

# 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

30-01-2018 (389)

- 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

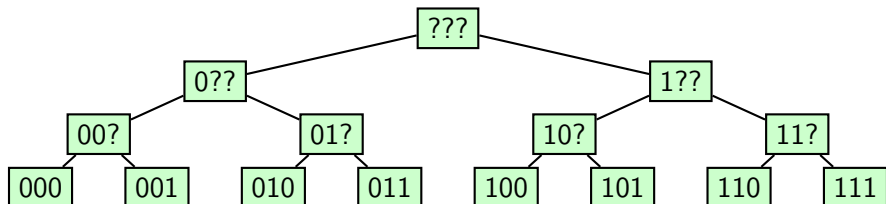
Soluciones factibles

Solución óptima

Solución voraz

Solución NO factible

# Generación de todas las combinaciones



Recorrer todas las combinaciones.

Llamada inicial: combinaciones(0,x)

```
1 void combinations(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         combinations(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

```

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 present_w = acc_w + x[k] * w[k];    // update weight
13        if ( present_w <= W )                      // if it is feasible
14            feasible(w, W, k+1, x, present_w);      // Expand
15    }
16 }
    
```

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

- 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 present_w = acc_w + x[k] * w[k];          // update weight
14         if ( present_w <= W )                             // it is feasible
15             knapsack(v, w, W, k+1, x, present_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 present_w = acc_w + x[k] * w[k]; // update weight  
14         double present_v = acc_v + x[k] * v[k]; // update value  
15         if ( present_w <= W ) // if it is feasible  
16             knapsack( v, w, W, k+1, x, present_w, present_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 present_w = acc_w + x[k] * w[k];           // update weight
15         float present_v = acc_v + x[k] * v[k];           // update value
16         if( present_w <= W &&                               // if is feasible ...
17             present_v + add_rest(v, k+1) > best_v         // ... and is promising
18         )
19             knapsack(v, w, W, k+1, x, present_w, present_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 present_w = acc_w + x[k] * w[k];          // update weight
14         double present_v = acc_v + x[k] * v[k];          // update value
15         if( present_w <= W &&                               // if it is promising
16             present_v + knapsack_c( v, w, k+1, W - present_w) > best_v
17         )
18             knapsack( v, w, W, k+1, x, present_w, present_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	–	$1054.8 \times 10^6$	875.65
Completando con todos los objetos restantes	No	$925.5 \times 10^3$	0.112
	Si	$389.0 \times 10^3$	0.072
Completando según la sol. voraz mochila continua	No	$2.3 \times 10^3$	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();
```



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  $9 \times 9$  celdas dispuestas en 9 subgrupos de  $3 \times 3$  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  $3 \times 3$
- 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