TEMA 2. ALGORITMOS VORACES

1. Introducción.

Los algoritmos voraces constituyen uno de los esquemas más simples y al mismo tiempo de los más utilizados. Típicamente se emplea para resolver problemas de optimización. Las condiciones que se cumplen son:

- Existe una entrada de tamaño n que son los candidatos a formar parte de la solución;
- Existe un subconjunto de esos n candidatos que satisface ciertas restricciones: se llama solución factible;
- Hay que obtener la solución factible que maximice o minimice una cierta función objetivo: se llama solución óptima.

El esquema voraz procede por pasos:

- Inicialmente el conjunto de candidatos escogidos está vacío.
- En cada paso, se intenta añadir al conjunto de los escogidos "el mejor" de los no escogidos (sin pensar en el futuro), utilizando una función de selección basada en algún criterio de optimización (puede ser o no ser la función objetivo).

Tras cada paso, hay que ver si el conjunto seleccionado es completable (es decir, si añadiendo más candidatos se puede llegar a una solución);

- Si el conjunto no es completable, se rechaza el último candidato elegido y no se vuelve a considerar en el futuro;
- Si es completable, se incorpora al conjunto de escogidos y permanece siempre en él;

Tras cada incorporación se comprueba si el conjunto resultante es una solución, de manera que el algoritmo termina cuando se obtiene una solución. Se considera que el algoritmo es correcto si la solución encontrada es siempre óptima. Sin embargo, aunque los algoritmos voraces suelen ser eficientes y fáciles de diseñar e implementar, no todos los problemas se pueden resolver utilizando algoritmos voraces, ya que a veces no encuentran la solución óptima o incluso no encuentran ninguna solución cuando el problema sí la tiene.

El esquema de los algoritmos voraces es el siguiente:

```
función voraz(C: conjunto): conjunto
    {C es el conjunto de candidatos}
    S:= ∅
    mientras que C≠∅ y no solución(S)
            x:=elemento de x que maximiza seleccionar(C)
            C:=C-{x}
            si factible{SU{x}} entonces S:= SU{x}
    si solución(S) entonces devolver S
    sino devolver "no hay soluciones"
```

2. El problema del cambio de monedas.

El objetivo de este problema es devolver una cantidad de dinero con el menor número posible de monedas, partiendo de un conjunto de tipos de monedas válidas, de las que se supone que hay cantidad suficiente para realizar el desglose, y de un importe a devolver. Los elementos fundamentales del esquema son:

- Conjunto de candidatos: cada una de las monedas de los diferentes tipos que se pueden usar para realizar el desglose del importe dado.
- Solución: un conjunto de monedas devuelto tras el desglose y cuyo valor total es igual al importe a desglosar.
- Completable: la suma de los valores de las monedas escogidas en un momento dado no supera el importe a desglosar.
- Función de selección: elegir si es posible la moneda de mayor valor de entre las candidatas.
- Función objetivo: número total de monedas utilizadas en la solución (debe minimizarse).

Esquema básico:

```
función devolver cambio(n): conjunto de monedas
    {Da el cambio de n unidades utilizando el menor número posible de monedas. La
    constante C especifica las monedas disponibles}
    const C=[100,25,10,5,1}
    S≠∅ {S es un conjunto que contendrá la solución}
    s=0 {s es la suma de los elementos de S}
    mientras s≠n hacer
            x:=el mayor elemento de C tal que s+x≤n
            si no existe ese elemento entonces devolver "no encuentro solución"
            S=SU{una moneda de valor x}}
            s=s+x
    devolver S
```

3. Problema de la mochila.

En este problema se dispone de una mochila que soporta un peso máximo W y existen n objetos que podemos cargar en la mochila, cada objeto tiene un peso w_i y un valor v_i. El objetivo es llenar la mochila maximizando el valor de los objetos que transporta. Suponemos que los objetos se pueden romper, de forma que podemos llevar una fracción x_i ($0 \leq x_i \leq 1$) de un objeto.

Matemáticamente se trata de maximizar $\Sigma x_i v_i$ (valor de la carga) con la restricción $\Sigma x_i w_i \leq W$ (peso de la carga menor que el peso total) donde $v_i > 0$, $w_i > 0$ y $0 \leq x_i \leq 1$ para $1 \leq i \leq n$.

Esquema básico:

```
función mochila (w[1..n],v[1..n],W):matriz[1..n]
    {Inicialización}
    para i=1 hasta n hacer x[i]:=0
    peso=0
    {bucle voraz}
    mientras peso<N hacer
        i:=el mejor elemento restante
        si peso+w[i]≤W entonces
            x[i]:=1
            peso:=peso+w[i]
        sino
            x[i]:=(W-peso)/w[i]
            peso:=W
    devolver x
```

Ejemplo: n=5; W=100.

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	$\sum_{i=1}^{n} w_i > W$

A la hora de seleccionar hay tres posibilidades:

1. ¿Objeto más valioso? ↔ v_i max
2. ¿Objeto más ligero? ↔ w_i min
3. ¿Objeto más rentable? ↔ v_i/w_i max

Dependiendo del caso cambiará el valor de la función objetivo:

	1	2	3	4	5	
v_i	20	30	66	40	60	
w_i	10	20	30	40	50	
v_i/w_i	2.0	1.5	2.2	1.0	1.2	Objetivo $\sum_{i=1}^{n} x_i w_i$
xi (v_i max)	0	0	1	0.5	1	146
xi (w_i min)	1	1	1	1	0	156
xi (v_i/w_i max)	1	1	1	0	0.8	164

Por tanto, en la primera estrategia se trata de elegir el objeto con mayor beneficio total. Sin embargo, podría ser que su peso sea excesivo y que la mochila se llene muy rápido con poco beneficio total. En la segunda estrategia se eliger el objeto que pese menos, para acumular beneficios de un número mayor de objetos. Sin embargo, es posible que se elijan objetos con poco beneficio simplemente porque pesan poco. La tercera estrategia, que es la óptima, es tomar los objetos que proporcionen mayor beneficio por unidad de peso. Es importante notar que los algoritmos resultantes de aplicar cualquiera de las dos primeras

estrategias también son voraces, pero no calculan la solución óptima. Por tanto, la mayor dificultad de un algoritmo voraz es elegir correctamente la función de selección.

Si calculamos la eficiencia del algoritmos, existen dos posibilidades:

1) Implementación directa: en este caso el bucle requiere como máximo n (num. obj.) iteraciones: O(n) (una para cada posible objeto en el caso de que todos quepan en la mochila) y la función de selección requiere buscar el objeto con mejor relación valor/peso: O(n). Por tanto, el coste del algoritmo es

$$O(n)\cdot O(n)=O(n^2)$$

2) Implementación preordenando los objetos: primero se ordenan los objetos de mayor a menor relación valor/peso (existen algoritmos de ordenación O(nlogn)). Con los objetos ordenados la función de selección es O(1). Por tanto, el coste del algoritmo es

$$O(\max(O(n\log n), O(n)\cdot O(1)))=O(n\log n)$$

4. Árboles de recubrimiento mínimo

El objetivo de este problema es, partiendo de un grafo dado, obtener un nuevo grafo que solo contenga las aristas imprescindibles para una optimización global de las conexiones entre todos los nodos, entendiendo como "optimización global" que algunos pares de nodos pueden no quedar conectados entre ellos con el mínimo coste posible en el grafo original. Normalmente este problema se aplica a situaciones que tienen que ver con distribuciones geográficas como, por ejemplo, un conjunto de computadores distribuidos geográficamente en diversas ciudades de diferentes países a los que se quiere conectar para intercambiar datos, compartir recursos, etc.; solicitud a las diferentes compañías telefónicas los precios de alquiler de líneas entre ciudades; o asegurar que todos los computadores pueden comunicar entre sí, minimizando el precio total de la red.

Definiciones importantes para este problema:

1) Árbol libre: es un grafo no dirigido, conexo y acíclico
   - todo árbol libre con n vértices tiene n-1 aristas
   - si se añade una arista se introduce un ciclo
   - si se borra una arista quedan vértices no conectados
   - cualquier par de vértices está unido por un único camino simple
   - un árbol libre con un vértice distinguido es un árbol con raíz

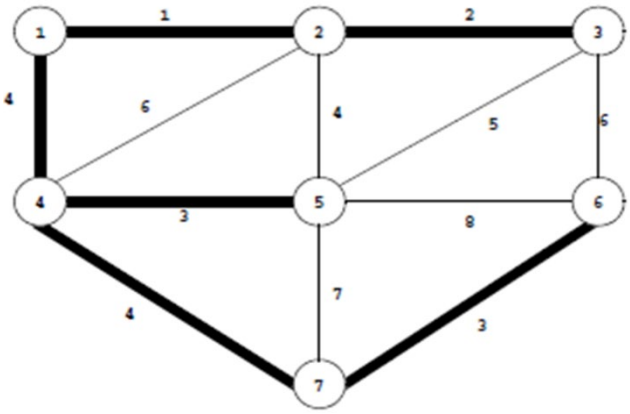
2) Árbol de recubrimiento de un grafo no dirigido y etiquetado no negativamente: es cualquier subgrafo que contenga todos los vértices y que sea un árbol libre.
3) Árbol de recubrimiento de coste mínimo: es un árbol de recubrimiento y no hay ningún otro árbol de recubrimiento cuya suma de aristas sea menor.

Propiedad fundamental de los árboles de recubrimiento de coste mínimo: "Sea g un grafo no dirigido conexo y etiquetado no negativamente, $g \in \{f:VxV\rightarrow E\}$,sea U un conjunto de vértices, $U\subset V$, $U\neq\emptyset$, si <u,v> es la arista más pequeña de g tal que $u\in U$ y $v \in V\backslash U$, entonces existe algún árbol de recubrimiento de coste mínimo de g que la contiene".
Partiendo de esa propiedad se puede diseñar un algoritmo voraz para resolver el problema, cuyos elementos fundamentales son:

- Función objetivo: minimizar la longitud del árbol de recubrimiento.
- Candidatos: las aristas del grafo.
- Función solución: árbol de recubrimiento de longitud mínima.
- Función factible: Conjunto de aristas que no contiene ciclos.
- Función de selección: varía con el algoritmo;
  -Seleccionar la arista de menor peso que aún no ha sido seleccionada y que no forme un ciclo (Algoritmo Kruskal).
  - Seleccionar la arista de menor peso que aún no ha sido seleccionada y que forme un árbol junto con el resto de aristas seleccionadas (Algoritmo Prim).

4.1. Algoritmo de Kruskal.

Se trata de seleccionar la arista de menor peso que aún no haya sido seleccionada y que no conecte dos nudos de la misma componente conexa, es decir, que no forme un ciclo. Ejemplo:



El algoritmo operaría como indica la siguiente tabla:

Paso	Arista considerada	Componentes conexas
-	-	{1},{2},{3},{4},{5},{6},{7}
1	{1,2}	{1,2},{3},{4},{5},{6},{7}
2	{2,3}	{1,2,3},{4},{5},{6},{7}
3	{4,5}	{1,2,3},{4,5},{6},{7}
4	{6,7}	{1,2,3},{4,5},{6,7}
5	{1,4}	{1,2,3,4,5},{6,7}
6	{2,5}	forma ciclo
7	{4,7}	{1,2,3,4,5,6,7}

En forma de seudocódigo:

```
función Kruskal (G=<N,A>: grafo, longitud: A→R+): conjunto de aristas
    {Inicialización}
    Ordenar A por longitudes crecientes
    n:= el número de nodos que hay en N
    T≠∅ {contendrá las aristas del árbol de recubrimiento mínimo}
    Iniciar n conjuntos, cada uno de los cuales contiene un elemento distinto de N
    {bucle voraz}
    repetir
        e:= (u,v) {arista más corta, aún no considerada
        compu:=buscar(u)
        compv:=buscar(v)
        si compu≠compv entonces
            fusionar(compu,compv)
            T:=TU{e}
    hasta que T contenga n-1 aristas
    devolver T
```

Análisis de eficiencia: suponemos con |N|=n^|A|=a. Entonces el coste de cada operación es:

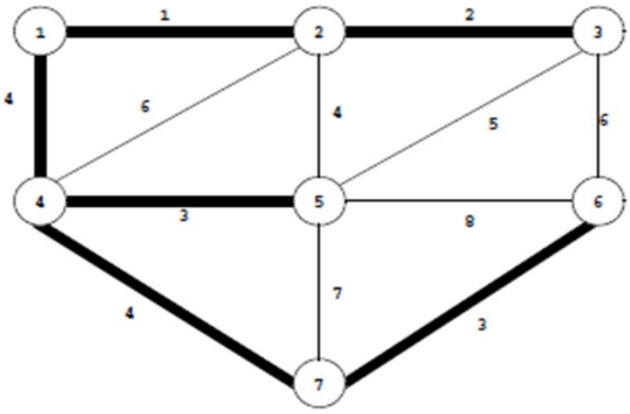
- Ordenar A: O(aloga)≡O(alogn) ya que $n-1 \leq a \leq n(n-1)/2$
- Inicializar n conjuntos disjuntos: O(n)
- Para las operaciones buscar y fusionar: O(alogn)
- Para las demás operaciones: O(a).

Por tanto:

$$T(n)=O(alogn)$$

4.2. Algoritmo de Prim.

En este caso se trata de comenzando por un nodo cualquiera, seleccionar la arista de menor peso que aún no haya sido seleccionada y que forme un árbol junto con el resto de aristas seleccionadas.



El algoritmo de Prim aplica reiteradamente la propiedad de los árboles de recubrimiento de coste mínimo incorporando a cada paso una arista. Se usa un conjunto B de vértices tratados y se selecciona en cada paso la arista mínima que une un vértice de B con otro de su complementario.

- Inicialización: B = {nodo arbitrario de N} = {1},
  T vacío.
- Selección: arista más corta que parte de B:
  (u, v), u∈B ^ v∈N\B → se añade (u, v) a T y v a B
  T define en todo momento un árbol expandido mínimo del subgrafo (B,A)
- Final del algoritmo: B = N. En este punto, T ofrece una solución optima al problema.

Por tanto, el algoritmo de Kruskal es un bosque que crece, en cambio, el algoritmo de Prim es un único árbol que va creciendo hasta alcanzar todos los nodos y calcula el árbol expandido mínimo. Por ejemplo:

Paso	Selección	B
ini	-	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{7,6}	{1,2,3,4,5,6,7}

La implementación del algoritmo de Prim sería:

```
función Prim (L[1..n, 1..n]): conjunto de aristas
    {Inicialización: solo el nodo 1 se encuentra en B}
    T≠∅ {contendrá las aristas del árbol de recubrimiento mínimo}
    para i:=2 hasta n hacer
        más próximo[i]:=1
        distmin[i]:=L[i,1]
    {bucle voraz}
    repetir n-1 veces
        min:=∞
        para j:=2 hasta n hacer
            si 0≤distmin[j]<min entonces
                min:=distmin[j]
                k:=j
        T:=TU{más próximo[k],k}
        distmin[k]
        para j:=2 hasta n hacer
            si L[j,k]<distmin[j] entonces
                distmin[k]:=L[j,k]
                más próximo[j]:=k
    devolver T
```

Calculando su eficiencia, tenemos dos fases:

- Inicialización = O(n)
- Bucle voraz: n−1 iteraciones, cada para anidado de O(n). Total: T(n) = $O(n^2)$

Por tanto, con estructuras de datos más eficientes (montículo) se podría mejorar a O(alogn), igual que Kruskal, como recoge esta tabla:

	Prim $O(n^2)$	Kruskal O(alogn)
Grafo denso: a→n(n-1)/2	**$O(n^2)$**	$O(n^2 \log n)$
Grafo disperso: a→n	$O(n^2)$	**O(nlogn)**

5. Caminos mínimos en grafos.

En este problema se parte de un grafo etiquetado con pesos no negativos y se trata de encontrar el camino de longitud mínima entre dos vértices dados, entendiendo como longitud o coste de un camino suma de los pesos de las aristas que lo componen.

5.1. Algoritmo de Dijkstra.

El problema consiste en dado un grafo G = (N,A) dirigido, con longitudes en las aristas ≥0,con un nodo distinguido como origen de los caminos (el nodo 1), encontrar los caminos mínimos entre el nodo origen y los demás nodos de N. Para ello, se supone que el camino mínimo de un nodo a sí mismo tiene coste nulo y un valor ∞ en la posición w del vector indica que no hay ningún camino desde v a w.

El algoritmo se utiliza para grafos dirigidos (aunque la extensión a no dirigidos es inmediata). El algoritmo genera uno a uno los caminos de un nodo v al resto por orden creciente de longitud y usa un conjunto de vértices donde, a cada paso, se guardan los nodos para los que ya se sabe el camino mínimo. En cada paso, el algoritmo devuelve un vector indexado por vértices: en cada posición w se guarda el coste del camino mínimo que conecta v con w, de manera que cada vez que se incorpora un nodo a la solución se comprueba si los caminos todavía no definitivos se pueden acortar pasando por él.

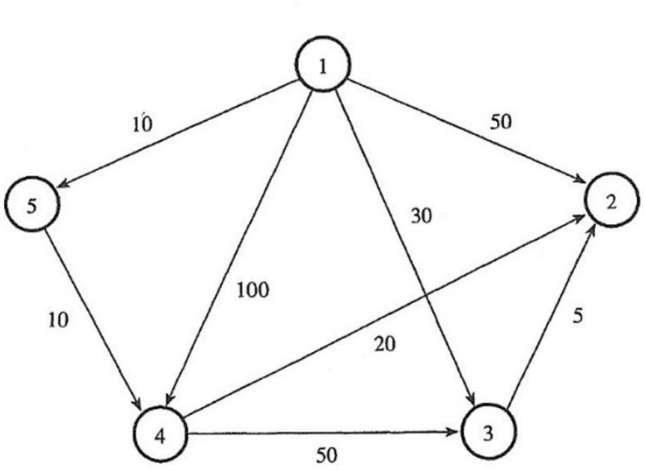
La técnica voraz consiste en lo siguiente:

- 2 conjuntos de nodos: S: seleccionados (camino mínimo establecido); C: candidatos (los demás).
- Invariante: N=SUC
- Inicialmente S=1→final: S=N: función solución.
- Función selección: nodo de C con menor distancia conocida desde 1→existe una información provisional sobre distancias mínimas.

La implementación sería:

```
función Dijkstra (L[1..n, 1..n]): matriz[2..n]
    matriz D[2..n]
    {Inicialización}
    C:={2,3,..,n}{S=N\C solo existe implícitamente}
    para i:=2 hasta n hacer D[i]:=L[1,i]
    {bucle voraz}
    repetir n-2 veces
        v:=algún elemento de C que minimiza D[v]
        C:=C\{v} {e implícitamente S:=S\{v}}
        para cada w∈C hacer
            D[w]:=min(D[w],D[v]+L[v,w])
    devolver D
```

Ejemplo:



Paso	Selección	C	D[2]	D[3]	D[4]	D[5]
Ini	-	{2,3,4,5}	50	30	100	**10**
1	5	{2,3,4}	50	30	**20**	10
2	4	{2,3}	40	**30**	20	10
3	3	{2,}	35	30	20	10

6. Problemas de planificación.

En este apartado se presentan dos tipos de problemas relativos a planificar tareas en una sola máquina. Por un lado, se trata de minimizar el tiempo medio que invierte cada tarea en el sistema y, por otro, las tareas tienen un plazo fijo de ejecución, y cada tarea aporta unos ciertos beneficios solo si está acabada al llegar su plazo. El objetivo en ambos casos es maximizar la rentabilidad.

6.1. Minimización de tiempo en el sistema.

Supongamos que tenemos tres clientes con t_1=5, t_2=10 y t_3=3. Existen seis ordenes de servicio posibles.

Orden	T	
123:	5+(5+10)+(5+10+3)=38	
132:	5+(5+3)+(5+3+10)=31	
213:	10+(10+5)+(10+5+3)=43	
231:	10+(10+3)+(10+3+5)=41	
312:	3+(3+5)+(3+5+10)=29	→ óptimo
321:	3+(3+10)+(3+10+5)=34	

Obviamente, en este caso, la planificación optima se obtiene cuando se sirve a los tres clientes por orden creciente de tiempos de servicio: el cliente 3, que necesita el menor tiempo se sirve el primero, mientras que el cliente 2 que necesita más tiempo se le sirve el último. Si servimos a los clientes por orden decreciente de tiempos de servicio se obtiene la peor planificación. Por tanto, un algoritmo voraz que construye la planificación optima elemento a elemento lo único que se necesita es ordenar los clientes por orden de tiempo no decreciente de servicio, lo cual requiere un tiempo que está en O(nlogn).

6.2. Planificación con plazo fijo.

En este caso el sistema tiene que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En cualquier instante podemos ejecutar únicamente una tarea. La tarea i produce unos beneficios g_i>0 solo en el caso de que sea ejecutada en un instante anterior a d_i.

Por ejemplo, con n=4 y los valores siguientes:

i	1	2	2	4
g_i	50	10	15	30
d_i	2	1	2	1

Las planificaciones que hay que considerar y los beneficios correspondientes son:

Secuencia	Beneficio	Secuencia	Beneficio	
1	50	2,1	60	
2	10	2,3	25	
3	15	3,1	65	
4	30	4,1	80	→ óptimo
1,3	65	4,3	45	

La secuencia 3,2 no se considera porque la tarea 2 se ejecutaría en el instante t=2, después de su plazo que es d_2=1. Para maximizar nuestros beneficios deberíamos ejecutar la planificación 4,1.

Se dice que un conjunto de tareas es factible si existe al menos una sucesión que permite que todas las tareas del conjunto se ejecuten antes de sus respectivos plazos. Un algoritmo

voraz consiste en construir la planificación paso a paso, añadiendo en cada paso la tarea que tenga el mayor valor de g_i entre las que aún no se hayan considerado, siempre y cuando el conjunto de tareas seleccionadas siga siendo factible.

En el ejemplo seleccionamos primero la tarea 1. Después la tarea 4; el conjunto {1,4} es factible porque se puede ejecutar en el orden 4,1. El conjunto {1,3.4} no es factible, por tanto se rechaza al tarea 3. El conjunto {1,2,4} tampoco es factible y se rechaza la tarea 2. Nuestra solución, en este caso optima, es ejecutar el conjunto de tareas {1,4}, que solo se puede efectuar en el orden 4,1.

Sea J un conjunto de k tareas. Supongamos que las tareas están numeradas de tal forma que $d_1 \leq d_2 \leq \ldots \leq d_k$. Entonces el conjunto J es factible si y solo si la secuencia 1,2,…k es factible. En este caso, el algoritmo voraz siempre encuentra una planificación optima.

Para implementar el algoritmo, supongamos que las tareas están numeradas de tal manera que $g_1 g_2 \ldots g_n$. Suponemos además que $n>0$ y $d_i>0$ para $1 \leq i \leq n$, y que está disponible un espacio adicional al principio de los vectores d (que contiene los plazos) y j (en donde se construye la solución). Estas celdas adicionales se llaman “centinelas”. Almacenando un valor adecuado en los centinelas se evitan comprobaciones repetitivas de rangos que consumen mucho tiempo. El seudocódigo del algoritmo es como sigue:

```
función secuencia (d[0..n]): k,matriz[1..k]
    matriz j[0..n]
    {La planificación se construye paso a paso en la matriz j. La variable k dice
    cuántas tareas están ya en la planificación}
    d[0]:=j[0]:=0 {centinelas}
    k:=j[1]:=1 {la tarea 1 siempre se selecciona}
    {bucle voraz}
    para i:=2 hasta n hacer {orden decreciente de g}
        r:=k
        mientras d[j[r]]>max(d[i],r) hacer r:=r-1
        si d[i]>r entonces
            para m:=k paso -1 hasta r+1 hacer j[m+1]:=j[m]
            j[r+1]:=i
            k:=k+1
    devolver k,j[1..k]
```

Las k tareas de la matriz j están por orden creciente de plazo. Cuando se esta considerando la tarea i, el algoritmo comprueba si se puede insertar en j en el lugar oportuno sin llevar alguna tarea que ya esté en j más allá de su plazo. De ser así, se acepta i, sino se rechaza. Ejemplo con seis tareas:

i	1	2	2	4	5	6
g_i	20	15	10	7	5	3
d_i	3	1	1	3	1	3

Eficiencia del algoritmo: la ordenación de tareas por orden decreciente de beneficio requiere un tiempo que está en O(nlogn). El caso peor es cuando clasifica las tareas por orden decreciente de plazos, y cuando todas ellas tienen cabida en la planificación. En este caso, cuando se está considerando la tarea i el algoritmo examina las k=i-1 tareas que ya están planificadas, para encontrar un lugar para el recién llegado, y después desplaza

todo un lugar. Hay $\sum_{k=1}^{n=1} k$ pasadas por el bucle **mientras** y $\sum_{m=1}^{n=1} m$ pasadas por el bucle **para** interno. Por tanto, la complejidad es $O(n^2)$.

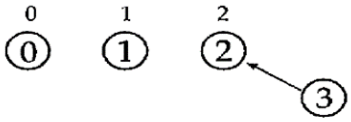
Una versión más rápida del algoritmo: se supones que la etiqueta del conjunto producido por una operación fusionar es necesariamente la etiqueta de uno de los conjuntos que hayan sido fusionados. La planificación producida en primer lugar puede contener huecos, el algoritmo acaba por trasladar tareas hacia adelante para llenarlos:

```
función secuencia2 (d[0..n]): k,matriz[1..k]
    matriz j,F[0..n]
    p=min(n,max{d[i]|1≤i≤n})
    para i:=0 hasta p hacer
        j[i]:=0
        F[i]:=1
        iniciar el conjunto {i}
    {bucle voraz}
    para i:=1 hasta n hacer {orden decreciente de g}
        k:=buscar(min(p,d[i]))
        m:=F[k]
        si m≠0 entonces
            j[m]:=i
            l:=buscar(m-1)
            F[k]:=F[l]
            fusionar(k,l) {el conjunto resultante tiene la etiqueta k o l}
    k:=0
    para i:=1 hasta p hacer
        si j[i]>0 entonces
            k:=k+1
            j[k]:=j[i]
    devolver k,j[1..k]
```

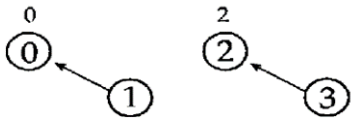
Si se nos da el problema con las tareas ya ordenadas por beneficios decrecientes, tal que es posible obtener una secuencia optima llamando al algoritmo anterior, entonces la mayor parte del conjunto se invertirá en manipular conjuntos disjuntos. Hay que ejecutar como máximo 2n operaciones buscar y n operaciones fusionar, luego el tiempo requerido esta en $O(n\alpha(2n,n))$ donde α es la función de crecimiento lento. Si las tareas están en un orden arbitrario, primero tendremos que ordenarlas, y la obtención de la secuencia inicial requiere un tiempo O(nlogn). Volviendo al ejemplo de las seis tareas:

Iniciación: p=min(6,max(d_i))=3

Intento 1: d_1=3, se asigna la tarea 1 a la posición 3

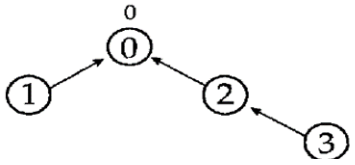


Intento 2: d_2=1, se asigna la tarea 2 a la posición 1



Intento 3: d_3=1, no hay posiciones libres disponibles porque el valor de F=0

Intento 4: d_4=3, se asigna la tarea 4 a la posición 2



Intento 5: d_5=1, no hay posiciones libres disponibles

Intento 6: d_6=3, no hay posiciones libres disponibles