

Resumen de programación 3

Tema 6. Algoritmos voraces.

Índice:

6.1. Dar la vuelta (1)	3
6.2. Características generales	4
6.3. Grafos: árboles de recubrimiento mínimo	6
6.3.1. Algoritmo de Kruskal	9
6.3.2. El algoritmo de Prim	14
6.4. Grafos: caminos mínimos	18
6.5. El problema de la mochila (1)	24
6.6. Planificación	28
6.6.1. Minimización del tiempo en el sistema	28
6.6.2. Planificación con plazo fijo	30

Bibliografía:

Se han tomado apuntes de los libros:

- *Fundamentos de algoritmia*. G. Brassard y P. Bratley

Empezaremos a ver los algoritmos voraces, ya que son los más fáciles de ver. Resultan **fáciles de inventar e implementar** y cuando funcionan son **muy eficientes**. Sin embargo, hay muchos problemas que no se pueden resolver usando el enfoque voraz.

Los algoritmos voraces se utilizan típicamente para resolver problemas de optimización. Por ejemplo, la búsqueda de la recta más corta para ir desde un nodo a otro a través de una red de trabajo o la búsqueda del mejor orden para ejecutar un conjunto de tareas en una computadora.

Un algoritmo voraz **nunca reconsidera su decisión**, sea cual fuere la situación que pudiera surgir más adelante.

Veremos en el siguiente apartado un ejemplo cotidiano para la que esta táctica funciona bien. En los siguientes apartados, se seguirán los apartados del temario, en los que ha sido imposible resumir las demostraciones, sobre todo, por lo que realmente este tema es una copia casi exacta del libro. Nuestro convenio es ver primero el funcionamiento del algoritmo, luego ejemplos (uno o varios), demostración de optimalidad y, por último, costes del algoritmo.

6.1. Dar la vuelta (1)

Se nos dan estas monedas: 100, 25, 10, 5 y 1 pts. Nuestro problema consiste en diseñar un algoritmo para pagar una cierta cantidad a un cliente, utilizando el menor número posible de monedas. Por ejemplo, si tenemos que pagar 289 pts., habría que usar estas monedas: 2 de 100, 3 de 25, 1 de 10 y 4 de 1 pts.

Usamos de modo inconsciente un algoritmo voraz: empezaremos por nada y en cada fase vamos añadiendo a las monedas que ya están seleccionadas una moneda de la mayor denominación posible, pero que no deben llevarnos más allá de la cantidad que haya que pagar.

El algoritmo formalizado es:

```
funcion 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 elemento de C tal que s + x ≤ n
    si no existe ese elemento entonces
        Devolver “no encuentro la solución”
    S ← S ∪ {una moneda de valor x};
    s ← s + x;
devolver S
```

Es fácil convencerse (aunque difícil de probar formalmente) que este algoritmo siempre produce una solución óptima para nuestro problema. En algunos casos, puede seleccionar un conjunto de monedas que no sea óptimo (más monedas que las necesarias), mientras que en otros casos no llegue a encontrar solución aún cuando exista, por lo que el algoritmo voraz no funciona adecuadamente.

El algoritmo es “voraz”, porque en cada paso selecciona la mayor de las monedas que puede encontrar, sin preocuparse por lo correcto de la decisión.

Además, nunca cambia de opinión. Una vez que una moneda se ha incluido en la solución, la moneda se queda allí para siempre.

6.2. Características generales

Generalmente, los algoritmos voraces y los problemas que éstos resuelven se caracterizan por la mayoría de propiedades siguientes:

- Son adecuadas para **problemas de optimización**, tal y como vimos en el ejemplo anterior.
- Para construir la solución de nuestro problema disponemos de un **conjunto (o lista) de candidatos**. Por ejemplo, para el caso de las monedas, los candidatos son las monedas disponibles, para construir una ruta los candidatos son las aristas de un grafo, etc. A medida que avanza el algoritmo tendremos estos conjuntos:
 - Candidatos considerados y seleccionados.
 - Candidatos considerados y rechazados.

Las funciones empleadas más destacadas de este esquema son:

1. **Función de solución:** Comprueba si un cierto conjunto de candidatos constituye una solución de nuestro problema, ignorando si es o no óptima por el momento. Puede que exista o no solución.
2. **Función factible:** Comprueba si el candidato es compatible con la solución parcial construida hasta el momento; esto es, si existe una solución incluyendo dicha solución parcial y el citado candidato.
3. **Función de selección:** Indica en cualquier momento cuál es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados. Es la más importante de todas.
4. **Función objetivo:** Da el valor de la solución que hemos hallado: el número de monedas utilizadas para dar la vuelta, la longitud de la ruta calculada, etc. Esta función no aparece explícitamente en el algoritmo voraz.

Para resolver nuestro problema, buscamos un conjunto de candidatos que constituyan una solución y que optimice (maximice o minimice, según los casos) el valor de la función objetivo. Los algoritmos voraces avanzan paso a paso:

- Inicialmente, el conjunto de elementos seleccionados está vacío y el de solución también lo está.
- En cada paso, se considera añadir a este conjunto el mejor candidato sin considerar los restantes, estando guiada nuestra selección por la función de selección. Se nos darán estos casos:
 1. Si el conjunto ampliado de candidatos seleccionados ya no fuera factible (no podemos completar el conjunto de solución parcial dado por el momento), **rechazamos** el candidato considerado por el momento y no lo volvemos a considerar.
 2. Si el conjunto aumentado sigue siendo factible, entonces **añadimos** el candidato actual al conjunto de candidatos seleccionados. Cada vez que se amplía el conjunto de candidatos seleccionados comprobamos si este constituye una solución para nuestro problema. Se quedará en ese conjunto para siempre.

Cuando el algoritmo voraz funciona correctamente, la primera solución que se encuentra es la óptima.

El **esquema voraz** es el siguiente:

```
funcion voraz (C: Conjunto): conjunto
{ C es el conjunto de candidatos }
 $S \leftarrow \emptyset$       { Construimos la solución en el conjunto S }
mientras  $C \neq \emptyset$  y  $\neg$ solución (S) hacer
     $x \leftarrow \text{seleccionar}(C)$ 
     $C \leftarrow C \setminus \{x\}$ ;
    si factible ( $S \cup \{x\}$ ) entonces  $S \leftarrow S \cup \{x\}$ 
si solución (S) entonces devolver S
si no devolver “no hay solución”
```

La función de selección suele estar relacionada con la función *objetivo*. Por ejemplo, si estamos intentando maximizar nuestros beneficios, es probable que seleccionemos aquel candidato restante que posea mayor valor individual. En ocasiones, puede haber varias funciones de selección plausibles.

Veremos una forma de adecuar las **características generales** de los algoritmos voraces a las particulares del problema de devolver cambio, aunque previamente hemos visto el esquema de este problema en particular:

- Los **candidatos** son un conjunto de monedas. Por ejemplo, 100, 25, 10, 5 y 1 pts.
- La **función de solución** comprueba si el valor de las monedas seleccionadas es exactamente el valor que hay que pagar.
- Un conjunto de monedas será **factible** si su valor no sobrepasa la cantidad que haya que pagar.
- La **función de selección** toma la moneda de valor más alto que quede en el conjunto de candidatos.
- La **función objetivo** cuenta el número de monedas utilizadas en la solución.

Está claro que es más eficiente rechazar todas las monedas restantes de 100 pts. (por ejemplo) cuando el valor restante que haya que pagar caiga por debajo de ese valor. El uso de la división entera para calcular cuántas monedas de un cierto valor hay que tomar también es más eficiente que actuar por sustracciones sucesivas.

6.3. Grafos: árboles de recubrimiento mínimo

Es el primer tipo de problemas que veremos que se pueden resolver con un algoritmo voraz.

Sea $G = \langle N, A \rangle$ un grafo conexo no dirigido en donde N es el conjunto de nodos y A el de aristas. Cada arista posee una longitud negativa. El problema consiste en hallar un subconjunto T de las aristas de G , tal que utilizando sólo las aristas de T , todos los nodos deben quedar conectados y además la suma de las longitudes de las aristas de T debe ser tan pequeña como sea posible (forman **árbol de recubrimiento mínimo**). Existirá, al menos, una solución y si hubiera dos soluciones de igual longitud total, escogemos la de menor número de aristas.

En lugar de hablar de longitudes, podemos asociar un *coste* a cada arista. Entonces, el problema consistirá en hallar un subconjunto T de las aristas cuyo coste total sea el menos posible.

Sea $G' = \langle N, T \rangle$ el grafo parcial formado por todos los nodos de G y las aristas de T ($T \subseteq A$: Contenido en A o incluso igual a A) y supongamos que en N hay n nodos. Un grafo conexo con n nodos debe tener, al menos $n - 1$ aristas como mínimo.

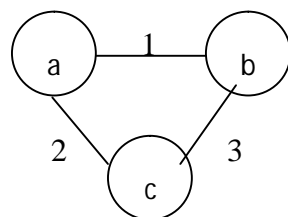
Por tanto, si G' es conexo y T tiene más de $n - 1$ aristas (al menos tiene un ciclo), se puede eliminar al menos una arista sin desconectar G' siempre y cuando seleccionemos una arista que forme parte de un ciclo, dándose estos **casos**:

1. Disminuye la longitud total de las aristas de T .
2. La longitud total queda intacta a la vez que disminuye el número de aristas en T .

El grafo G' se denomina árbol de recubrimiento mínimo para el grafo G si tiene $n - 1$ aristas tales que G' es conexo y el coste global de las aristas que quedan es el mínimo posible.

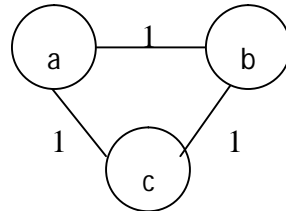
Ejemplo: Supongamos que los nodos de G representan unidades y sea el coste de una arista $\{a, b\}$ el de tender una línea telefónica desde a hasta b . Entonces, un árbol de recubrimiento mínimo de G se corresponde con la red más barata posible para dar servicio a todas las ciudades en cuestión, siempre y cuando sólo se pueda utilizar conexiones directas entre ciudades.

Esto es un añadido del autor. Puede ser que el algoritmo voraz no de una única solución, puede ser que haya más, como **ejemplos** podremos poner los siguientes. Es importante, ya que suele caer preguntas similares en exámenes:



El coste de llegar de b a c es similar en ambos caminos (b-a y a-c) y directo. Se quitaría la arista de coste 3, por ser la mayor de todas. Con esto además, se demuestra que puede haber dos árboles de recubrimiento mínimo distintos en el mismo grafo, porque forman un ciclo.

El segundo **ejemplo** podría ser:



En este caso, podremos quitar cualquier arista y llegar a todos los nodos con el mismo coste. Igualmente, hay varios posibles árboles de recubrimiento mínimo.

Podremos tener dos tácticas para resolver el problema que posteriormente veremos con más detalle:

1ª táctica: Consiste en comenzar por un conjunto vacío T y seleccionar en cada etapa la arista más corta que todavía no haya sido seleccionada o rechazada, independientemente de donde se encuentra esa arista en G .

2ª táctica: Implica seleccionar un nodo y construir un árbol a partir de él, seleccionando en cada etapa la arista más corta posible que pueda extender el árbol hasta un nodo adicional.

Las **componentes** para el problema del recubrimiento mínimo serán:

- Los **candidatos** son las aristas de G .
- Un conjunto de aristas es una **solución** si constituye un árbol de recubrimiento para los nodos de N (no tiene que ser recubrimiento mínimo).
- Un conjunto de aristas es **factible** si no contiene ningún ciclo.
- La **función de selección** que utilizamos varía con el algoritmo (según las tácticas antes expuestas).
- La **función objetivo** que hay que minimizar es la longitud total de las aristas de la solución.

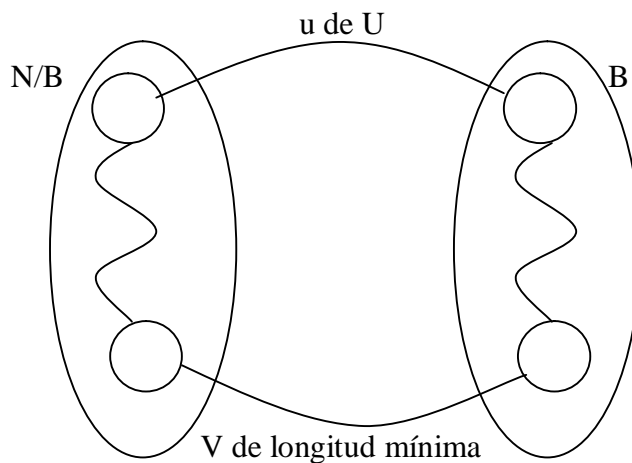
El **lema** siguiente es crucial para demostrar la corrección de los próximos algoritmos:

Lema 6.3.1 Sea $G = \langle N, A \rangle$ un grafo conexo no dirigido en el cual está dado la longitud de todas las aristas. Sea $B \subset N$ un subconjunto estricto de los nodos de G . Sea $T \subseteq A$ un conjunto prometedor de aristas, tal que no haya ninguna arista de T que salga de B . Sea v la arista más corta que sale de B (o una de las más cortas si hay empates). Entonces $T \cup \{v\}$ es prometedor.

Demostración: Sea U un árbol de recubrimiento mínimo de G tal que $T \subseteq U$. Este U tiene que existir puesto que T es prometedor por hipótesis. Se nos darán estos casos:

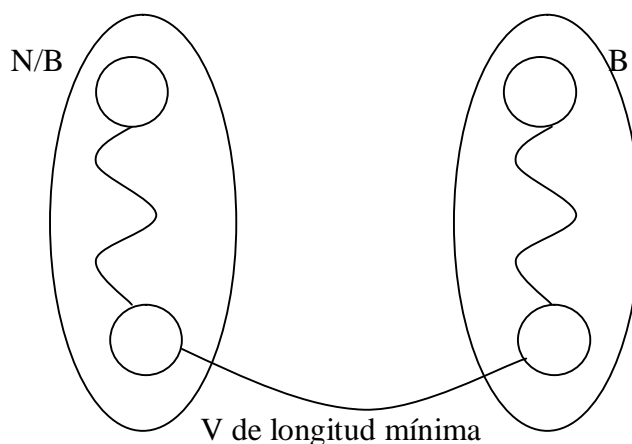
1. Si $v \in U$, entonces no hay nada que probar (es ya árbol de recubrimiento mínimo).
2. Si no, cuando añadimos v a U creamos exactamente un ciclo, en el que debe existir otra arista llamada u , ya que sería quien cerrara dicho ciclo.

La situación hasta el momento es la siguiente (es una interpretación de un alumno):



Si ahora eliminamos u , el ciclo desaparece y obtenemos un *nuevo árbol* V que abarca G (recordemos que era un grafo conexo no dirigido en el cual está dada la longitud de todas las aristas). Sin embargo, la longitud de v , por definición, no es mayor que la de u (v tiene longitud mínima) y, por tanto, la longitud total de las aristas de U . Por tanto, V es también un árbol de recubrimiento mínimo de G y contiene a v .

De nuevo, la situación sería la siguiente:



Para completar la demostración sólo hay que comentar que $T \subseteq V$ porque la arista u que se ha eliminado sale de B y, por tanto, no podría haber sido una arista de T (recordemos que es el conjunto promotor de aristas, tal que no haya arista de T que sale de B).

6.3.1. Algoritmo de Kruskal.

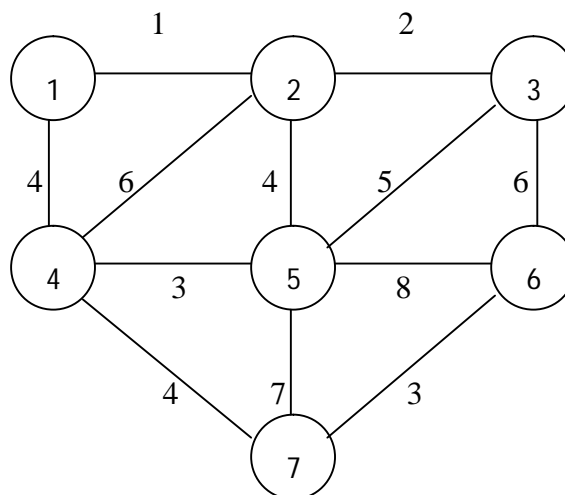
Para este tipo de problema se nos darán estos conjuntos:

- **T**: Conjunto de *aristas* seleccionadas.

Este algoritmo hará lo siguiente:

- El conjunto de aristas T está inicialmente vacío.
- A medida que progresa el algoritmo, se añaden aristas a T , que forman componentes conexas separadas.
- Para construir componentes conexas más y más grandes, examinaremos las aristas de G por orden creciente de longitud. Se nos dan dos casos:
 1. Si una arista une a dos componentes distintas, se la **añadimos** a T . Consiguientemente, las dos componentes conexas forman ahora una única componente.
 2. En caso contrario, se **rechaza** la arista: une a dos nodos de la misma componente conexa y, por tanto, no se puede añadir a T sin formar un ciclo.
- El algoritmo se detiene cuando sólo queda una componente conexa.

Ejemplo del algoritmo de Kruskal:



Seguiremos estos pasos para resolverlo:

1^{er} paso. Ordenamos aristas por orden creciente de costes:

Nodo	Coste
{1,2}	1
{2,3}	2
{4,5}	3
{6,7}	3
{1,4}	4
{2,5}	4
{4,7}	4
{3,5}	5
{2,4}	6
{3,6}	6
{5,7}	7
{5,6}	8

2º paso. Seleccionamos aristas de la lista por orden. Vemos si unen componentes conexas distintas, si es así fusionamos esas componentes conexas para unir las en una misma partición.

Paso	Arista seleccionada	Componentes conexas
Inicialización	-	{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}	Rechazada, por formar ciclo. Están en el mismo conjunto.
7	{4,7}	{1,2,3,4,5,6,7}

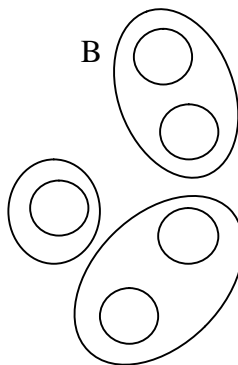
El coste del árbol de recubrimiento mínimo es 17.

En cada paso, el algoritmo de Kruskal va cogiendo aristas prometedoras (aquéllas que se extiende la solución a la óptima) hasta llegar a $n - 1$ prometedor como este ejemplo.

Teorema 6.3.2 El algoritmo de Kruskal halla un árbol de recubrimiento mínimo

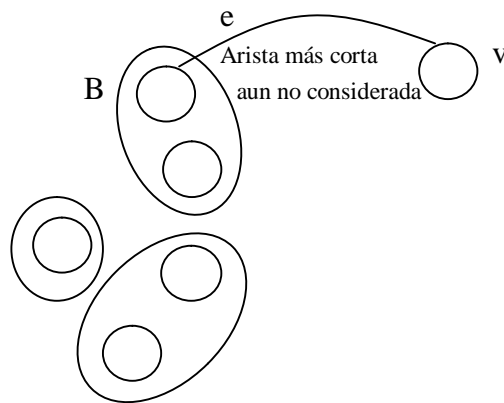
Demostración: se hace por inducción matemática sobre el número de aristas que hay en el conjunto T . Mostraremos que si T es prometedor, entonces sigue siendo prometedor en cualquier fase del algoritmo cuando se le añade una arista adicional. Cuando se detiene el algoritmo, T da una solución de nuestro problema; puesto que también es prometedora, la solución es **óptima**.

- Base: El conjunto vacío es prometedor porque G es conexo y, por tanto, tiene que existir una solución.
- Paso de inducción: Supongamos que T (recordemos que es el conjunto de aristas seleccionadas) es prometedor antes de que el algoritmo añada una nueva arista $e = \{u, v\}$. Las aristas de T dividen a los nodos de G en dos o más componentes conexas; el nodo u se encuentra en una de estas componentes y v está en otra componente conexa. Sea B el conjunto de nodos de esa componente que contiene a u . Ahora:
 - El conjunto de B es un subconjunto estricto de los nodos de G , puesto que no incluye a v , por ejemplo. Gráficamente, quedaría algo así (es una interpretación de un alumno), siendo estos conjuntos subconjuntos de G , que por problemas con el dibujo es imposible hacerlo. Por tanto, sería:



- T es un conjunto prometedor de aristas tal que ninguna arista de T sale de B , porque una arista de T tiene o bien ambas aristas en B , o ninguna arista en B , así que por definición, no sale de B .
- e es una de las aristas más cortas que salen de B , porque todas las aristas estrictamente más cortas ya se han examinado o bien se han añadido a T (conjunto de aristas seleccionadas) o bien se han rechazado porque tenían los dos extremos en la misma componente conexa.

Otra apreciación del alumno sería que en este punto, quedaría algo así:



Recordemos que por ser Kruskal cogería la arista con el coste menor, que sería e .

Concluimos que el conjunto $T \cup \{e\}$ también es prometedor.

Para implementar el algoritmo, es preciso efectuar rápidamente las operaciones *buscar* (x), que nos dice en qué componente conexa se encuentra el nodo x y *fusionar* (A, B) para fusionar dos componentes conexos. Por eso, utilizamos la estructura de partición.

Para el algoritmo es necesario representar el grafo como un vector de aristas con sus longitudes asociadas. El algoritmo es:

```

funcion Kruskal ( $G = \langle N, A \rangle$ : grafo, longitud:  $A \rightarrow R^+$ ): conj. aristas
{ Iniciación }
Ordenar  $A$  por longitudes crecientes
 $n \leftarrow$  el número de nodos que hay en  $N$ 
 $T \leftarrow \emptyset$  { Contendrá las aristas del árbol de recubrim. mínimo }
Iniciar  $n$  conjuntos cada uno de los cuales contiene un elemento
distinto de  $N$ 
{ Bucle voraz }
repetir
     $e \leftarrow \{u, v\} \leftarrow$  arista más corta aun no considerada
    compu  $\leftarrow$  buscar ( $u$ )
    compv  $\leftarrow$  buscar ( $v$ )
    si compu  $\neq$  compv entonces
        fusionar (compu, compv)
     $T \leftarrow T \cup \{e\}$ ;
hasta que  $T$  contenga  $n - 1$  aristas
devolver  $T$ 
    
```

Coste del algoritmo: Calculamos el tiempo de ejecución del algoritmo en la forma siguiente. El número de operaciones es:

- $\theta(a * \log(a))$: Coste para ordenar las aristas.
- $\theta(n)$: Para iniciar los n conjuntos disjuntos.
- $\theta(2 * a * \alpha(2 * a, n))$: Para las operaciones de fusionar (tendrá como máximo $2 * a$ operaciones buscar y $n - 1$ operaciones fusionar).
- $O(a)$: Para el caso peor de las operaciones restantes.

Vamos a detallar más el coste de ordenar las aristas, que es $\theta(a * \log(a))$. Veremos la **parte logarítmica**, teniendo en cuenta que el número de aristas seguirá esta fórmula: $n - 1 \leq a \leq \frac{n*(n-1)}{2}$, por eso se nos darán estos casos:

- El grafo es disperso, es decir, $a \approx n$, que corresponde con la parte izquierda de la fórmula. Por tanto,

$$\log(a) \approx \log(n)$$

- El grafo es denso, por lo que $a \approx n^2$, que será más cercano a la parte derecha de la fórmula. Por tanto,

$$\log(a) \approx \log(n^2) \approx \log(n).$$

Por la propiedad de los logaritmos, $\log(n^2) = 2 * \log(n)$.

En conclusión, asintóticamente, tanto si el grafo es disperso como denso es $\theta(a * \log(n))$.

En cuanto a la **parte multiplicativa**, razonaremos de la misma manera:

- El grafo es disperso, por lo que el coste es $\theta(n * \log(n))$.
- El grafo es denso, por lo que el coste es $\theta(n^2 * \log(n))$.

Una **mejora del algoritmo**, que es bastante importante el saberla es usar un montículo invertido, en el que la raíz es el mínimo elemento. Se nos darán estos costes:

- o Crear un montículo: $\theta(a)$
- o Restaurar la condición del montículo (hundir la raíz como vimos en el tema de estructura de datos): $\theta(\log(a)) \approx \theta(\log(n))$.

Al repetirse un número de a veces la restauración de la condición del montículo, en el **caso peor** será: $\theta(a * \log(n))$.

En el **caso mejor**, será coste lineal $\theta(n)$, ya que tendríamos ordenadas las aristas de menor a mayor (es otra apreciación del alumno, no aparece en el libro, ni en otros libros que he consultado).

6.3.2. El algoritmo de Prim.

Recordemos que en el algoritmo de Kruskal se toman las aristas por orden creciente sin preocuparse por su conexión y teniendo cuidado de no crear ciclos. El bosque crecerá al azar hasta formar un árbol de recubrimiento mínimo con todos los componentes conexos.

En el **algoritmo de Prim**, por otra parte, el árbol de recubrimiento mínimo crece de forma natural, comenzando por una raíz arbitraria. En cada fase, se añade una nueva rama al árbol ya construido. El algoritmo se detiene cuando se han alcanzado todos los nodos.

Veremos con más detalle el algoritmo. Se nos dan estos conjuntos:

B: Conjunto de nodos que contiene la solución (inicialmente contiene al nodo de partida).

T: Conjunto de aristas solución (inicialmente vacío).

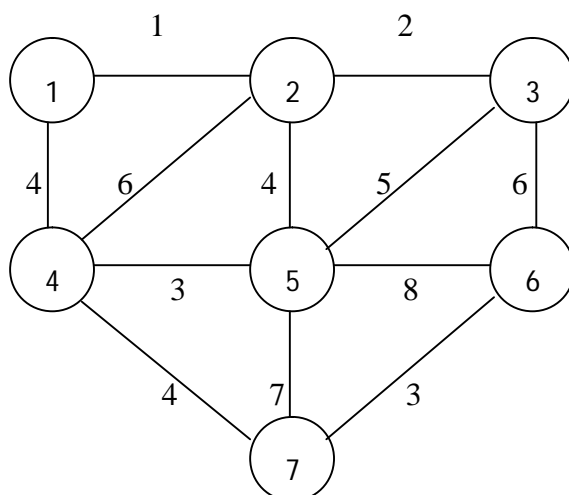
El algoritmo hará lo siguiente:

- Inicialmente, B contiene un único nodo arbitrario y T está vacío.
- En cada paso, el algoritmo de Prim busca la arista más corta posible $\{u, v\}$, tal que $u \in B$ y $v \in N/B$. Entonces añade v a B y $\{u, v\}$ a T. De esta manera, las aristas de T forman en todo momento un árbol de recubrimiento mínimo para los nodos de B. Continuamos mientras $B \neq N$.

Un **enunciado informal** del algoritmo es:

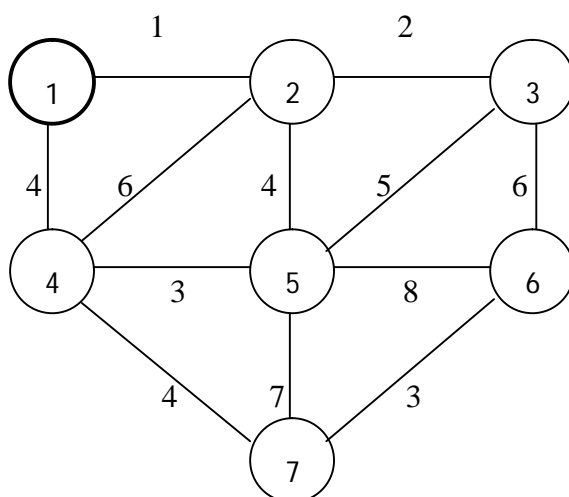
```
funcion prim ( $G = \langle N, A \rangle$ : grafo, longitud:  $A \rightarrow R^+$ ): conj. aristas
{ Iniciación }
 $T \leftarrow \emptyset$ ;
 $B \leftarrow \{ \text{un miembro arbitrario de } N \}$ 
  mientras  $B \neq N$  hacer
    buscar  $e = \{u, v\}$  de longitud mínima tal que
       $u \in B$  y  $v \in N/B$ 
       $T \leftarrow T \cup \{e\}$ ;
       $B \leftarrow B \cup \{v\}$ ;
devolver T
```

Ejemplo de algoritmo de Prim:



Lo resolveremos realizando estos pasos:

1^{er} paso: seleccionamos un nodo como raíz arbitraria. En este caso, es el nodo 1.



2^o paso: En cada paso, el algoritmo de Prim busca la arista más corta posible $\{u, v\}$ tal que $u \in B$ y $v \in N/B$. Entonces añade v a B y $\{u, v\}$ a T .

En nuestro ejemplo tenemos:

Paso	Arista seleccionada	B
Inicialización	-	{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	{6,7}	{1,2,3,4,5,6,7}

La longitud (o el coste) total es 17, como en el caso de Kruskal.

Cuando haya soluciones distintas, ante empate de costes podemos escoger el que queramos.

No es habitual escoger los nodos en orden, pero como en este caso se puede dar. Vemos que hasta el paso 4 se añaden en orden, dependerá en todo caso de los costes de las aristas.

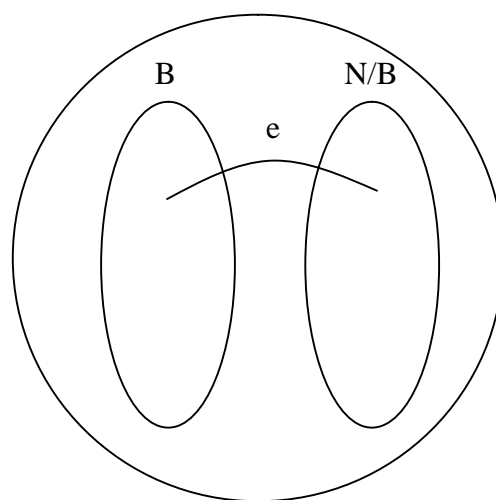
Veremos la demostración del algoritmo de Prim aunque es parecida a la de Kruskal.

Teorema 6.3.3 El algoritmo de Prim halla un árbol de recubrimiento mínimo.

Demostración: Es por inducción matemática sobre el número de aristas que hay en el conjunto T (conjunto de aristas). Demostraremos que si T es *prometedor* en alguna fase del algoritmo entonces al añadir una arista adicional sigue siendo *prometedor*. Cuando se detiene el algoritmo, T da una solución para nuestro problema, puesto que también es *prometedor* la solución es óptima.

- Base: El conjunto vacío es prometedor.
- Paso de inducción: Suponga que T es prometedor inmediatamente antes de que el algoritmo añada una nueva arista $e = \{u, v\}$. Ahora:
 - B es un subconjunto estricto de N .
 - T es un conjunto de aristas prometedor, por hipótesis de inducción.
 - e es por definición una de las aristas más cortas que salen de B .

La situación será la siguiente. Hay que destacar que aunque sea el dibujo algo distinto al que vimos en Kruskal, resaltamos que la idea es similar y la demostración igualmente. Quedaría:



siendo:

T : Contiene las aristas seleccionadas.

G : Grafo completo (conjunto de nodos)

B : Conjunto de esos componentes que contiene a u (conjunto de nodos).

La **segunda implementación**, que es más sencilla será:

```

funcion prim (L[1..n,1..n]): conj. aristas
{ Iniciación: solo el nodo 1 se encuentra en B }
 $T \leftarrow \emptyset$ ; { Contendrá las aristas del árbol de recubrim. mínimo }
para  $i \leftarrow 2$  hasta  $n$  hacer
    mas próximo  $[i] \leftarrow 1$ 
    distmin  $[i] \leftarrow L[i, 1]$ 
{ Bucle voraz }
repetir  $n - 1$  veces
    min  $\leftarrow \infty$ 
    para  $j \leftarrow 2$  hasta  $n$  hacer
        si  $0 \leq \text{distmin}[j] < \text{min}$  entonces min  $\leftarrow \text{distmin}[j]$ 
                                      $k \leftarrow j$ 

     $T \leftarrow T \cup \{ \text{mas próximo } [k], k \}$ 
    distmin  $[k] \leftarrow -1$ 
    para  $j \leftarrow 2$  hasta  $n$  hacer
        si  $L[j, k] \leq \text{distmin}[j]$  entonces distmin  $[k] \leftarrow L[j, k]$ 
                                     mas próximo  $[j] \leftarrow k$ 

devolver T

```

Supongamos que los nodos de G que están numeradas de 1 a n , así que $N = \{1, 2, \dots, n\}$, siendo:

L: Matriz simétrica que da la longitud de todas las aristas, con $L[i, j] = \infty$, si no existe la arista correspondiente.

mas próximo [i]: Proporciona el nodo de B que está más próximo a i .

distmin [i]: Da la distancia desde i hasta el nodo más próximo. Si $\text{distmin}[i] = -1$, sabremos si un nodo está o no en B .

mas próximo[1] y distmin[1] no se utilizan, por ser el nodo inicial el 1.

Análisis del coste del algoritmo:

- El bucle “para”, que es interno, requiere un tiempo $\theta(n)$.
- El bucle “repetir”, que es exterior, se repite $n - 1$ veces, por lo que requiere $\theta(n)$.

El coste del algoritmo de Prim requiere un tiempo que está en $\theta(n^2)$.

Comparación de costes de Prim y Kruskal:

	Prim	Kruskal
General	$\theta(n^2)$	$\theta(a * \log(n))$
Grafo denso ($a \approx n^2$)	$\theta(n^2)$	$\theta(n^2 * \log(n))$
Grafo disperso ($a \approx n$)	$\theta(n^2)$	$\theta(n * \log(n))$

Como hemos visto anteriormente, para un grafo denso, a tiende a $\frac{n*(n-1)}{2}$, el algoritmo de Kruskal requiere un tiempo de $\theta(n^2 * \log(n))$, por lo que es más eficiente el algoritmo de Prim.

Para un grafo disperso, a tiende a n , por lo que el algoritmo de Kruskal mejora.

Mejora del algoritmo: Si usamos montículos invertidos, tendremos que al igual que pasaba antes el coste es $\theta(a * \log(n))$. Si el grafo es denso o disperso se ve como antes.

6.4. Grafos: caminos mínimos

Es el segundo tipo de problemas que veremos.

Considérese ahora un grafo dirigido $G = \langle N, A \rangle$ en donde N es el conjunto de nodos de G y A es el de aristas dirigidas. Podría ser el caso de aristas no dirigidas considerándose el grafo siguiente:



Cada arista posee una longitud *no* negativa. Se toma uno de los nodos como nodo origen. El problema consiste en determinar la longitud del camino mínimo que va desde el origen hasta cada uno de los demás nodos del grafo.

Este problema se puede resolver mediante un algoritmo voraz que recibe el nombre de algoritmo de Dijkstra. Emplea estos conjuntos:

- **S:** Contiene aquellos nodos que ya han sido seleccionados, cuya distancia es conocida para todos los nodos de este conjunto.
- **C:** Contiene todos los demás nodos, cuya distancia mínima desde el origen todavía no es conocida y que son candidatos a ser seleccionados posteriormente.

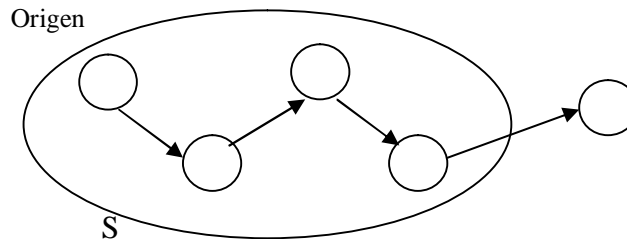
La unión de ambos conjuntos sería $N = S \cup C$, o lo que es igual, ambos conjuntos formarán el conjunto total.

El **funcionamiento** del algoritmo es el siguiente:

- En un primer momento, S contiene nada más que el origen; cuando se detiene el algoritmo, S contiene todos los nodos del grafo y el problema está resuelto.
- En cada paso, seleccionamos aquel nodo de C cuya distancia al origen sea mínima y se lo añadimos a S .

Camino especial: Diremos que un camino desde el origen a algún otro nodo es especial si todos los nodos intermedios a lo largo del camino pertenecen a S.

Gráficamente, sería:



En cada fase del algoritmo, hay una matriz D que contiene la longitud del camino especial más corto que va hasta cada nodo del grafo. En el momento en que se desee añadir un nuevo nodo v a S , el camino especial más corto hasta v es también el más corto de los caminos posibles hasta v (minimiza D). Al acabar el algoritmo, todos los caminos desde el origen hasta algún nodo son especiales. Consiguientemente, los valores que hay en D dan la solución del problema de caminos mínimos.

Suponemos una vez más que los nodos de G están numerados de 1 a n , por tanto, $N = \{1, 2, \dots, n\}$. Podemos suponer que el nodo uno es el origen. Supongamos que la matriz L da la longitud de todas las aristas dirigidas: $L[i, j] \geq 0$ si la arista $(i, j) \in A$ y $L[i, j] = \infty$, en caso contrario.

Con estos datos, tendremos este algoritmo:

```

funcion Dijkstra (L[1..n, 1..n]): matriz [2..n]
    matriz D[2..n]
    { Iniciación }
     $C \leftarrow \{2, 3, \dots, n\}$       {  $S = N/C$  sólo existe implícitamente }
    para  $i \leftarrow 2$  hasta  $n$  hacer  $D[i] \leftarrow L[1, i]$ 
    { Bucle voraz }
    repetir  $n-2$  veces
         $v \leftarrow$  algún elemento de  $C$  que minimiza  $D[v]$ 
         $C \leftarrow C \setminus \{v\}$       { e implícitamente  $S \leftarrow S \cup \{v\}$  }
        para cada  $w \in C$  hacer
             $D[w] \leftarrow \min(D[w], D[v] + L[v, w]);$ 
    devolver D
    
```

siendo:

$D[i]$: Vector de distancias, que indica la distancia hasta el nodo i .

$L[i, j]$: Matriz de longitud, que indica longitud del nodo i y al j .

w : Arista del conjunto C , que contiene todos los demás nodos, cuya distancia mínima desde el origen todavía no es conocida y que son candidatos a ser seleccionados posteriormente.

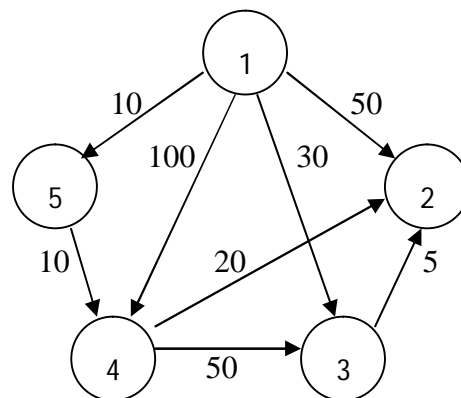
El bucle principal se repite $n - 2$ veces, ya que D no cambiaría si hiciéramos una iteración más para eliminar el último elemento de C . Añadimos una segunda matriz $P[2..n]$ llamada **matriz de precedencias o vector de predecesores**, en donde $P[v]$ contiene el número del nodo que precede a v dentro del camino más corto. Para hallarlo se tendría que seguir los punteros hacia atrás desde el destino hacia el origen.

Por tanto, el algoritmo definitivo incluyendo esta matriz de precedencia o vector de predecesores (según se guste más, en algunos exámenes he encontrado la segunda acepción, pero es igual) quedaría así:

```

funcion Dijkstra (L[1..n, 1..n]): matriz [2..n]
    matriz D[2..n]
    matriz P[2..n]
    { Iniciación }
     $C \leftarrow \{2, 3, \dots, n\}$       {  $S = N/C$  sólo existe implícitamente }
    para  $i \leftarrow 2$  hasta  $n$  hacer  $D[i] \leftarrow L[1, i]$ 
    para  $i \leftarrow 2$  hasta  $n$  hacer  $P[i] \leftarrow 1$ 
    { Bucle voraz }
    repetir  $n - 2$  veces
         $v \leftarrow$  algún elemento de  $C$  que minimiza  $D[v]$ 
         $C \leftarrow C \setminus \{v\}$       { e implícitamente  $S \leftarrow S \cup \{v\}$  }
        para cada  $w \in C$  hacer
            si  $D[w] > D[v] + L[v, w]$  entonces
                 $D[w] \leftarrow D[v] + L[v, w];$ 
                 $P[w] \leftarrow v;$ 
    devolver D
  
```

Un **ejemplo** del algoritmo de Dijkstra será el siguiente:



El nodo de partida es el 1. Será importante identificarlo bien, ya que para resolverlo no es lo mismo empezar por el nodo 3 que por el 1 o cualquier otro nodo distinto. No saldrá la misma solución.

Tendremos estos pasos para resolverlo este ejemplo. Añadimos el vector de precedencias y a continuación los explicamos:

Paso	v	C	D	P
Inicialización	-	{2,3,4,5}	² ³ ⁴ ⁵ [50,30,100,10]	[1,1,1,1]
1	5	{2,3,4}	[50,30,20,10]	[1,1,5,1]
2	4	{2,3}	[40,30,20,10]	[4,1,5,1]
3	3	{2}	[35,30,20,10]	[3,1,5,1]

Inicialización: El conjunto de candidatos C es de 4 nodos, sin incluir el origen, que es el nodo 1. Ponemos todos los costes del nodo 1 al resto de nodos. Si no hubiera conexión directa la distancia será ∞ .

1^{er} paso: Se modifica el valor para llegar a 4 a través de 5.

2^o paso: Modificamos el valor de 2 a través de 4.

3^{er} paso y último: Modificamos el valor de 2, ahora a través de 3.

NOTA: Es fundamental decir que se parará en este último paso, aunque quede un candidato por escoger, ya que tenemos todos los caminos mínimos posibles.

La demostración que el algoritmo funciona es por **inducción matemática**.

Teorema 6.4.1 El algoritmo de Dijkstra halla los caminos mínimos desde un único origen hasta los demás nodos del grafo.

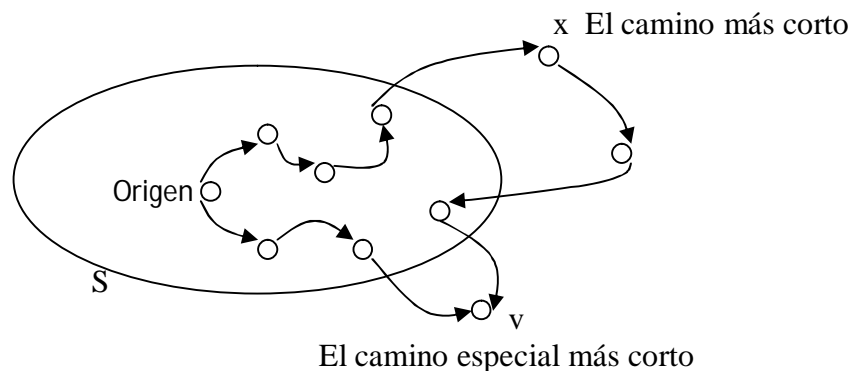
Demostración: Demostraremos por inducción matemática que:

- a) Si un nodo $i \neq 1$ está en S, entonces $D[i]$ da la longitud del camino más corto desde el origen hasta i, y
- b) Si un nodo y no está en S, entonces $D[i]$ da la longitud del camino *especial* más corto desde el origen hasta i.
- Base: Inicialmente, sólo el nodo 1, que es el origen, se encuentra en S, así que la situación a) es cierta sin más demostración. Para los demás nodos, el único camino especial desde el origen es el camino directo y D recibe valores iniciales en consecuencia. Por tanto, la situación b) es también cierta cuando comienza el algoritmo.
- Hipótesis de inducción: La hipótesis de inducción es que tanto la situación a) como la b) son válidas inmediatamente antes de añadir un nodo v a S (conjunto de nodos seleccionados). Detallamos los pasos de inducción por separado para ambas situaciones.
- Paso de inducción para la situación a): Para todo nodo que ya esté en S antes de añadir v no cambia nada, así que la situación a) sigue siendo válida. En cuanto al nodo v, ahora pertenecerá a S. Antes de añadirlo a S, es preciso comprobar que $D[v]$ proporcione la longitud del camino más corto que va desde el origen hasta v. Por hipótesis de inducción, nos da ciertamente la longitud del camino más corto. Por tanto, hay que verificar

que el camino más corto desde el origen hasta v no pase por ninguno de los nodos que no pertenecen a S .

Supongamos lo contrario; supongamos que cuando se sigue el camino más corto desde el origen hasta v , se encuentran uno o más nodos (sin contar el propio v) que no pertenecen a S . Sea x el primer nodo encontrado con estas características. Ahora el segmento inicial de esa ruta, hasta llegar a x , es una ruta especial, así que la distancia hasta x es $D[x]$, por la parte b) de la hipótesis de inducción. Claramente, la distancia total hasta v a través de x no es más corta que este valor, porque las longitudes de las aristas son no negativas. Finalmente, $D[x]$ no es menor que $D[v]$, porque el algoritmo ha seleccionado a v antes que a x . Por tanto, la distancia total hasta v a través de x es como mínimo $D[v]$ y el camino a través de x no puede ser más corto que el camino especial que lleva hasta v .

Gráficamente, sería:



- Paso de inducción para la situación b): Considérese ahora un nodo w , distinto de v , que no se encuentre en S . Cuando v se añade a S , hay dos posibilidades para el camino especial más corto desde el origen hasta w :
 1. O bien no cambia.
 2. O bien ahora pasa a través de v .

En el **segundo caso**, sea x el último nodo de S visitado antes de llegar a w . La longitud de este camino es $D[x] + L[x, w]$. Parece a primera vista que para calcular el nuevo valor de $D[w]$ deberíamos comparar el valor anterior de $D[w]$ con $D[x] + L[x, w]$ para todo nodo x de S (incluyendo a v). Sin embargo, para todos los nodos x de S salvo v , esta comparación se ha hecho cuando se añadió x a S y $D[x]$ no ha variado desde entonces. Por tanto, el nuevo valor de $D[w]$ se puede calcular sencillamente comparando el valor anterior con $D[v] + L[v, w]$.

Puesto que el algoritmo hace esto explícitamente, asegura que la parte b) de la inducción siga siendo cierta también cuando se añade a S un nuevo nodo v .

Para completar la demostración de que el algoritmo funciona, obsérvese que cuando se detenga el algoritmo, todos los nodos menos uno estarán en S . En ese momento queda claro que el camino más costo desde el origen hasta el nodo restante es un camino especial.

Coste del algoritmo: Recordemos que hemos visto estas dos implementaciones, por el momento:

1. La primera teníamos dos vectores L y D.
2. Añadimos el vector de precedencia o matriz de predecesores P.

El coste de ambas implementaciones lo determinara el bucle “mientras”, en el que se irán seleccionando una arista cada vez. Tendremos lo siguiente:

$$(n - 1) + (n - 2) + \dots + 1 \approx \sum_{i=1}^n i \approx n^2$$

Por tanto, el coste del algoritmo es $\theta(n^2)$, ya que aunque se añada un vector más en la segunda implementación las operaciones son elementales, por lo que se quedaría igual.

Mejora del algoritmo: Usaremos un montículo invertido, que contiene un nodo para cada elemento v de C que minimiza $D[v]$ que se encuentra siempre en la raíz.

El coste como hemos visto en los algoritmos anteriores corresponderá con:

- Inicialización: $\theta(a)$
- Flotar o hundir la raíz n veces: $\theta(a * \log(n))$.

Igualmente, tendremos dos casos, según el tipo de grafo:

- Si es grafo disperso ($a \approx n$): $\theta(n * \log(n))$.
- Si es grafo denso ($a \approx n^2$): $\theta(n^2 * \log(n))$.

Se concluye, por tanto, que es más conveniente si fuera grafo disperso.

6.5. El problema de la mochila (1).

Nos dan n objetos y una mochila. Para $i = 1, 2, \dots, n$, el objeto i tiene un peso positivo w_i ($w_i > 0$) y un valor positivo v_i ($v_i > 0$). La mochila puede llevar un peso que no sobrepase W .

Podremos fraccionar los objetos (es muy importante): $0 \leq x_i \leq 1$, de manera que podamos decidir llevar solamente una fracción del objeto x_i .

Nuestro objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos transportados, respetando la limitación de la capacidad impuesta (sin sobrepasar W).

El problema, por tanto, se puede enunciar de la siguiente manera:

$$\boxed{\text{Maximizar } \sum_{i=1}^n x_i * v_i \text{ con la restricción } \sum_{i=1}^n x_i * w_i \leq W}$$

donde ($v_i > 0$), ($w_i > 0$) y $0 \leq x_i \leq 1$ para $1 \leq i \leq n$.

Utilizaremos un algoritmo voraz para resolverlo, para lo cual tendremos estas **componentes**:

- **Candidatos**: Son los propios objetos.
- **Solución**: Es un conjunto (x_1, x_2, \dots, x_n) que indica que fracción de cada objeto hay que incluir.
- La solución será **factible** cuando se respeten las restricciones indicadas antes.
- La **función objetivo** es el valor total de los objetos que están en la mochila.
- La **función de selección** es la que quedaría por ver. Tendremos tres posibles funciones de selección:
 - Seleccionar el *objeto más valioso*, argumentando que esto incrementa el valor de la carga más rápido posible.
 - Seleccionar el *objeto más pequeño restante*, basándonos en que de este modo la capacidad se agota de forma más lenta posible.
 - Seleccionar aquel *objeto cuyo valor por unidad de peso sea el mayor posible* (*mayor $\frac{v_i}{w_i}$*).

El **algoritmo** es:

```

funcion mochila (w[1..n], v[1..n], W): matriz [1..n]
{ Iniciación }
para i ← 1 hasta n hacer x[i] ← 0
peso ← 0
{ Bucle voraz }
mientras peso < W hacer
    i ← el mejor objeto restante
    { Si el objeto se puede incluir entero }
    si peso + w[i] ≤ W entonces x[i] ← 1
                                peso ← peso + w[i]
    { Si no se puede incluir entero, se fracciona }
    si no x[i] ←  $\frac{(W-peso)}{w[i]}$ 
    peso ← W
devolver x

```

Ejemplo de problema de la mochila: Se nos dan 5 objetos distintos, con estos pesos y valores. Además, el peso máximo será de 100. La tabla es la siguiente:

$$n = 5, W = 100$$

w (pesos)	10	20	30	40	50
v (valores)	20	30	66	40	60
$\frac{v}{w} \left(\frac{\text{valores}}{\text{pesos}} \right)$	2.0	1.5	2.2	1.0	1.2

Resolveremos este algoritmo usando las tres **funciones de selección** antes puestas. Nos quedará:

Seleccionar:	x_i					Valor
Max. v_i	0	0	1	0.5	1	146
Min. w_i	1	1	1	1	1	156
Max. $\frac{v_i}{w_i}$	1	1	1	0	0.8	164

Observamos que la solución óptima es la tercera, puesto que es el de mayor valor, que era lo que nos pedían en el problema. Se ve que las otras funciones de selección no son óptimas. Si quisiéramos probarlo lo haríamos mediante un **contraejemplo** (importante para exámenes, que lo suelen pedir).

Veremos el teorema que corrobora que esta última función de selección es óptima.

Teorema 6.5.1 Si se seleccionan los objetos por orden decreciente de $\frac{v_i}{w_i}$ entonces el algoritmo de la mochila encuentra una solución óptima.

Demostración: Supongamos que los objetos disponibles están ordenados por orden decreciente de coste por unidad de peso:

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \dots \geq \frac{v_n}{w_n}$$

Nuestro método para averiguarlo lo denominaremos **reducción de diferencias**.

Sea $X = (x_1, x_2, \dots, x_n)$ la solución hallada por el algoritmo voraz. Se nos darán estos casos:

- Si todos los x_i son iguales a 1, entonces esta solución es claramente óptima.
- En caso contrario, supongamos que j denota el menor índice tal que $x_j < 1$. Examinando la forma en que funciona el algoritmo, está claro que:

1. $x_i = 1$, cuando $i < j$
2. $x_i = 0$, cuando $i > j$

y que $\sum_{i=1}^n x_i * w_i = W$.

Tendremos que $V(X) = \sum_{i=1}^n x_i * v_i$ es el valor de la solución X .

Ahora, sea $Y = (y_1, y_2, \dots, y_n)$ cualquier solución factible. Como Y es factible, $\sum_{i=1}^n y_i * w_i \leq W$ y, por tanto, $\sum_{i=1}^n (x_i - y_i) * w_i \geq 0$. Tendremos que $V(Y) = \sum_{i=1}^n y_i * v_i$ es el valor de la solución Y . Multiplicando y dividiendo por w_i , nos queda la resta de ambas soluciones:

$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) * v_i = \sum_{i=1}^n (x_i - y_i) * w_i * \frac{v_i}{w_i}.$$

Vemos de nuevo los casos posibles:

1. Cuando $i < j$, $x_i = 1$ y, por tanto, $(x_i - y_i)$ es positivo o nulo, mientras que $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$.
2. Cuando $i > j$, $x_i = 0$ y, por tanto, $(x_i - y_i)$ es negativo o nulo, mientras que $\frac{v_i}{w_i} \leq \frac{v_j}{w_j}$.
3. Cuando $i = j$, $\frac{v_i}{w_i} = \frac{v_j}{w_j}$.

Por tanto, en todos los casos se tiene que $(x_i - y_i) * \frac{v_j}{w_j} \geq (x_j - y_j) * \frac{v_i}{w_i}$

Con lo que se deduce que:

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

Por tanto, hemos demostrado que ninguna solución factible puede tener un valor mayor que $V(X)$, por lo que la solución X es óptima.

NOTA DEL AUTOR: Es una interpretación escrita de otra manera a la del libro, pero tomando el texto. Se distinguen casos para que sea más entendible.

Análisis del coste: como en ocasiones anteriores, tendremos estos costes:

- Calcular los $\frac{v_i}{w_i}$ tiene coste $O(n)$.
- Ordenar de mayor a menor relación valor-peso $\frac{v_i}{w_i}$: $O(n * \log(n))$, pudiendo usar cualquier algoritmo para ello, heapsort, quicksort, etc.
- El esquema voraz tiene coste $O(n)$, en el caso peor.

Por tanto, la ordenación determina el coste del algoritmo, que sería por norma general **$O(n * \log(n))$** .

Mejora del algoritmo: Usaremos **montículos de máximos**, estando el mayor valor $\frac{v_i}{w_i}$ en la raíz. Las operaciones, como en ocasiones anteriores serán:

- Crear montículo tiene coste $O(n)$.
- Para flotar o hundir requeriremos $O(\log(n))$, por lo que la propiedad del montículo debe ser restaurada como máximo n veces (tantas como nodos haya).

En el caso peor, el coste será de $O(n * \log(n))$.

En el caso mejor, es más rápido si solo se necesitan unos pocos objetos para llenar la mochila, será coste casi lineal (estimación del autor, no del libro): $O(n)$.

6.6. Planificación.

Presentaremos dos **problemas** que conciernen a la forma óptima de planificar tareas en una sola máquina:

1. El problema consiste en minimizar el tiempo que invierte cada tarea en el sistema.
2. Las tareas tienen un plazo fijo de ejecución y cada tarea aporta unos ciertos beneficios sólo si está acabada al llegar al plazo: nuestro objetivo es maximizar la rentabilidad.

Pasamos a ver estos problemas con más detenimiento.

6.6.1. Minimización del tiempo en el sistema.

Un único servidor, como, por ejemplo, un dentista, un surtidor de gasolina o un cajero de un banco, tiene que dar servicio a n clientes. El tiempo requerido por cada cliente se conoce de antemano: el cliente i requerirá un tiempo t_i para $1 \leq i \leq n$. Deseamos minimizar el tiempo medio requerido por cada cliente en el sistema. A ser el número de clientes predeterminado equivale a minimizar el tiempo total invertido en el sistema por todos los clientes. Por tanto, deseamos **minimizar**

$$T = \sum_{i=1}^n (\text{tiempo en el sistema para el cliente } i)$$

Ejemplo de minimización del tiempo en el sistema: Tenemos tres clientes $t_1 = 5$, $t_2 = 10$ y $t_3 = 3$. Existen 6 órdenes de servicio posibles:

<u>Orden</u>	<u>Tiempo total invertido en el sistema</u>
1 2 3	$5 + (5 + 10) + (5 + 10 + 3) = 38$
1 3 2	$5 + (5 + 3) + (5 + 3 + 10) = 31$
2 1 3	$10 + (10 + 5) + (10 + 5 + 3) = 43$
2 3 1	$10 + (10 + 3) + (10 + 3 + 5) = 41$
3 1 2	$3 + (3 + 5) + (3 + 5 + 10) = 29 \quad \leftarrow \text{Óptimo}$
3 2 1	$3 + (3 + 10) + (3 + 10 + 5) = 34$

En el primer orden, se sirve al cliente 1, el cliente 2 espera mientras se sirve al cliente 1 y entonces le llega el turno y el cliente 3 espera mientras se sirve a los clientes 1 y 2 y se le sirve en el último lugar. El tiempo total invertido en el sistema por los 3 clientes es de 38.

La planificación óptima se obtiene por **orden creciente de tiempos de servicio**: el cliente 3, que es el de menor tiempo de servicio es servido el primero, mientras que el cliente 2, que es el de mayor tiempo de servicio, es servido en último orden.

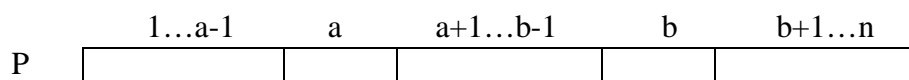
Para ver esta idea de que puede ser óptimo planificar los clientes por orden creciente de tiempos de servicio, imaginemos un algoritmo voraz que construya la solución óptima elemento a elemento. Supongamos que después de planificar el servicio para los clientes i_1, i_2, \dots, i_m se añade un cliente j . El incremento de tiempos en esta fase es igual a la suma de los tiempos de servicio para los clientes desde i_1 hasta i_m más t_j , que es el tiempo necesario para servir al cliente j . Para minimizar esto, debemos **minimizar** t_j . Nuestro algoritmo voraz es bastante sencillo: en cada paso se añade al final de la planificación al cliente que requiere el menor servicio entre los restantes.

Teorema 6.6.1 El algoritmo voraz es óptimo.

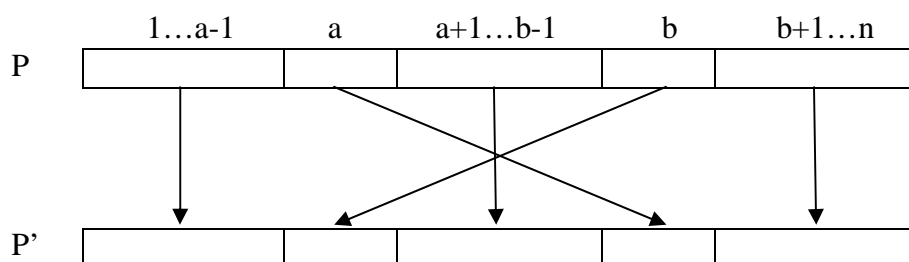
Demostración: Sea $P = p_1 p_2 \dots p_n$ cualquier permutación de enteros del 1 al n y sea $s_i = t_{p_i}$. Si se sirven clientes en el orden P , entonces el tiempo requerido por el j -ésimo cliente que haya que servir será s_j y el tiempo transcurrido en el sistema por todos los clientes es:

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \dots + (s_1 + s_2 + s_3 + \dots + s_n) = \\ &= n * s_1 + (n-1) * s_2 + \dots + 2 * s_{n-1} + s_n = \\ &= \sum_{k=1}^n (n-k+1) * s_k. \end{aligned}$$

Supongamos ahora que P no organiza a los clientes por orden de tiempos crecientes de servicio. Entonces, se pueden encontrar dos enteros a y b con $a < b$ y $s_a > s_b$. Es decir, se sirven al cliente a -ésimo antes que al b -ésimo, aun cuando el primero necesite más tiempo de servicio que el segundo. Sería algo así:



Si intercambiamos la posición de esos dos clientes, obtendremos un nuevo orden de servicio o permutación P' , que es simplemente el orden P después de intercambiar p_a y p_b :



El **tiempo total** transcurrido pasado en el sistema por todos los clientes si se emplea la planificación P' es:

$$T(P') = (n-a+1) * s_b + (n-b+1) * s_a + \sum_{\substack{k=1 \\ k \neq a, b}}^n (n-k+1) * s_k$$

La nueva planificación es preferible a la vieja, porque:

$$\begin{aligned} T(P) - T(P') &= (n-a+1) * (s_a - s_b) + (n-b+1) * (s_b - s_a) = \\ &= \underbrace{(b-a)}_{>0} * \underbrace{(s_a - s_b)}_{>0} > 0 \end{aligned}$$

Se observa tras el intercambio que los clientes salen en su posición adecuada, ya que $s_b < s_a$ por nuestra suposición inicial, estando el resto ordenados. Por tanto, P' es mejor que P en conjunto.

De esta manera, se puede optimizar toda planificación en la que se sirva a un cliente antes que requiera menos servicio. Las únicas planificaciones que quedan son aquellas que se obtienen poniendo a los clientes por orden creciente de tiempo de servicio. Todas las planificaciones son equivalentes y, por tanto, todas son óptimas.

La **implementación** de este algoritmo es sencilla.

Análisis del coste: tendremos este coste del algoritmo:

- Ordenar los elementos por orden de tiempos creciente (no decreciente, usando cualquier algoritmo de ordenación (montículo, quicksort, ...): $O(n * \log(n))$)
- Coste del algoritmo voraz: $O(n)$.

Podemos **mejorarlo** usando montículos invertidos (de mínimos):

- Crear montículo: $O(n)$
- Restaurar la condición del montículo n veces: $O(n * \log(n))$

Por tanto, el coste asintóticamente coincidirá en ambas y será $O(n * \log(n))$.

6.6.2. Planificación con plazo fijo.

Tenemos que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En cualquier instante $T = 1, 2, \dots, n$ podemos ejecutar únicamente una tarea. La tarea i nos produce unos beneficios $g_i > 0$ sólo en el caso en que sea ejecutada en un instante anterior a d_i .

En resumen:

- n :** Número de tareas de tiempo unitario. Por ejemplo, una hora, días,...
- $T = 1, 2, \dots, n$** En cada instante solo podemos realizar una tarea.
- g_i :** Beneficio asociado a la tarea i .
- d_i :** Plazo máximo de la tarea i .

El problema consiste en **maximizar el beneficio total**.

Un **ejemplo** de este algoritmo será:

i	1	2	3	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	← Óptima
1,3	65	4, 3	45	

Puede haber tareas que se queden sin realizar. Tendremos, por tanto:

- **Conjunto de candidatos:** Las tareas.
- **Conjunto factible:** Se dice que un conjunto de tareas es factible si existe, al menos, una sucesión de sus tareas que permite que todas ellas se ejecute dentro de plazo.
- **Función de selección:** Un algoritmo voraz evidente consiste en construir la planificación paso a paso, añadiendo en cada paso la tarea que tenga el mayor valor de g_i (ganancia) y cuando el conjunto de tareas seleccionadas siga siendo *factible*.

Nuestra solución óptima es ejecutar las tareas en el orden 4, 1. Queda por *demonstrar* que este algoritmo siempre encuentra una planificación óptima, y además hay que buscar una forma eficiente de implementarlo.

Sea J un conjunto de tareas. Necesitamos probar las $k!$ permutaciones de estas tareas para ver si J es factible. Vemos un lema que nos indicará que esto no es así:

Lema 6.6.2 Sea J un conjunto de k tareas. Supongamos que las tareas están numeradas de tal forma que $d_1 \leq d_2 \leq \dots \leq d_k$. Entonces el conjunto J es factible si y sólo si la secuencia $1, 2, \dots, k$ es factible.

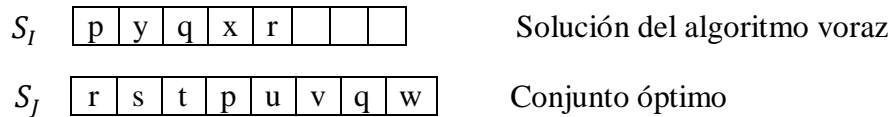
Demostración (por contradicción): El “si” (\Rightarrow) es evidente. Para el “sólo si” (\Leftarrow), supongamos que la secuencia $1, 2, \dots, k$ *no* es factible. Entonces, al menos, una de estas tareas se planifica después del plazo. Sea r cualquiera de estas tareas, de tal manera que $d_r \leq r - 1$. Dado que las tareas se planifican por orden no decrecientes (crecientes) de plazos, esto significa que, al menos, r tareas tienen como fecha final $r - 1$ o anterior. Sea cual fuere la forma en que se planifiquen, la última siempre llegará tarde, es decir, se saldrá del plazo.

Esto demuestra que basta comprobar una sola secuencia, en orden no decreciente, para saber si un conjunto de tareas J es o no factible.

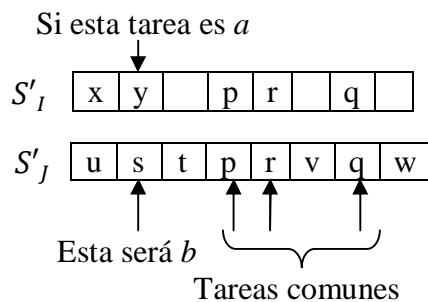
NOTA DEL AUTOR: Lo de la demostración por contradicción, de nuevo, es un añadido, por el texto de la propia demostración.

Teorema 6.6.3 El algoritmo voraz esbozado anteriormente siempre encuentra solución óptima.

Demostración: Supongamos que el algoritmo voraz decide ejecutar un conjunto de tareas I , y supongamos que el conjunto J es óptimo. Sean S_I y S_J secuencias factibles, que posiblemente incluyan huecos, para los dos conjuntos de tareas en cuestión. Gráficamente, sería esto:



Reorganizando las tareas de S_I y S_J , podemos obtener dos secuencias factibles S'_I y S'_J , que también pueden contener huecos tales que toda tarea común a I y a J se planifique en el mismo instante en ambas secuencias. Tras reorganizar las tareas quedaría así:



Para ver esto, imaginemos que alguna tarea a aparece en las dos secuencias factibles S_I y S_J , en donde queda planificada en los instantes t_I y t_J , respectivamente. Se nos darán estos casos:

- Si $t_I = t_J$ no hay nada que hacer, ya que coinciden ambas tareas en tiempo (apreciación del autor).
- En caso contrario, supongamos que $t_I < t_J$ (es decir, se ejecuta antes la misma tarea para la secuencia S_I que para la S_J , apreciación del autor). Dado que la secuencia S_I es factible, se sigue que el plazo para la tarea a no es anterior a t_J . Se modifica la secuencia S_I de la siguiente manera:
 - Si hay un hueco en la secuencia S_I en el instante t_J , se atrasa la tarea a del instante t_I al hueco en el instante t_J .
 - Si hay una tarea b planificada en S_I en el instante t_J , se intercambian las tareas a y b en la secuencia S_I .

La secuencia resultante sigue siendo factible, puesto que, en cualquier caso, a se ejecutará antes de su plazo y en el segundo caso el traslado de b a un instante anterior, no puede causar daños. Ahora, se planifica a en un instante t_I en las dos secuencias modificadas S_I y S_J .

- Se puede aplicar un argumento similar cuando $t_I > t_J$ salvo que, en este caso, es S_J quien debe ser modificada.

Una vez que se ha tratado una tarea a de esta manera, está claro que nunca será preciso volver a trasladarla. Por tanto, si las secuencias S_I y S_J tienen en común, después de un máximo de m modificaciones de S_I o de S_J podemos asegurar que todas las tareas comunes a I y a J estarán planificadas al mismo tiempo en ambas secuencias. Las secuencias resultantes S'_I y S'_J pueden no ser iguales si $i \neq j$. Por tanto, supongamos que existe un instante en el cual la tarea planificada en S_I es distinta de la planificada en S'_I .

- Si alguna tarea a está planificada en S'_I frente a un hueco de S'_J , entonces a no pertenece a J . El conjunto $J \cup \{a\}$ es factible, porque podríamos poner a en el hueco y sería más rentable que J . Esto es **imposible**, puesto que J es óptimo por hipótesis.
- Si alguna tarea b está planificada en S'_J frente a un hueco S'_I , el conjunto $I \cup \{b\}$ sería factible, así que el algoritmo voraz habría incluido a b en I . Esto también es **imposible**, porque no lo hizo.
- La única posibilidad restante es que alguna tarea a esté planificada en S'_I al lado de una tarea distinta b en S'_J (como las tareas y y s en el dibujo anterior). En este caso, **a no aparece en J y b no aparece en I** . Aparentemente, hay 3 posibilidades:
 - Si $g_a > g_b$ (la ganancia de la tarea a es mayor que la de b), se podría sustituir a por b en J y mejorarla. Esto es **imposible**, porque J es óptima.
 - Si $g_a < g_b$, el algoritmo voraz habrá seleccionado a b antes de considerar a a , puesto que $(I \setminus \{a\}) \cup \{b\}$ sería factible. Esto es **imposible**, puesto que el algoritmo no incluyó a b en I .
 - La única posibilidad restante es que $g_a = g_b$.

Concluimos que para toda posición temporal las secuencias S'_I y S'_J , o bien:

- No planifican bien las tareas.
- Planifican la misma tarea.
- Planifican dos tareas distintas que producen idéntico beneficio.

El beneficio total de I es igual al beneficio del conjunto óptimo J , así que J es óptimo.

Implementaciones: Tendremos dos tipos:

1ª implementación: Suponemos que las tareas están numeradas de tal manera que $g_1 \geq g_2 \geq \dots \geq g_n$, además que $n > 0$ y $d_i > 0$, para $1 \leq i \leq n$.

Añadimos una posición en el vector que será nuestro “centinela”, usados para evitar comprobaciones repetitivas de rangos que consumen mucho tiempo.

```

funcion 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 cuantas 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 está considerando la tarea i, el algoritmo comprueba si se puede insertar en j en lugar oportuno sin llevar alguna tarea que ya está en j más allá de su plazo. De ser así, i se acepta; en caso contrario, i se rechaza.

Las tareas están numeradas por **orden decreciente de beneficios**.

Un **ejemplo** de este algoritmo será:

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

Observamos que ya están ordenados por orden decreciente de beneficios, tal y como dijimos antes (es importante).

Los pasos serán:

	1	2	3	4	5	6
Inicialmente:	1					

Primer intento:	2	1				
-----------------	---	---	--	--	--	--

Segundo intento: Sin cambios

Tercero intento:	2	1	4			
------------------	---	---	---	--	--	--

Cuarto intento: Sin cambios

Quinto intento: Sin cambios

La secuencia óptima es la 2, 1, 4 con valor 42

Análisis del coste: tendremos como en ocasiones anteriores estos pasos:

- Ordenación de las tareas: $O(n * \log(n))$. Recordemos que se puede emplear cualquier algoritmo, como el de heapsort, quicksort, etc.
- Algoritmo voraz: En el **caso peor**, las tareas están clasificadas por orden decreciente de plazos. En este caso, cuando se está considerando la tarea i , el algoritmo examina las $k = i - 1$ tareas ya planificadas, para encontrar un lugar para el recién llegado y luego las desplaza todas un lugar.

El algoritmo requiere, por tanto, un tiempo está en $\Omega(n^2)$. Es, por tanto, *ineficiente*, por ver donde “insertamos” la tarea en cuestión.

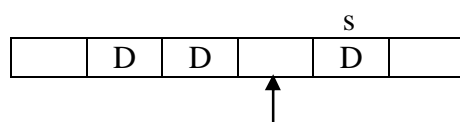
2ª implementación: Lo veremos mediante un lema, que será:

Lema 6.6.4 Un conjunto J de n tareas es factible si y sólo si se puede construir una secuencia factible que incluya a todas las tareas de J en la forma siguiente. Se empieza por una planificación vacía, de longitud n . entonces para cada tarea $i \in J$ sucesivamente, se planifica i en el instante t_i en donde t es el mayor entero tal que $1 \leq t \leq \min(n, d_i)$ y la tarea que se ejecuta en el instante t no está decidida todavía.

En otras palabras, se empieza por una planificación vacía, se considera cada tarea sucesivamente, y se añade a la planificación que se está construyendo en **el momento más tardío posible** (mucho cuidado, que es básico para esta implementación), pero no antes de su fecha final.

Demostración: El “sí” es evidente. Para el “sólo sí”; obsérvese primero que si existe una secuencia factible, entonces existe una secuencia factible de longitud n (número de tareas). Puesto que sólo hay n tareas por planificar, toda secuencia más larga tendrá que contener huecos y siempre se puede trasladar una tarea a un hueco anterior sin afectar a la factibilidad de la secuencia.

Cuando se intenta añadir una nueva tarea, la secuencia que se está construyendo contiene siempre al menos un hueco. Supongamos que no se puede añadir una tarea cuyo plazo sea d . Esto puede suceder solamente si todas las posiciones desde $t = 1$ hasta $t = r$ están ya reservadas, en donde $r = \min(n, d)$. Sea $s > r$ el menor entero tal que la posición $t = s$ está vacía. La planificación que ya se ha construido incluye, por tanto, $s - 1$ tareas, ninguna tarea con plazo exactamente a s y quizá otra más con plazos posteriores a s . Por tanto, J contiene al menos s tareas cuyos plazos son $s - 1$ o anteriores. Sea cual fuere la forma en que se planifiquen, la última llegará tarde con certeza. La situación quedaría así:



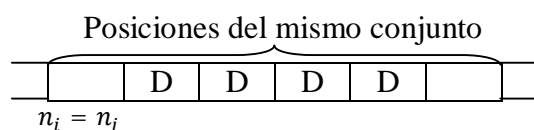
Hemos marcado con una D las tareas ya decididas. Pondremos **la tarea lo más tarde posible**.

Por ejemplo, si tenemos que la tarea tiene plazo 5, lo pondremos en la posición 4 (en la flecha) y no en 1.

El lema sugiere que deberíamos considerar un algoritmo que intente llenar una por una las posiciones de una secuencia de longitud p , donde $p = \min(n, \max_{1 \leq i \leq n} d_i)$. Para cualquier posición t , se define:

$$n_t = \max\{k \leq t \mid \text{la posición } k \text{ no está decidida todavía}\}$$

Se definen ciertos conjuntos de posiciones en la forma siguiente: Dos posiciones i y j están en el mismo conjunto si $n_i = n_j$



Al igual que antes, D indica que la posición está ocupada (la tarea ya decidida), mientras que la que está en blanco la posición está libre.

A medida que se asignan nuevas tareas a posiciones vacantes, los conjuntos se fusionan para formar conjuntos más grandes, para ello se usan **estructuras de partición**.

Para un conjunto dado K de posiciones, sea $F(K)$ el menor elemento de K . finalmente, se define una posición ficticia cero, que siempre está libre.

El **algoritmo** será el siguiente:

- i. Iniciación: Toda posición $0, 1, 2, \dots, p$ está en un conjunto diferente y $F([i]) = i, 0 \leq i \leq p$.

$$p = \min(n, \max(d_i)).$$

Mayor de los plazos

Número de tareas

La posición 0 sirve para ver cuando la planificación está llena.

- ii. Adición de una tarea con plazo d ; se busca un conjunto que contenga a d ; sea K este conjunto. Si $F(K) = 0$ se rechaza la tarea, en caso contrario:
- Se asigna la nueva tarea a la posición $F(K)$.
 - Se busca el conjunto que contenga $F(K) - 1$. Llamemos L a este conjunto (no puede ser igual a K).
 - Se fusionan K y L . El valor de F para este nuevo conjunto es el valor viejo de $F(L)$.

Tendremos un **enunciado más preciso** del algoritmo *rápido*. Para simplificar la descripción, suponemos que la etiqueta del conjunto producido por una operación de fusionar es necesariamente la etiqueta de uno de los conjuntos que hayan sido fusionados. La planificación en primer lugar puede contener huecos; el algoritmo acaba por trasladar tareas hacia delante para llenarlos.

```

funcion secuencia2 ( $d[1..n]$ ):  $k$ , matriz  $[1..k]$ 
    matriz  $j$ ,  $F[0..n]$ 
    { Iniciación }
     $p = \min(n, \max\{d[i] | 1 \leq i \leq n\})$ ;
    para  $i \leftarrow 0$  hasta  $p$  hacer  $j[i] \leftarrow 0$ 
                                 $F[i] \leftarrow i$ 
                                Iniciar el conjunto  $\{i\}$ 

    { Bucle voraz }
    para  $i \leftarrow 1$  hasta  $n$  hacer    { Orden decreciente de  $g$  }
         $k \leftarrow \text{buscar}(\min(p, d[i]))$ 
         $m \leftarrow F[k]$ 
        si  $m \neq 0$  entonces
             $j[m] \leftarrow i$ ;
             $l \leftarrow \text{buscar}(m - 1)$ 
             $F[k] \leftarrow F[l]$ 
            { El conjunto resultante tiene la etiqueta  $k$  o  $l$  }
            fusionar  $(k, l)$ 
    { Sólo queda comprimir la solución }
     $k \leftarrow 0$ 
    para  $i \leftarrow 1$  hasta  $p$  hacer
        si  $j[i] > 0$  entonces  $k \leftarrow k + 1$ 
                                 $j[k] \leftarrow j[i]$ 
    devolver  $k, j[1..k]$ 

```

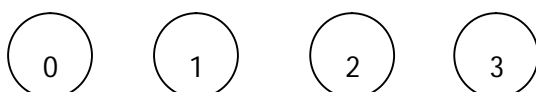
Ejemplo de la segunda implementación, siendo el mismo ejemplo anterior:

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

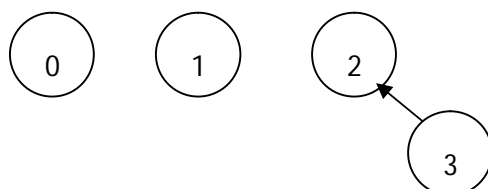
De nuevo, están ordenados por orden decreciente de ganancias. Los pasos son los siguientes:

Inicialmente: $p = \min(n, \max(d_i)) = \min(6, 3) = 3$.

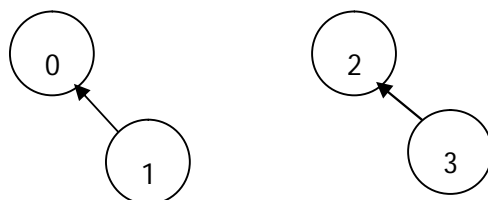
Por tanto, como máximo tendremos una planificación de 3 tareas:



Primer intento: $d_1 = 3$. Se asigna la tarea 1 a la posición 3.
 $F(K) = 3, F(L) = F(K) - 1 = 2$. Fusionamos K con L



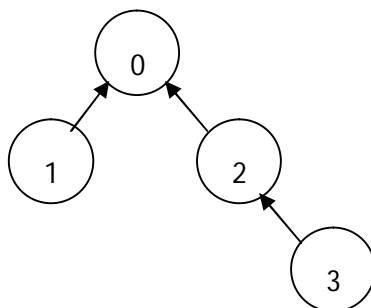
Segundo intento: $d_2 = 1$. Se asigna la tarea 2 a la posición 1.
 $F(K) = 1, F(L) = F(K) - 1 = 0$. Fusionamos K con L



Tercer intento: $d_3 = 1$. No hay posiciones libres disponibles porque el valor de F es 0.

Cuarto intento: $d_4 = 3$. Se asigna la tarea 4 a la posición 3.

$F(K) = 1, F(L) = F(K) - 1 = 0$. Fusionamos K con L



Quinto y sexto intento: No hay posiciones libres disponibles.

La secuencia óptima es la 2, 4, 1 con valor 42.

Análisis del coste: Usaremos operaciones de conjuntos disjuntos, en la que la hay que ejecutar, como máximo, $2 * n$ operaciones *buscar* y n operaciones *fusionar*, por lo que el tiempo requerido está en $O(n * \alpha(2 * n, n))$, por tanto, tiene coste lineal ($O(n)$).

En el **caso peor**, en el que suponemos que todas las tareas están desordenadas, habría que ordenarlas, por lo que, de nuevo, el coste es $O(n * \log(n))$, que determinará el coste del algoritmo (apreciación del autor).