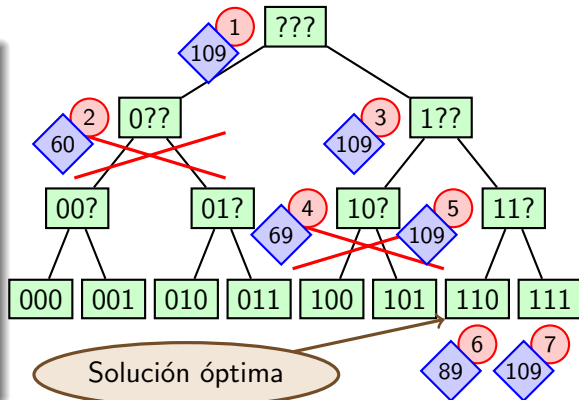
Ejemplo Introdutorio

- Combinaciones posibles:

- Supongamos el ejemplo: $W = 16$ $w = (7, 8, 2)$ $v = (49, 40, 20)$

Función de cota

Cada nodo toma cota el valor que resultaría de incluir en la solución aquellos objetos pendientes de tratar (sustituir en cada nodo los '?' por '1') independientemente de que quepan o no.



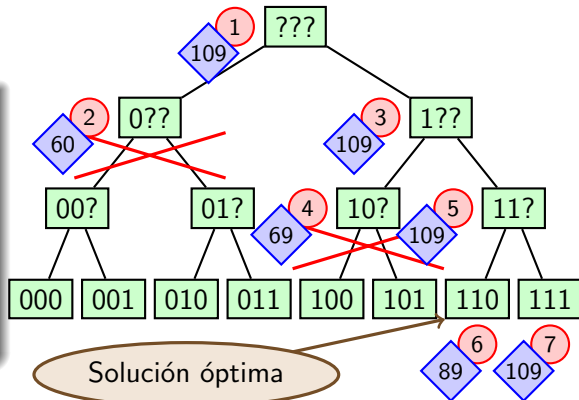
Ejemplo Introdutorio

- Combinaciones posibles:

- Supongamos el ejemplo: $W = 16$ $w = (7, 8, 2)$ $v = (49, 40, 20)$

Espacio de soluciones

- Orden de expansión priorizando los nodos con mayor cota
- Generados: 7
- Expandidos: 3
- Reducción $\geq 50\%$



Implementación

```
1 float knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2
3     typedef vector<short> sol;
4     typedef tuple<double, double, sol, int > node; // value, weight, vector, k
5     priority_queue< node > pq;
6
7     double best_val = knapsack_d( v, w, 0, W);
8     pq.push( node(0.0, 0.0, sol(v.size()), 0 ) );
9
10    while( !pq.empty() ) {
11        double value, weight;
12        sol x;
13        unsigned k;
14
15        tie(value, weight, x, k) = pq.top();
16        pq.pop();
17        /* ... next slide ...*/
18    }
19    return best_val;
20 }
```

Dentro del bucle ...

```
1  /* ... */
2  if( k == v.size() ) {                                // base case
3      best_val = max( value, best_val ) ;
4      continue;
5  }
6
7  for (unsigned j = 0; j < 2; j++ ) {                  // no base case
8      x[k] = j;
9
10     double new_weight = weight + x[k] * w[k]; // updating weight
11     double new_value = value + x[k] * v[k];    // updating value
12
13     if( new_weight <= W &&                          // is feasible & is promising
14         new_value + knapsack_c( v, w, k + 1, W - new_weight ) > best_val
15     )
16         pq.push( node( new_value, new_weight, x, k + 1 ) );
17 }
18 /* ... */
```


Usando podas pesimistas

```
1  /* ... */
2  if( k == v.size() ) {                                // base case
3      best_val = max( value, best_val ) ;
4      continue;
5  }
6
7  for (unsigned j = 0; j < 2; j++ ) {
8      x[k] = j;
9
10     double new_weight = weight + x[k] * w[k];          // updating weight
11     double new_value = value + x[k] * v[k];            // updating value
12
13     if( new_weight <= W ) {                             // is feasible
14                                     // updating pessimistic bound
15         best_val = max( best_val, new_value
16             + knapsack_d( v, w, k+1, W - new_weight )
17         );
18
19         double opt_bound = new_value + knapsack_c(v, w, k+1, W-new_weight);
20         if( opt_bound > best_val )                       // is promising
21             pq.push( node( new_value, new_weight, x, k+1 ) );
22     }
23 }
24 /* ... */
```

Ordenando por cota optimista

```
1 double knapsack( const vector<double> &v, const vector<double> &w, double W ) {
2     typedef vector<short> sol;
3     //          opt_bound, value, weight, x, k
4     typedef tuple< double, double, double, sol, unsigned > node;
5     priority_queue< node > pq;
6
7     double best_val = knapsack_d( v, w, 0, W);
8     double opt_bound = knapsack_c(v, w, 0, W);
9
10    pq.push( node( opt_bound, 0.0, 0.0, sol(v.size()), 0 ) );
11
12    while( !pq.empty() ) {
13        double value, weight;
14        sol x;
15        unsigned k;
16
17        tie(ignore, value, weight, x, k) = pq.top();
18        pq.pop();
19        /* ... Next slide ... */
20    }
21    return best_val;
22 }
```

Dentro del bucle

```
1  /* ... */
2  if( k == v.size() ) { // base case
3      best_val = max( best_val, value ) ;
4      continue;
5  }
6
7  for (unsigned j = 0; j < 2; j++ ) { // non base
8      x[k] = j;
9
10     float new_weight = weight + x[k] * w[k]; // updating weight
11     float new_value = value + x[k] * v[k]; // updating value
12
13     if( new_weight <= W ) {
14         best_val = max( best_val, new_value
15             + knapsack_d( v, w, k+1, W - new_weight));
16         float opt_bound = new_value
17             + knapsack_c( v, w, k+1, W - new_weight);
18         if( opt_bound > best_val ) // is promising
19             pq.push( node( opt_bound, new_value, new_weight, x, k+1 ) );
20     }
21 }
22 /* ... */
```

Promedio del número de iteraciones para 100 instancias aleatorias del problema de la mochila con 100 objetos.

	optimista	inicializando	pesimista
Vuelta Atrás	4 491	277	253
RyP (por valor)	2 406	229	197
RyP (por cota optimista)	206	122	112

- 1 Ejemplo introductorio
- 2 Esquema de ramificación y poda
- 3 Ejemplos resueltos
 - El viajante de comercio
 - La función compuesta mínima
- 4 Ejercicios

Definición y ámbito de aplicación

- Variante del diseño de Vuelta Atrás
 - Realiza una enumeración parcial del espacio de soluciones mediante la generación de un árbol de expansión
 - Uso de **cotas para podar** ramas que no son de interés
- **Nodo vivo**: aquel con posibilidades de ser ramificado (visitado pero no completamente expandido)
- Los nodos vivos se almacenan en estructuras que faciliten su **recorrido** y eficiencia de la búsqueda:
 - En profundidad (estrategia LIFO) \Rightarrow pila
 - En anchura (estrategia FIFO) \Rightarrow cola
 - Dirigida (primero el mas prometedor) \Rightarrow cola de prioridad

Definición y ámbito de aplicación

- Funcionamiento de un algoritmo de ramificación y poda
- 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 conjuntos de nodos vivos

- **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:**

- 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

Esquema de Ramificación y Poda

```
1 solution branch_and_bound( problem p ) {
2
3     node initial = initial_node(p);                // supposed feasible
4     solution current_best = pessimistic_solution(initial); // pessimistic
5     priority_queue<Node> q.push(initial);
6
7     while( ! q.empty() ) {
8         node n = q.top();
9         q.pop();
10
11         if( is_leaf(n) ) {
12             if( is_better( solution(n), current_best) )
13                 current_best = solution(n);
14             continue;
15         }
16
17         for( node a : expand(n) )
18             if( is_feasible(a) && is_promising( a, current_best))
19                 q.push(a);
20     }
21
22     return current_best;
23 }
```

- Funciones:

- `initial_node(p)`: obtiene el nodo inicial para la expansión
- `pesimistic_solution(n)`: devuelve una solución aproximada (factible pero no la óptima)
- `is_leaf(n)`: mira si `n` es una posible solución
- `solution(n)` devuelve el valor del nodo `n`
- `expand(n)`: devuelve la expansión de `n`
- `is_feasible(n)`: comprueba si `n` cumple las restricciones
- `is_promising(n, current_best)`: mira si a partir del nodo `n` se pueden obtener soluciones mejores que `current_best` (normalmente se obtiene mediante una solución optimista)

Podando con cotas pesimistas

```
1 solution branch_and_bound( problem p ) {  
2     node initial = initial_node(p);                                // supposed feasible  
3     solution current_best = pessimistic_solution(initial);        // pessimistic  
4     priority_queue<Node> q.push(initial);  
5  
6     while( ! q.empty() ) {  
7         node n = q.top();  
8         q.pop();  
9  
10        if( is_leaf(n) ) {  
11            if( is_better( solution(n), current_best) )  
12                current_best = solution(n);  
13            continue;  
14        }  
15        for( node a : expand(n) )  
16            if( is_feasible(a) ) {  
17                if( is_better( pessimistic_solution(n), current_best ) )  
18                    current_best = pessimistic_solution(n);  
19                if( is_promising (a, current_best ))  
20                    q.push(a);  
21            }  
22        }  
23        return current_best;  
24    }
```

- 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

Esquema de Ramificación y Poda

- Para saber si un nodo es prometedor, se compara la cota optimista del nodo con la mejor solución hasta el momento (mejor cota pesimista encontrada hasta el momento)

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

- Para acelerar la búsqueda se pueden hacer **podas agresivas** cambiando lo anterior por:

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

Pero **¡cuidado!** puede que se pierda la solución óptima.

- 1 Ejemplo introductorio
- 2 Esquema de ramificación y poda
- 3 Ejemplos resueltos**
 - El viajante de comercio
 - La función compuesta mínima
- 4 Ejercicios

El viajante de comercio

Dado un grafo ponderado $g = (V, A)$ con pesos no negativos, el problema consiste en encontrar un *ciclo hamiltoniano* de mínimo coste.

- Un *ciclo hamiltoniano* es un recorrido en el grafo que recorre todos los vértices sólo una vez y regresa al de partida.
- El coste de un ciclo viene dado por la suma de los pesos de las aristas que lo componen.

El viajante de comercio

- Expresamos la solución mediante una tupla $X = (x_1, x_2, \dots, x_n)$ donde $x_i \in \{1, 2, \dots, n\}$ es el vértice visitado en i -ésimo lugar.
 - Asumimos que los vértices están numerados,
 $V = \{1, 2, \dots, n\}$, $n = |V|$
 - Fijamos el vértice de partida (para evitar rotaciones):
 - $x_1 = 1$; $x_i \in \{2, 3, \dots, n\} \quad \forall i : 2 \leq i \leq n$
- Restricciones
 - No se puede visitar dos veces el mismo vértice:
 $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
 - Existencia de arista: $\forall i : 1 \leq i < n, \text{ peso}(g, x_i, x_{i+1}) \neq \infty$
 - Existencia de arista que cierra el camino: $\text{peso}(g, x_n, x_1) \neq \infty$
- Función objetivo:

$$\min \sum_{i=1}^{n-1} \text{peso}(g, x_i, x_{i+1}) + \text{peso}(g, x_n, x_1)$$

El viajante de comercio

```
1 unsigned travelling_salesman( const graph &g) {
2     struct node {
3         unsigned opt_bound, length;
4         vector<short> x;
5         unsigned k;
6     };
7     struct is_worse {
8         bool operator() (const Node& a, const Node& b) {
9             return a.opt_bound > b.opt_bound;
10        }
11    };
12    priority_queue< node, vector<node>, is_worse > pq;
13    vector<short> x(g.num_cities());
14    for( unsigned i = 0; i < g.num_cities(); i++ ) x[i] = i;
15    unsigned shortest = pessimistic_bound( g, x, 1);
16    unsigned opt_bound = optimistic_bound( g, x, 1);
17    pq.push( { opt_bound, 0, x, 1 } );
18    while( !pq.empty() ) {
19        node n = pq.top();
20        pq.pop();
21        /* ... Next slide ... */
22    }
23    return shortest;
24 }
```

Dentro del bucle ...

```
1  /* ... */
2  if( n.k == g.num_cities() ) {
3      shortest = min( shortest, n.length + g.dist(n.x[n.k-1],n.x[0]) );
4      continue;
5  }
6  for( unsigned c = n.k; c < n.x.size(); c++ ) {
7      swap( n.x[n.k], n.x[c] );
8
9      unsigned new_length = n.length + g.dist(n.x[n.k-1],n.x[n.k]);
10     unsigned opt_bound = new_length + optimistic_bound(g,n.x,n.k+1);
11     unsigned pes_bound = new_length + pessimistic_bound(g,n.x,n.k+1);
12
13     shortest = min( shortest, pes_bound);
14
15     if( opt_bound <= shortest )
16         pq.push( { opt_bound, new_length, n.x, n.k+1 } );
17
18     swap( n.x[n.k], n.x[c] );
19 }
20 /* ... */
```

Cambiando la prioridad ...

Por cota optimista

```
1 struct is_worse {  
2     bool operator() (const node& a, const node& b) {  
3         return a.opt_bound > b.opt_bound;  
4     }  
5 };
```

Por distancia recorrida

```
1 struct is_worse {  
2     bool operator() (const node& a, const node& b) {  
3         return a.length > b.length;  
4     }  
5 };
```

Por distancia media recorrida

```
1 struct is_worse {  
2     bool operator() (const node& a, const node& b) {  
3         return a.length/a.k > b.length/b.k;  
4     }  
5 };
```

Algunos ejemplos de ejecución

Promedio del número de iteraciones necesitado para resolver 100 instancias del problema del viajante de comeecio con 15 ciudades

- **Cota optimista:** mininum spanning tree de las ciudades restantes
- **Cota pesimista:** algoritmo voraz basado en la ciudad más cercana

Algoritmo	cota opt.	dist. recorrida	dist. media
Vuelta atrás	23 478	23 478	23 478
Ramificación y poda	10 798	12 285	11 421
factor de aceleración	2.17	1.91	2.05

La función compuesta mínima

Dadas dos funciones $f(x)$ y $g(x)$ y dados dos números cualesquiera x e y , encontrar la función compuesta mínima que obtiene el valor y a partir de x tras aplicaciones sucesivas e indistintas de $f(x)$ y $g(x)$

- Ejemplo: Sean $f(x) = 3x$, $g(x) = \lfloor x/2 \rfloor$, y sean $x = 3$, $y = 6$
 - Una transformación de 3 en 6 con operaciones f y g es:

$$(g \circ f \circ g \circ f \circ f \circ g)(3) = 6 \quad (5 \text{ composiciones})$$

- La mínima (aunque no única) es:

$$(f \circ g \circ g \circ f)(3) = 6 \quad (3 \text{ composiciones})$$

La función compuesta mínima

Solución:

- $X = (x_1, x_2, \dots, x_k)$ $x_i \in \{0, 1\}$ $\begin{cases} 0 \equiv \text{se aplica } f(x) \\ 1 \equiv \text{se aplica } g(x) \end{cases}$
 - $(x_1, x_2, \dots, x_k) \equiv (x_k \circ \dots \circ x_2 \circ x_1)$
 - El tamaño de la tupla no se conoce a priori
 - Se pretende minimizar el tamaño de la tupla solución (función objetivo)
 - asumiremos un máximo de M composiciones (evitar ramas infinitas)
- Llamamos $F(X, k, x)$ al resultado de aplicar al valor x la composición representada en la tupla X hasta su posición k

$$F(X, k, x) = (x_k \circ \dots \circ x_2 \circ x_1)(x) = x_k(\dots x_2(x_1(x)) \dots)$$

- Restricciones:
 - $F(X, k, x) \neq F(X, i, x) \ \forall i < k$, para recálculos
 - $k < M$, para evitar búsquedas infinitas
 - siempre se puede calcular $F(X, k, x)$
 - $k < v_b$, tupla “prometedora”

Composición de funciones

```
1 int composition( unsigned M, int first, int last ) {
2
3     typedef tuple< short, int> node; // steps, hit
4
5     struct is_worse {
6         bool operator() (const node& a, const node& b) {
7             return get<0>(a) > get<0>(b);
8         }
9     };
10    priority_queue< node, vector<node>, is_worse > pq;
11
12    unsigned best = numeric_limits<unsigned>::max();
13    pq.push( node( 0, first) );
14
15    while( !pq.empty() ) {
16        node n = pq.top();
17        pq.pop();
18        /* ... next slide ... */
19    }
20    return best;
21 }
```

Dentro del bucle ... (sin podas)

```
1  ...
2  unsigned k = get<0>(n);
3  int hit = get<1>(n);
4
5  if( hit == last && k < best )
6      best = k;
7
8  if( k == M )
9      continue;
10
11  for( short i = 0; i < 2; i++ ) {
12      int next = F1(i,hit);
13      pq.push( node( k+1, next ) );
14  }
15  ...
```


Dentro del bucle ... (poda: mejor en curso)

poda: mejor en curso

```
1 ...
2     unsigned k = get<0>(n);
3     int hit = get<1>(n);
4
5     if( k >= best )
6         continue;
7
8     if( hit == last && k < best ) {
9         best = k;
10        continue;
11    }
12
13    if( k == M )
14        continue;
15
16    for( short i = 0; i < 2; i++ ) {
17        int next = F1(i,hit);
18        pq.push( node( k+1, next ) );
19    }
20 ...
21 _
```

Podando con memoria

```
1 int composition( unsigned M, int first, int last ) {
2
3     typedef tuple< short, int> node; // steps, hit
4
5     struct is_worse {
6         bool operator() (const node& a, const node& b) {
7             return get<0>(a) > get<0>(b);
8         }
9     };
10
11     priority_queue< node, vector<node>, is_worse > pq;
12
13     unordered_map<int, Default<unsigned, u_max>> best;
14
15     pq.push( node( 0, first) );
16     while( !pq.empty() ) {
17         node n = pq.top();
18         pq.pop();
19         /* ... next slide ... */
20     }
21     return best[last];
22 }
```

Dentro del bucle ...

```
1  /* ... */
2  unsigned k = get<0>(n);
3  int hit = get<1>(n);
4
5  if( k >= best[last] )
6      continue;
7
8  if( best[hit] <= k )
9      continue;
10 best[hit] = k;
11
12 if( k == M )
13     continue;
14
15 for( short i = 0; i < 2; i++ ) {
16     int next = F1(i,hit);
17     pq.push( Node( k+1, next ) );
18 }
19 /* ... */
20
```

Ejemplo de ejecución

Número de iteraciones para alcanzar el valor 11 desde 1 (con $M = 20$)

Algoritmo	Vuelta Atrás	Ramificación y Poda
básico	65 535	65 535
mejor en curso	9 073	8 311
memorizando	565	329

- 1 Ejemplo introductorio
- 2 Esquema de ramificación y poda
- 3 Ejemplos resueltos
 - El viajante de comercio
 - La función compuesta mínima
- 4 Ejercicios

El Puzzle

- Disponemos de un tablero con n^2 casillas y de $n^2 - 1$ piezas numeradas del uno al $n^2 - 1$. Dada una ordenación inicial de todas las piezas en el tablero, queda sólo una casilla vacía (con valor 0), a la que denominamos *hueco*.
- El objetivo del juego es transformar dicha disposición inicial de las piezas en una disposición final ordenada, en donde en la casilla (i, j) se encuentra la pieza numerada $n(i - 1) + j$ y en la casilla (n, n) se encuentra el hueco
- Los únicos movimientos permitidos son los de las piezas adyacentes al hueco (horizontal y verticalmente), que pueden ocuparlo. Al hacerlo, dejan el hueco en la posición en donde se encontraba la pieza antes del movimiento. Otra forma de abordar el problema es considerar que lo que se mueve es el hueco, pudiendo hacerlo hacia arriba, abajo, izquierda o derecha. Al moverse, su casilla es ocupada por la pieza que ocupaba la casilla a donde se ha *movido* el hueco

El Puzzle

- Por ejemplo, para el caso $n = 3$ se muestra a continuación una disposición inicial junto con la disposición final:

1	5	2
4	3	
7	8	6

Disposición Inicial

1	2	3
4	5	6
7	8	

Disposición final

- Regla: una pieza se puede mover de A a B si:
 - A está al lado de B
 - B es el hueco
- Según se relaje el problema tenemos dos funciones de coste diferentes:
 - 1 Calcular el número de piezas que están en una posición distinta de la que les corresponde en la disposición final
 - 2 Calcular la suma de las distancias de Manhattan desde la posición de cada pieza a su posición en la disposición final
- La distancia de Manhattan entre dos puntos del plano de coordenadas (x_1, y_1) y (x_2, y_2) viene dada por la expresión:

$$|x_1 - x_2| + |y_1 - y_2|$$