

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

26-01-2015 (318)

El problema de la mochila (general)

Dados:

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

Seleccionar un conjunto de objetos de forma que:

- no se sobrepase el peso límite P (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?**

Ejemplo Introdutorio

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:

$$\sum_{i=1}^n x_i p_i \leq P$$

- Función objetivo:

$$\text{máx} \sum_{i=1}^n x_i v_i$$

Ejemplo Introdutorio

- Supongamos el siguiente ejemplo:

$$P = 16$$

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

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

- Combinaciones posibles (espacio de soluciones):

Solución **Peso** **Valor**

(0, 0, 0) 0 0

(0, 0, 1) 2 20

(0, 1, 0) 8 40

(0, 1, 1) 10 60

(1, 0, 0) 7 49

(1, 0, 1) 8 69

(1, 1, 0) 15 89

(1, 1, 1) 17 109

Soluciones factibles

Solución voraz

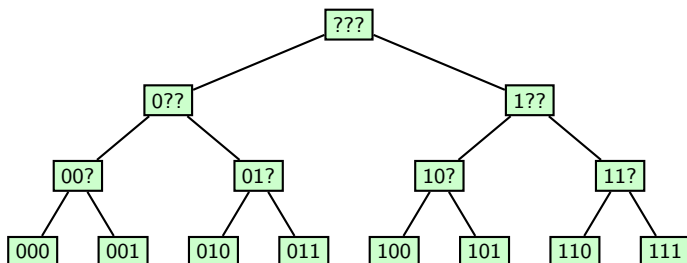
Solución óptima

Solución NO factible

Ejemplo Introdutorio

- Generación de todas las combinaciones

Ejemplo $n = 3$: $X = (x_1, x_2, x_3)$ $x_i \in \{0, 1\}$



Todas las combinaciones.

Llamada inicial: combinaciones(0,x)

```
1 void combinaciones(unsigned k, vector <short> &x){  
2     for (unsigned j=0; j<2; j++) {  
3         x[k]=j; //Nueva alternativa para esta componente de X  
4         if (k==x.size()-1) imprimir_hoja_del_arbol(x);  
5         else combinaciones(k+1, x); //No es hoja: descender en el arbol  
6     } }  
7
```

Ejemplo Introdutorio

- **Generación de todas las soluciones factibles**

es decir, $\{X = (x_1, x_2 \dots x_n), x_i \in \{0, 1\}, \sum_{i=1}^n x_i p_i \leq P\}$

Todas las soluciones factibles.

Llamada inicial: factibles(p, P, 0, x)

```
1 inline float peso(vector <float> &p, unsigned k, vector <short> &x){
2 float _peso=0;
3 for (unsigned i=0;i<=k;i++) _peso+=x[i]*p[i];
4 return _peso;
5 }
6
7 void factibles(vector <float> &p,float P,unsigned k,vector <short> &x){
8     for (unsigned j=0; j<2; j++) {
9         x[k]=j;
10        if (peso(p,k,x)<=P)
11            if (k==x.size()-1) imprimir_sol_factible(x);
12            else factibles(p,P,k+1,x);
13    }
14 }
```

Ejemplo Introductorio

- Complejidad temporal de la función “combinaciones”
 - **Coste exacto** $C_e^T(n)$: $f(n) = 2f(n-1) + 1 \in \Theta(2^n)$.
- Complejidad temporal de la función “factibles”
 - **Cota superior** $C_p^T(n)$: Todos los objetos caben en la mochila, $f(n) = 2f(n-1) + n \in O(2^n)$.
 - **Cota inferior** $C_m^T(n)$: Ningún objeto cabe en la mochila, $f(n) = f(n-1) + n \in \Omega(n^2)$.
 - Aunque el recorrido lineal que comprueba la restricción del peso se puede evitar:

Todas las soluciones factibles.

Llamada inicial: float p_a=0; factibles(p, P, 0, x, p_a)

```
1 void factibles(vector <float> &p,float P,unsigned k,  
2               vector <short> &x, float &peso_acum){  
3     for (unsigned j=0; j<2; j++) {  
4         x[k]=j;  
5         peso_acum+=x[k]*p[k];  
6         if (peso_acum<=P)  
7             if (k==x.size()-1) imprimir_sol_factible(x);  
8             else factibles(p,P,k+1,x);  
9         peso_acum-=x[k]*p[k];  
10    } }
```

• Búsqueda de una solución óptima

Solución óptima. Llamada inicial: float vMejor=-1; optima_v0(v, p, P, 0, x, xMejor, vMejor)

```
1 inline float valor(vector <float> & v, unsigned k, vector <short> & x){
2   float _valor=0;
3   for (unsigned i=0;i<=k;i++) _valor+=x[i]*v[i];
4   return _valor;
5 }
6
7 void optima_v0(vector <float> & v,vector <float> & p, float P,
8               unsigned k, vector <short> & x,
9               vector <short> & x_mejor, float & v_mejor){
10  llamadasRecurtivas++; //para comprobaciones de eficiencia
11  for (unsigned j=0; j<2; j++) {
12    x[k]=j;
13    if (peso(p,k,x)<=P)
14      if (k==x.size()-1) {
15        if (valor(v,k,x)>v_mejor){
16          x_mejor=x; v_mejor=valor(v,k,x); }
17      }
18    else optima_v0(v,p,P,k+1,x,x_mejor,v_mejor);
19  } }
20
```


- **Búsqueda de una solución óptima con poda basada en la mejor solución en curso**

¿Se puede mejorar la eficiencia de la función anterior?

Sí. Evitando (podando) la exploración de tuplas factibles que no van a mejorar la mejor solución que se tiene hasta el momento.

- Son las denominadas *tuplas no prometedoras*

Por ejemplo, cualquier tupla incompleta $(x_0, x_1 \cdots x_k, ?, ? \cdots ?)$ se puede descartar si no cumple la condición:

- $valor(v, k, x) + \sum_{i=k+1}^{n-1} v_i > v_{mejor}$

Ejemplo Introductorio

- Búsqueda de una solución óptima con poda basada en la mejor solución en curso

Solución óptima v1.
vMejor)

Llamada inicial: float vMejor=-1; optima_v1(v, p, P, 0, x, xMejor,

```
1 inline float suma_valores(vector <float> & v, unsigned k){
2 float _valor=0;
3 for (unsigned i=k;i<v.size();i++) _valor+=v[i];
4 return _valor;
5
6 void optima_v1(vector <float> & v,vector <float> & p, float P,
7             unsigned k, vector <short> & x,
8             vector <short> & x_mejor, float & v_mejor){
9     llamadasRekursivas++; //para comprobaciones de eficiencia
10    for (unsigned j=0; j<2; j++) {
11        x[k]=j;
12        if (peso(p,k,x)<=P)
13            if (k==x.size()-1) {
14                if (valor(v,k,x)>v_mejor){
15                    x_mejor=x; v_mejor=valor(v,k,x); }
16            }
17        else if (valor(v,k,x) + suma_valores(v,k+1) > v_mejor)
18            optima_v1(v,p,P,k+1,x,x_mejor,v_mejor);
19    } }
```

Interesa que los mecanismos de poda “actúen” lo más arriba posible en el árbol, es decir, cuanto más incompleta esté la tupla

- En este sentido, la poda anterior se puede mejorar mediante una estimación más realista (menos optimista):
- Teniendo en cuenta que el mayor beneficio que se puede obtener viene dado por la solución al problema de la mochila continua (con solución voraz), se descarta toda tupla incompleta que no cumple:
 - $valor(v, k, x) + mochila_continua(v', p', P', k + 1) > v_mejor$
 - donde $mochila_continua(v', p', P', k + 1)$ es la solución al problema de la mochila con fraccionamiento considerando que los k primeros objetos ya han sido tratados.
 - Se trata de una condición más restrictiva y por tanto con mayor capacidad de poda

Ejemplo Introductorio

- Búsqueda de una solución óptima con poda basada en la mejor solución en curso

Solución óptima v2.

Llamada inicial: float vMejor=-1; optima_v2(v, p, P, 0, x, xMejor, vMejor)

```
1 float mochila_continua(vector<float> v,vector<float> p,float P, int j);
2
3
4 void optima_v2(vector <float> & v,vector <float> & p, float P,
5             unsigned k, vector <short> & x,
6             vector <short> & x_mejor, float & v_mejor){
7     llamadasRecurativas++; //para comprobaciones de eficiencia
8     for (unsigned j=0; j<2; j++) {
9         x[k]=j;
10        if (peso(p,k,x)<=P)
11            if (k==x.size()-1) {
12                if (valor(v,k,x)>v_mejor){
13                    x_mejor=x; v_mejor=valor(v,k,x); }
14            }
15            else if (valor(v,k,x) +
16                    mochila_continua(v,p,P-peso(p,k,x),k+1) > v_mejor)
17                optima_v2(v,p,P,k+1,x,x_mejor,v_mejor);
18    }
```

Ejemplo Introductorio

- Búsqueda de una solución óptima con poda basada en la mejor solución en curso

Sol. óptima v2. Iterativo

Llam. inicial: float vMejor=-1; optima_v2I(v, p, P, 0, x, xMejor, vMejor)

```
1 void optima_v2I(vector <float> & v, vector <float> & p, float P,
2             unsigned k, vector <short> & x,
3             vector <short> & x_mejor, float & v_mejor){
4     x[k]=-1;
5     while (k>-1){
6         while (x[k]<1){
7             x[k]++;
8             if (peso(p,k,x)<=P)
9                 if (k==x.size()-1) {
10                     if (valor(v,k,x)>v_mejor){x_mejor=x; v_mejor=valor(v,k,x); }
11                 }
12                 else if (valor(v,k,x) +
13                     mochila_continua(v,p,P-peso(p,k,x),k+1) > v_mejor){
14                     k++; //avanzar en la tupla: sig. nivel en el arbol
15                     x[k]=-1;
16                 }
17             }
18             k--; //no quedan mas posibilidades: retroceder al nivel superior.
19         } }
```

Ejemplo Introductorio

- Los mecanismos de poda basados en la mejor solución en curso son mucho más eficientes si se llega pronto a una “buena” solución
 - En este sentido, la forma en la que se “despliega el árbol” puede ser relevante:
 - Por ejemplo, para este problema: completar la tupla primero con los unos y después con los ceros
 - Pero lo que, en general, aumenta drásticamente la eficiencia es partir de una solución factible:
 - Cuanto más se aproxime a la solución óptima mejor
 - Aunque no se trata de una cota optimista en el sentido estudiado anteriormente (puesto que es imprescindible que sea factible)
 - Por ejemplo, para este problema: se puede partir de la solución (subóptima) que aporta el método voraz a la mochila discreta

Solución óptima v2 partiendo de un subóptimo.

```
1 v_mejor=mochila_discreta_voraz(v,p,P,x_mejor); //Heuristico voraz
2 optima_v2(v,p,P,0,x,x_mejor,v_mejor);
3
```

Ejemplo Introdutorio

- Análisis de eficiencia: 25 muestras aleatorias de tamaño $n = 30$.

Tipo de poda ¹	Partiendo de un subóptimo voraz ²	Llamadas recursivas realizadas (promedio)	Tiempo medio (segundos)
Ninguna (v0)	–	1054.8×10^6	875.65
Completando con todos los objetos restantes (v1)	No	925.5×10^3	0.112
	Si	389.0×10^3	0.072
Completando según la sol. voraz mochila continua (v2)	No	2.3×10^3	0.034
	Si ³	18	0.002

¹Estimando el beneficio que se obtendrá con el resto de la tupla (poda basada en la mejor solución en curso)

²Heurístico voraz: selección de objetos de mayor valor específico. Objetos no fraccionables.

³En la práctica, resulta elevada la probabilidad de que el subóptimo obtenido mediante el heurístico voraz coincida con el óptimo global

Vuelta atrás: definición y ámbito de aplicación

- Algunos problemas sólo podemos resolverlos mediante la obtención y 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)
- Para llevar a cabo el estudio exhaustivo *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 / 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

- 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 tengan 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.
 - 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 $(x_1, x_2 \cdots x_{i-1} \cdots)$ 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 en preorden sobre una estructura arbórea imaginaria.

Vuelta atrás: Esquema general recursivo

Esquema recursivo de *backtracking*.

Llamada inicial: `backtracking_R(P,0,x)`

```
1 void backtracking_R (problema P; int k; tupla & x){
2
3   x[k]= preparar_recorrido_nivel_k;
4   while (existe_hermano_nivel_k){
5     x[k]=siguiente_hermano_nivel_k;
6     if (factible(x,k))
7       if (completada(x)) tratar(x) //la mejor, todas, una cualquiera, etc.
8       else [if (prometedora(x,k)) /*solo si se busca un optimo*/]
9         backtracking_R(P,k+1,x)
10  }
11 }
```

Vuelta atrás: Esquema general iterativo

Esquema iterativo de *backtracking*.

Llamada inicial: `bactracking_I(P,0,x)`

```
1 void bactracking_I (problema P){
2     tupla x;
3     int k = 0;
4     x[k]= preparar_recorrido_nivel_k;
5     while (k>-1){
6         while (existe_hermano_nivel_k){
7             x[k]=siguiente_hermano_nivel_k;
8             if (factible(x,k))
9                 if (completada(x)) tratar(x) //la mejor, todas, la primera...
10                else [if (prometedora(x,k)) /*solo si se busca un optimo*/]{
11                    k++; //avance
12                    x[k]= preparar_recorrido_nivel_k;
13                }
14        }
15        k--; //retroceso
16    }
17 }
```

Voraz

- Los dominios son continuos
- Buscamos la solución óptima y existe un criterio de decisión voraz que nos conduce a ella.
- Buscamos una aproximación a la solución óptima (un subóptimo).

Backtracking

- Los dominios no son continuos
- Buscamos todas las soluciones factibles o todas las soluciones óptimas
- Buscamos una solución óptima y el problema no tiene solución voraz.

- El viajante de comercio (*the travelling salesman problem*)
- Permutaciones de los elementos de una lista
- El problema de las 8 reinas
- La función compuesta mínima

Vuelta Atrás. 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.

Vuelta Atrás. Ejercicios

El viajante de comercio

Solución:

- La expresaremos 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,
 $V = \{1, 2, \dots, n\}, \quad n = |V|$
 - 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:

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

Vuelta Atrás. Ejercicios

El viajante de comercio

Viajante de comercio - v0

```
1 void viajante(grafo g){
2     unsigned n=g.V.size();
3     vector <short> x_mejor, x(n);
4     x[0]=1; //el primer vertice queda fijado
5     unsigned k=1; float v_mejor=INT_MAX; x[k]=1;
6     while (k>0){
7         while (x[k]<n){
8             x[k]++;
9             if (hayArista(g,x[k-1],x[k]) && !repetido(x,k))
10                 if (k==n-1){
11                     if (hayArista(g,x[k],x[0])&&peso_camino(x,k)<v_mejor){
12                         v_mejor=peso_camino(x,k); x_mejor=x; }
13                 }
14                 else{ k++; x[k]=1;}
15             }
16             k--;
17         }
18         if (v_mejor==INT_MAX) cout << "Sin_solucion" << endl;
19         else {cout << "Solucion: "; mostrar(x_mejor);
20     }
```

Vuelta Atrás. Ejercicios

El viajante de comercio

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

Vuelta Atrás. Ejercicios

Permutaciones

Dada una lista L de n elementos cualesquiera, escribir un algoritmo que muestre todas las permutaciones de sus elementos.

- Solución:

- Sea $X = (x_1, x_2, \dots, x_n)$ $x_i \in \{1, 2, \dots, n\}$ un vector que indica qué elemento de L , identificado por su posición x_i , se debe mostrar en i -ésimo lugar.
- Por tanto, cada permutación vendrá por cada uno de los vectores distintos X que se puedan obtener.
- Restricción: X no puede tener elementos repetidos:
 - $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
- No hay función objetivo: se buscan todas las combinaciones factibles.

Vuelta Atrás. Ejercicios

Permutaciones

Permutaciones (v0,iterativo)

Restricciones con coste lineal

```
1 inline bool repetido(vector <unsigned> &x, int k){
2     unsigned i=0;
3     while (x[i]!=x[k]) i++;
4     return i!=k; }
5
6 void permuta_v0(lista <T> &L){
7     unsigned n=L.size(); vector <unsigned> x(n); int k=0;
8
9     x[k]=-1; //asumimos que el primer elemento en L ocupa la posicion 0
10    while (k>-1){
11        while (x[k]<n-1){
12            x[k]++;
13            if (!repetido(x,k)){
14                if (k==n-1) mostrar_permutacion(L,x);
15                else {k++; x[k]=-1;}
16            }
17        }
18        k--;
19    } }
```

Vuelta Atrás. Ejercicios

Permutaciones

- Restricciones con coste constante
 - Asumiendo este coste en el operador [] de los vectores "x" y "yaUtilizada"

Permutaciones (v1, iterativo)

Restricciones con coste constante









```
1 void permuta_v1(lista <T> &L){
2     unsigned n=L.size(); vector <unsigned> x(n); int k=0;
3     vector <bool> yaUtilizada(n,false); //n comp. inicial. con "false"
4     x[k]=-1; //asumimos que el primer elemento en L ocupa la posicion 0
5     while (k>-1){
6         while (x[k]<n-1){
7             x[k]++;
8             if (!yaUtilizada[x[k]]){
9                 yaUtilizada[x[k]]=true;
10                if (k==n-1) mostrar_permutacion(L,x);
11                else { k++; x[k]=-1; }
12                yaUtilizada[x[k]]=false;
13            }
14        }
15        k--;
16    } }
```

Vuelta Atrás. Ejercicios

El problema de las 8 reinas

En un tablero de ajedrez (8×8) obtener todas las formas de colocar 8 reinas de forma que no se “ataquen” mutuamente (ni en la misma fila, ni columna, ni diagonal)⁴.

- Ejemplo:

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

⁴De forma trivial se puede generalizar a n reinas y un tablero $n \times n$.

Vuelta Atrás. Ejercicios

El problema de las 8 reinas

Solución:

- Como no puede haber dos reinas en la misma fila, la reina i se colocará en la fila i :
 - El problema entonces es determinar la columna en la que colocarla.
- Sea $X = (x_1, x_2, \dots, x_n)$ donde $x_i \in \{1, 2, \dots, n\}$ representa la 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, es decir, X no puede tener elementos repetidos:
 - $i \neq j \rightarrow x_i \neq x_j \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$
 - 3 No puede haber dos reinas en la misma diagonal:
 - $i \neq j \rightarrow |i - j| \neq |x_i - x_j| \quad \forall i, j : 1 \leq i \leq n \quad 1 \leq j \leq n$

Vuelta Atrás. Ejercicios

El problema de las 8 reinas

8_reinas (recursivo)

Llamada inicial: 8_reinas(0,x)

```
1 inline bool factible(vector <unsigned> &x, int k){
2     bool _factible=true; short i=0:
3     while (i<k && _factible){
4         _factible= (x[i]!=x[k] && (k-i)!= abs(x[k]-x[i]));
5         i++;
6     }
7     return _factible;
8 }
9 void 8_reinas(short k, vector <short> &x){
10     x[k]=-1;
11     while (x[k]<7){
12         x[k]++;
13         if (factible(x,k))
14             if (k==7) mostrar_solucion(x);
15             else 8_reinas(k+1,x)
16     }
17 }
```


Vuelta Atrás. Ejercicios

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 función compuesta que transforma 3 en 6 con operaciones f y g es: (consta de 5 composiciones)

$$(g \circ f \circ g \circ f \circ g)(3) = 6$$

- Pero la mínima, formada por tres composiciones, es: (aunque no es la única)

$$(f \circ g \circ g \circ f)(3) = 6$$

Vuelta Atrás. Ejercicios

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 (a diferencia de los ejemplos anteriores)
 - Se pretende minimizar el tamaño de la tupla solución (función objetivo)
 - Para evitar ramas infinitas en el árbol de búsqueda, asumiremos un máximo de composiciones M
 - Sea $F(X, k, x)$ el resultado de aplicar al valor x la composición representada en la tupla X hasta su posición k , es decir,

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

- Restricciones:
 - $F(X, k, x) \neq F(X, i, x) \ \forall i < k$, para evitar ciclos
 - $k < M$, para evitar búsquedas infinitas
 - $F(X, k, x) \neq 0$
 - $k < v_{\text{mejor, tupla "prometedora"}}$

Vuelta Atrás. Ejercicios

La función compuesta mínima

```
1 void composicion(unsigned x, unsigned y, unsigned M){
2     vector <short> X_mejor, X(M+1);
3     vector <unsigned> valor(M+1); //Almacen del resultado de la composicion
4     short k=0, v_mejor=M+1;
5
6     X[k]=2;
7     while (k>-1){
8         while (X[k]>0){
9             X[k]--;
10            valor[k]=F(X,k,x);
11            if (valor[k]==y) { if (k<v_mejor){v_mejor=k; X_mejor=X;} }
12            else if (k<v_mejor && // Tupla prometedora
13                    k<M && valor[k]!=0 && !repetido(valor,k)){
14                k++; X[k]=2;}
15        }
16        k--;
17    }
18    if (v_mejor==M+1) cout<<"Sin solución con " << M <<" composiciones.";
19    else { cout << "Solución encontrada:" << endl;
20          cout << "Número de composiciones:" << v_mejor << endl;
21    } }
```

1 El coloreado de grafos (el coloreado de mapas)

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.

2 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.

3 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 $(n \times m)$ con cuatro tipos de movimiento (arriba, abajo, derecha, izquierda).

4 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.

5 El reparto de paquetes

- Una empresa de transportes dispone de M vehículos para repartir N paquetes todos al mismo destino. Cada paquete i tiene un peso P_i y tiene que estar entregado antes de un tiempo TP_i . Por otra parte, cada vehículo j puede transportar una carga máxima C_j , tarda un tiempo TV_j en llegar al destino y consume una cantidad L_j de litros de combustible independiente de la carga que transporte.
- Diseñar un algoritmo que obtenga la forma en que se deben transportar los objetos (en qué vehículo j debe ir cada objeto i) para que el consumo sea mínimo.

6 La empresa naviera

- 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).

7 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; T 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.

8 Sudoku

- 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.