

Algoritmos voraces

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Octubre 2015

Bibliografía

- R. Neapolitan. *Foundations of Algorithms*. Quinta edición. Jones and Bartlett Learning, 2015.
Capítulo 4
- E. Horowitz, S. Sahni y S. Rajasekaran. *Computer Algorithms*. Tercera edición. Computer Science Press, 1998.
Capítulo 4
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición, Garceta, 2013.
Capítulo 12

Método voraz

Las características generales de los algoritmos voraces son las siguientes:

- Para construir la solución se dispone de un **conjunto de candidatos**. A medida que avanza el algoritmo se van formando dos conjuntos: el conjunto de candidatos **seleccionados** (formarán parte de la solución), y el conjunto de candidatos **rechazados** definitivamente.
- Existe una **función de selección** que indica cuál es el candidato más prometedor de entre los aún no considerados.
- Existe un **test de factibilidad** que comprueba si un candidato es compatible con la solución parcial construida hasta el momento.
- Existe un **test de solución** que determina si una solución parcial forma una solución “completa”.
- Con frecuencia, se trata de problemas de optimización, es decir, se tiene que obtener una solución óptima según una **función objetivo** que asocia un valor a cada solución.

Método voraz

- El funcionamiento general consiste en añadir a los seleccionados el candidato devuelto por la función de selección si así se sigue cumpliendo el test de factibilidad. Si no, rechazarlo.

Método voraz

- El funcionamiento general consiste en añadir a los seleccionados el candidato devuelto por la función de selección si así se sigue cumpliendo el test de factibilidad. Si no, rechazarlo.
- Lo fundamental es que cada candidato se trata **una sola vez**, es decir, no hay posibilidad de replantearse una decisión tomada.
- Los algoritmos son sencillos y eficientes.

Método voraz

- El funcionamiento general consiste en añadir a los seleccionados el candidato devuelto por la función de selección si así se sigue cumpliendo el test de factibilidad. Si no, rechazarlo.
- Lo fundamental es que cada candidato se trata **una sola vez**, es decir, no hay posibilidad de replantearse una decisión tomada.
- Los algoritmos son sencillos y eficientes.
- Importante: **demostración de corrección**.
- Método de **reducción de diferencias**: comparar una solución óptima con la solución obtenida por el algoritmo voraz. Si ambas soluciones no son iguales, se va transformando la solución óptima de partida de forma que continúe siendo óptima, pero siendo más parecida a la del algoritmo voraz.

Esquema de algoritmo voraz

```
fun voraz(datos : conjunto) dev S : conjunto
var candidatos : conjunto
    S :=  $\emptyset$     { en S se va construyendo la solución }
    candidatos := datos
    mientras candidatos  $\neq \emptyset$   $\wedge$   $\neg$ es-solución?(S) hacer
        x := seleccionar(candidatos)
        candidatos := candidatos - {x}
        si es-factible?(S  $\cup$  {x}) entonces S := S  $\cup$  {x} fsi
    fmientras
ffun
```

Maximizar número de programas

Consideremos n programas P_1, P_2, \dots, P_n que debemos almacenar en un disco. El programa P_i requiere s_i megabytes de espacio de disco, y la capacidad del disco es D megabytes. Queremos maximizar el *número de programas* almacenados en el disco.

Maximizar número de programas

Consideremos n programas P_1, P_2, \dots, P_n que debemos almacenar en un disco. El programa P_i requiere s_i megabytes de espacio de disco, y la capacidad del disco es D megabytes. Queremos maximizar el *número de programas* almacenados en el disco.

Supondremos que $D < \sum_{i=1}^n s_i$.

Considerar los programas **de menor a mayor tamaño**: si el programa cabe en el espacio de disco todavía no utilizado, grabar el programa y pasar al siguiente; y si el programa no cabe, descartarlo y terminar.

Maximizar número de programas

Consideremos n programas P_1, P_2, \dots, P_n que debemos almacenar en un disco. El programa P_i requiere s_i megabytes de espacio de disco, y la capacidad del disco es D megabytes. Queremos maximizar el *número de programas* almacenados en el disco.

Supondremos que $D < \sum_{i=1}^n s_i$.

Considerar los programas **de menor a mayor tamaño**: si el programa cabe en el espacio de disco todavía no utilizado, grabar el programa y pasar al siguiente; y si el programa no cabe, descartarlo y terminar.

```
//  $S[0] \leq S[1] \leq \dots \leq S[n-1] \wedge D < \sum_{i=0}^{n-1} S[i]$ 
vector<bool> programas_en_disco(vector<int> const& S, int D) {
    vector<bool> solucion(S.size(), false);
    int acumula = 0;
    for (size_t i = 0; acumula + S[i] <= D; ++i) {
        solucion[i] = true;
        acumula += S[i];
    }
    return solucion;
}
```

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

En una solución cualquiera $Z = (z_1, z_2, \dots, z_n)$, $z_i = 0$ indica que el programa P_i no se coge, mientras que $z_i = 1$ indica que P_i forma parte de la solución. El número de programas seleccionados es $\sum_{i=1}^n z_i$.

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

En una solución cualquiera $Z = (z_1, z_2, \dots, z_n)$, $z_i = 0$ indica que el programa P_i no se coge, mientras que $z_i = 1$ indica que P_i forma parte de la solución. El número de programas seleccionados es $\sum_{i=1}^n z_i$.

La estrategia voraz propuesta escoge solo los k primeros programas. Entonces, en la solución X se tiene $\forall i : 1 \leq i \leq k : x_i = 1$ y $\forall i : k < i \leq n : x_i = 0$.

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

En una solución cualquiera $Z = (z_1, z_2, \dots, z_n)$, $z_i = 0$ indica que el programa P_i no se coge, mientras que $z_i = 1$ indica que P_i forma parte de la solución. El número de programas seleccionados es $\sum_{i=1}^n z_i$.

La estrategia voraz propuesta escoge solo los k primeros programas. Entonces, en la solución X se tiene $\forall i : 1 \leq i \leq k : x_i = 1$ y $\forall i : k < i \leq n : x_i = 0$.

Empezando a comparar (de izquierda a derecha) con Y , sea $j \geq 1$ la primera posición donde $x_j \neq y_j$.

Se tiene la siguiente situación:

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

En una solución cualquiera $Z = (z_1, z_2, \dots, z_n)$, $z_i = 0$ indica que el programa P_i no se coge, mientras que $z_i = 1$ indica que P_i forma parte de la solución. El número de programas seleccionados es $\sum_{i=1}^n z_i$.

La estrategia voraz propuesta escoge solo los k primeros programas. Entonces, en la solución X se tiene $\forall i : 1 \leq i \leq k : x_i = 1$ y $\forall i : k < i \leq n : x_i = 0$.

Empezando a comparar (de izquierda a derecha) con Y , sea $j \geq 1$ la primera posición donde $x_j \neq y_j$.

Se tiene la siguiente situación:

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & & & & & & & & & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

En una solución cualquiera $Z = (z_1, z_2, \dots, z_n)$, $z_i = 0$ indica que el programa P_i no se coge, mientras que $z_i = 1$ indica que P_i forma parte de la solución. El número de programas seleccionados es $\sum_{i=1}^n z_i$.

La estrategia voraz propuesta escoge solo los k primeros programas. Entonces, en la solución X se tiene $\forall i : 1 \leq i \leq k : x_i = 1$ y $\forall i : k < i \leq n : x_i = 0$.

Empezando a comparar (de izquierda a derecha) con Y , sea $j \geq 1$ la primera posición donde $x_j \neq y_j$.

Se tiene la siguiente situación:

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & & & & & & & & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

En una solución cualquiera $Z = (z_1, z_2, \dots, z_n)$, $z_i = 0$ indica que el programa P_i no se coge, mientras que $z_i = 1$ indica que P_i forma parte de la solución. El número de programas seleccionados es $\sum_{i=1}^n z_i$.

La estrategia voraz propuesta escoge solo los k primeros programas. Entonces, en la solución X se tiene $\forall i : 1 \leq i \leq k : x_i = 1$ y $\forall i : k < i \leq n : x_i = 0$.

Empezando a comparar (de izquierda a derecha) con Y , sea $j \geq 1$ la primera posición donde $x_j \neq y_j$.

Se tiene la siguiente situación:

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & \dots & \neq & & & & & & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Maximizar número de programas, demostración de corrección

Demostramos que la estrategia conduce siempre a una solución óptima, comparando la solución devuelta por la estrategia voraz, X , con otra solución óptima, Y .

En una solución cualquiera $Z = (z_1, z_2, \dots, z_n)$, $z_i = 0$ indica que el programa P_i no se coge, mientras que $z_i = 1$ indica que P_i forma parte de la solución. El número de programas seleccionados es $\sum_{i=1}^n z_i$.

La estrategia voraz propuesta escoge solo los k primeros programas. Entonces, en la solución X se tiene $\forall i : 1 \leq i \leq k : x_i = 1$ y $\forall i : k < i \leq n : x_i = 0$.

Empezando a comparar (de izquierda a derecha) con Y , sea $j \geq 1$ la primera posición donde $x_j \neq y_j$.

Se tiene la siguiente situación:

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & \dots & \neq & & & & & \neq & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & \dots & \neq & & & & & \neq & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Si $y_j \neq x_j = 1$, tenemos $y_j = 0$, y $\sum_{i=1}^j y_i = j - 1 < j = \sum_{i=1}^j x_i$.

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & \dots & \neq & & & & & \neq & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Si $y_j \neq x_j = 1$, tenemos $y_j = 0$, y $\sum_{i=1}^j y_i = j - 1 < j = \sum_{i=1}^j x_i$.

Como Y es óptima, $\sum_{i=1}^n y_i \geq \sum_{i=1}^n x_i = k$. Existe $l > k \geq j$ tal que $y_l = 1$.

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & \dots & \neq & & & & & \neq & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Si $y_j \neq x_j = 1$, tenemos $y_j = 0$, y $\sum_{i=1}^j y_i = j - 1 < j = \sum_{i=1}^j x_i$.

Como Y es óptima, $\sum_{i=1}^n y_i \geq \sum_{i=1}^n x_i = k$. Existe $l > k \geq j$ tal que $y_l = 1$.

Por la ordenación de los programas sabemos que $s_j \leq s_l$, es decir, que si P_l cabe en el disco, podemos poner en su lugar P_j sin sobrepasar la capacidad total.

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & \dots & \neq & & & & & \neq & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Si $y_j \neq x_j = 1$, tenemos $y_j = 0$, y $\sum_{i=1}^j y_i = j - 1 < j = \sum_{i=1}^j x_i$.

Como Y es óptima, $\sum_{i=1}^n y_i \geq \sum_{i=1}^n x_i = k$. Existe $l > k \geq j$ tal que $y_l = 1$.

Por la ordenación de los programas sabemos que $s_j \leq s_l$, es decir, que si P_l cabe en el disco, podemos poner en su lugar P_j sin sobrepasar la capacidad total.

Realizando este cambio en Y , obtenemos otra solución Y' en la cual $y'_j = 1 = x_j, y'_l = 0$ y para el resto $y'_i = y_i$. Esta nueva solución es más parecida a X , y tiene el mismo número de programas que Y , por lo que sigue siendo óptima.

$$\begin{array}{cccccccccccc}
 1 & 1 & \dots & 1 & \dots & 1 & 0 & \dots & 0 & \dots & 0 \\
 x_1 & x_2 & \dots & x_j & \dots & x_k & x_{k+1} & \dots & x_l & \dots & x_n \\
 = & = & \dots & \neq & & & & & \neq & & \\
 y_1 & y_2 & \dots & y_j & \dots & y_k & y_{k+1} & \dots & y_l & \dots & y_n
 \end{array}$$

Si $y_j \neq x_j = 1$, tenemos $y_j = 0$, y $\sum_{i=1}^j y_i = j - 1 < j = \sum_{i=1}^j x_i$.

Como Y es óptima, $\sum_{i=1}^n y_i \geq \sum_{i=1}^n x_i = k$. Existe $l > k \geq j$ tal que $y_l = 1$.

Por la ordenación de los programas sabemos que $s_j \leq s_l$, es decir, que si P_l cabe en el disco, podemos poner en su lugar P_j sin sobrepasar la capacidad total.

Realizando este cambio en Y , obtenemos otra solución Y' en la cual $y'_j = 1 = x_j$, $y'_l = 0$ y para el resto $y'_i = y_i$. Esta nueva solución es más parecida a X , y tiene el mismo número de programas que Y , por lo que sigue siendo óptima.

Repitiendo este proceso, podemos ir igualando los programas en la solución óptima a los de la solución voraz X , hasta alcanzar la posición k .

Planificación de tareas: tiempo medio mínimo

Un sistema da servicio a n tareas, cada una con un tiempo de ejecución t_i . Se desea minimizar el tiempo medio de estancia de una tarea en el sistema, esto es, el tiempo transcurrido desde el comienzo de todo el proceso hasta que la tarea termina de ejecutarse.

Planificación de tareas: tiempo medio mínimo

Un sistema da servicio a n tareas, cada una con un tiempo de ejecución t_i . Se desea minimizar el tiempo medio de estancia de una tarea en el sistema, esto es, el tiempo transcurrido desde el comienzo de todo el proceso hasta que la tarea termina de ejecutarse.

El problema consiste en minimizar

$$TM = \frac{1}{n} \sum_{i=1}^n T_i$$

donde T_i es el tiempo en el sistema de la tarea i , y como n está fijo, esto es equivalente a minimizar el tiempo total

$$T = \sum_{i=1}^n T_i.$$

Para la tarea i , T_i es la suma de su tiempo de ejecución t_i más todos los tiempos de las tareas que se ejecutan antes que ella.

Ejemplo: $n = 3$, $t_1 = 5$, $t_2 = 10$, $t_3 = 3$

Orden de ejecución: 1, 2, 3

$$T = 5 + (5 + 10) + (5 + 10 + 3) = 38$$

Ejemplo: $n = 3$, $t_1 = 5$, $t_2 = 10$, $t_3 = 3$

Orden de ejecución: 1, 2, 3

$$T = 5 + (5 + 10) + (5 + 10 + 3) = 38$$

Orden de ejecución: 1, 3, 2

$$T = 5 + (5 + 3) + (5 + 3 + 10) = 31$$

Ejemplo: $n = 3$, $t_1 = 5$, $t_2 = 10$, $t_3 = 3$

Orden de ejecución: 1, 2, 3

$$T = 5 + (5 + 10) + (5 + 10 + 3) = 38$$

Orden de ejecución: 1, 3, 2

$$T = 5 + (5 + 3) + (5 + 3 + 10) = 31$$

Orden de ejecución: 3, 1, 2

$$T = 3 + (3 + 5) + (3 + 5 + 10) = 29$$

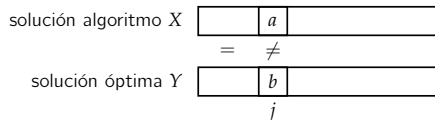
La planificación óptima se consigue cuando se atiende a las tareas por **orden creciente de tiempo de ejecución**.

La planificación óptima se consigue cuando se atiende a las tareas por **orden creciente de tiempo de ejecución**.

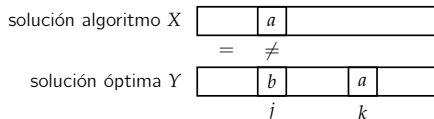
Sea $I = i_1, i_2, \dots, i_n$ una permutación cualquiera de los enteros del 1 al n .

El tiempo total correspondiente a esta permutación será

$$\begin{aligned} T(I) &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \dots + (t_{i_1} + t_{i_2} + \dots + t_{i_n}) \\ &= nt_{i_1} + (n-1)t_{i_2} + \dots + 2t_{i_{n-1}} + t_{i_n} \\ &= \sum_{k=1}^n (n-k+1)t_{i_k}. \end{aligned}$$

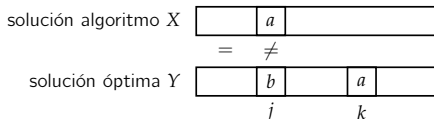


Las soluciones son iguales hasta la posición j . Como el algoritmo planifica las tareas en orden creciente de tiempo de ejecución, $t_a \leq t_b$.



Las soluciones son iguales hasta la posición j . Como el algoritmo planifica las tareas en orden creciente de tiempo de ejecución, $t_a \leq t_b$.

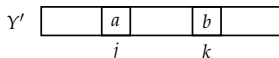
Existe una posición $k > j$ donde la solución óptima Y planifica la tarea a .



Las soluciones son iguales hasta la posición j . Como el algoritmo planifica las tareas en orden creciente de tiempo de ejecución, $t_a \leq t_b$.

Existe una posición $k > j$ donde la solución óptima Y planifica la tarea a .

Modificamos la solución Y intercambiando las tareas de las posiciones j y k :



Y' sigue siendo solución (es una permutación de las tareas), pero ¿sigue siendo óptima?

$$T(Y) = \left(\sum_{\substack{l=1 \\ l \neq j, k}}^n (n-l+1)t_{y_l} \right) + (n-j+1)t_b + (n-k+1)t_a$$

$$T(Y') = \left(\sum_{\substack{l=1 \\ l \neq j, k}}^n (n-l+1)t_{y_l} \right) + (n-j+1)t_a + (n-k+1)t_b$$

Si restamos ambos tiempos obtenemos:

$$\begin{aligned} T(Y) - T(Y') &= (n-j+1)t_b + (n-k+1)t_a - (n-j+1)t_a - (n-k+1)t_b \\ &= -jt_b + kt_b - kt_a + jt_a \\ &= t_b(k-j) - t_a(k-j) \\ &= (k-j)(t_b - t_a) \\ &\geq 0 \end{aligned}$$

Y' es también óptima.

Tareas con plazo y beneficio

El tío Facundo posee n huertas, cada una con un tipo diferente de árboles frutales. Las frutas ya han madurado y es hora de recolectarlas. La recolección de una huerta exige un día completo.

El tío Facundo conoce, para cada una de las huertas, el beneficio que obtendría por la venta de lo recolectado. También sabe los días que tardan en pudrirse los frutos de cada huerta.

Queremos decidir qué debe recolectarse y cuándo debe hacerse, para maximizar el beneficio total obtenido.

Cada una de las n huertas lleva asociado un beneficio positivo b_i que solo podrá contabilizarse si la huerta es recolectada sin superar el plazo correspondiente p_i .

Cada una de las n huertas lleva asociado un beneficio positivo b_i que solo podrá contabilizarse si la huerta es recolectada sin superar el plazo correspondiente p_i .

Un conjunto de huertas es **factible** si existe alguna secuencia de recolección admisible que permita recolectar todas las huertas del conjunto dentro de sus respectivos plazos.

Una permutación (i_1, i_2, \dots, i_k) de las huertas de un conjunto es **admisibile** si

$$\forall j : 1 \leq j \leq k : p_{i_j} \geq j$$

(el plazo correspondiente es posterior al momento en que se recolecta).

Cada una de las n huertas lleva asociado un beneficio positivo b_i que solo podrá contabilizarse si la huerta es recolectada sin superar el plazo correspondiente p_i .

Un conjunto de huertas es **factible** si existe alguna secuencia de recolección admisible que permita recolectar todas las huertas del conjunto dentro de sus respectivos plazos.

Una permutación (i_1, i_2, \dots, i_k) de las huertas de un conjunto es **admisibile** si

$$\forall j : 1 \leq j \leq k : p_{i_j} \geq j$$

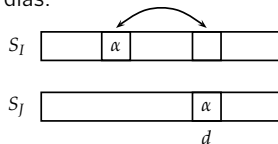
(el plazo correspondiente es posterior al momento en que se recolecta).

La estrategia voraz consiste en ir seleccionando las huertas de forma que, en cada etapa, se escoge la huerta aún no considerada con mayor beneficio, con la condición de que el subconjunto formado siga siendo *factible*.

Demostración de la optimalidad por reducción de diferencias

Llamamos I a la solución del algoritmo y J a una solución óptima.

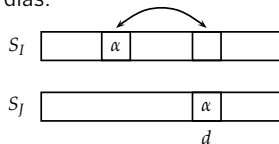
Sean S_I y S_J secuencias admisibles de las huertas de los subconjuntos I y J . Primero transformamos S_I y S_J de tal forma que las huertas comunes se recolecten en los mismos días.



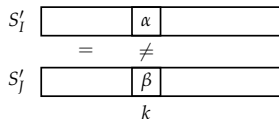
Demostración de la optimalidad por reducción de diferencias

Llamamos I a la solución del algoritmo y J a una solución óptima.

Sean S_I y S_J secuencias admisibles de las huertas de los subconjuntos I y J . Primero transformamos S_I y S_J de tal forma que las huertas comunes se recolecten en los mismos días.



Comparemos ahora las secuencias S'_I y S'_J



- $b_\alpha > b_\beta$ imposible, mejoraríamos J .
- $b_\alpha < b_\beta$ imposible, el algoritmo habría considerado antes β que α .
- $b_\alpha = b_\beta$. Se puede cambiar β por α en J .

Comprobación de la factibilidad, primera posibilidad

Para determinar si un conjunto de huertas es factible, basta con encontrar una secuencia de recolección admisible.

Es suficiente comprobar la secuencia que ordena las huertas correspondientes por plazos crecientes.

Comprobación de la factibilidad, primera posibilidad

Para determinar si un conjunto de huertas es factible, basta con encontrar una secuencia de recolección admisible.

Es suficiente comprobar la secuencia que ordena las huertas correspondientes por plazos crecientes.

Lema Un conjunto H de huertas es factible si y solo si la secuencia de las huertas de H ordenadas de forma creciente según plazo es admisible.

Demostración: La implicación hacia la izquierda es trivial, así que nos limitamos a demostrar la implicación hacia la derecha, lo que se hará por contrarrecíproco. Sea $H = \{h_1, \dots, h_k\}$ con $p_1 \leq p_2 \leq \dots \leq p_k$, tal que la secuencia h_1, h_2, \dots, h_k no es admisible, es decir, existe alguna huerta $h_r, r \in \{1, \dots, k\}$ tal que $p_r < r$. Pero entonces se cumple que

$$p_1 \leq p_2 \leq \dots \leq p_{r-1} \leq p_r \leq r - 1,$$

lo que muestra que hay r huertas cuyos plazos son menores o iguales que $r - 1$, y por tanto es imposible recolectar todas las huertas dentro de su plazo, por lo que se concluye que el conjunto H no es factible.

Comprobación de la factibilidad, segunda posibilidad

Lema Un conjunto H de huertas es factible si y solo si la secuencia de las huertas de H donde estas se recolectan “lo más tarde posible”, es admisible.

Comprobación de la factibilidad, segunda posibilidad

Lema Un conjunto H de huertas es factible si y solo si la secuencia de las huertas de H donde estas se recolectan “lo más tarde posible”, es admisible.

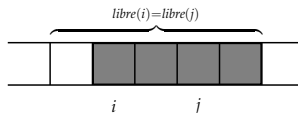
Demostración: Comenzamos formalizando el concepto de “lo más tarde posible”, que se refiere a que para cada huerta se elige el día libre más tardío y que no se pase de plazo; es decir, que para cada huerta h_i el día elegido será

$$d(i) = \text{máx}\{d \mid 1 \leq d \leq \text{mín}\{n, p_i\} \wedge (\forall j : 1 \leq j < i : d \neq d(j))\}.$$

Como en el lema anterior, la implicación hacia la izquierda es trivial y demostraremos la implicación hacia la derecha por contrarrecíproco. Supongamos que existe alguna huerta en H tal que todos los días antes de que expire su plazo p están ocupados. Para $l = \text{mín}\{n, p\}$ sea $r > l$ el primer día libre; entonces en H hay al menos r huertas ($r - 1$ puestas más la que se está intentando colocar) con plazo menor que r , siendo por tanto imposible recolectar todas las huertas de H dentro de su plazo, por lo que el conjunto H no es factible.

Para implementar este test de factibilidad definimos una relación de equivalencia sobre los días de recolección.

Si definimos $\text{libre}(i) = \max\{d \leq i \mid d \text{ libre}\}$, es decir, el primer predecesor libre de i , entonces i y j estarán en la misma clase si y solo si $\text{libre}(i) = \text{libre}(j)$.



Para que $\text{libre}(i)$ esté definido incluso si todos los días están ocupados, se considera también el día 0, que permanecerá siempre libre.

La idea es que cada huerta h_i debería recolectarse en el día $\text{libre}(p_i)$, porque representa el último día libre que respeta su plazo.

$libre(i)$ va cambiando a medida que se va planificando la recolección de las huertas.

Inicialmente, como no se ha planificado ninguna recolección, se tiene $\forall i : 0 \leq i \leq n : libre(i) = i$ (cada día está en una clase de equivalencia diferente).

Si recolectamos la huerta h_i en el día $libre(p_i)$ (siempre que este no sea 0), al ocupar dicho día hay que *unir* la clase de equivalencia a la que pertenece $libre(p_i)$ con la correspondiente al día anterior.

```
struct Huerta {
    unsigned int benef; size_t plazo; size_t id;
};

bool operator>(Huerta const& a, Huerta const& b) {
    return a.benef > b.benef;
}

bool resuelveCaso() { //  $O(N \log N)$ ,  $N$  es el número de huertas
    size_t N;
    cin >> N;
    if (N == 0) return false;

    vector<Huerta> huertas(N);
    for (size_t i = 0; i < N; ++i) {
        cin >> huertas[i].benef >> huertas[i].plazo;
        huertas[i].id = i + 1;
    }
    sort(huertas.begin(), huertas.end(), greater<Huerta>());

    vector<size_t> seleccion = resolver(huertas);
    for (size_t i = 0; i < seleccion.size(); ++i)
        cout << (i > 0 ? " " : "") << seleccion[i];
    cout << "\n";
    return true;
}
```

```
// las huertas están ordenadas de mayor a menor beneficio
vector<size_t> resolver(vector<Huerta> const& huertas) { // O(N)
    size_t N = huertas.size();

    size_t maxp = huertas[0].plazo;
    for (size_t i = 1; i < N; ++i) // buscar plazo máximo
        maxp = max(maxp, huertas[i].plazo);
    maxp = min(N, maxp);

    ConjuntosDisjuntos particion(maxp+1);
    vector<size_t> libre(maxp+1);
    for (size_t i = 0; i <= maxp; ++i)
        libre[i] = i;

    // recorrer las huertas planificándolas lo más tarde posible
    vector<int> aux(maxp+1, -1);
    for (size_t i = 0; i < N; ++i) {
        size_t c1 = particion.buscar(min(maxp, huertas[i].plazo));
        size_t m = libre[c1];
        if (m != 0) {
            aux[m] = huertas[i].id;
            size_t c2 = particion.buscar(m-1);
            particion.unir(c1, c2);
            libre[c1] = libre[c2];
        }
    }
}
```



```
// compactar solución
vector<size_t> sol;
for (size_t i = 1; i <= maxp; ++i) {
    if (aux[i] >= 0) {
        sol.push_back(aux[i]);
    }
}
return sol;
}
```

Problema de la mochila real

Cuando Alí-Babá consigue por fin entrar en la Cueva de los Cuarenta Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce muy bien el peso y valor de cada uno de los objetos en la cueva.

Debido a los peligros que tiene que afrontar en su camino de vuelta, solo puede llevar consigo aquellas riquezas que quepan en su pequeña mochila, que soporta un peso máximo conocido.

Suponiendo que los objetos **son fraccionables**, determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n .

La mochila soporta un peso total máximo $M > 0$.

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n .

La mochila soporta un peso total máximo $M > 0$.

El problema consiste en maximizar

$$\sum_{i=1}^n x_i v_i$$

con la restricción

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

donde x_i es la **fracción** del objeto i tomada, $0 \leq x_i \leq 1$.

Hay n objetos, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para todo i entre 1 y n .

La mochila soporta un peso total máximo $M > 0$.

El problema consiste en maximizar

$$\sum_{i=1}^n x_i v_i$$

con la restricción

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

donde x_i es la **fracción** del objeto i tomada, $0 \leq x_i \leq 1$.

Ya que el caso en el que todos los objetos caben juntos en la mochila no tiene mucho interés, consideraremos el caso en el que $\sum_{i=1}^n p_i > M$.

En ese caso la solución óptima deberá llenar la mochila por completo:

$$\sum_{i=1}^n x_i p_i = M.$$

Problema de la mochila real: ejemplo

$$M = 100, n = 5$$

	1	2	3	4	5
p_i	10	20	30	40	50
v_i	20	30	66	40	60

Seleccionar	x_i					valor
máx v_i	0	0	1	0.5	1	146

Problema de la mochila real: ejemplo

$$M = 100, n = 5$$

	1	2	3	4	5
p_i	10	20	30	40	50
v_i	20	30	66	40	60

Seleccionar	x_i					valor
máx v_i	0	0	1	0.5	1	146
mín p_i	1	1	1	1	0	156

Problema de la mochila real: ejemplo

$$M = 100, n = 5$$

	1	2	3	4	5
p_i	10	20	30	40	50
v_i	20	30	66	40	60
$\frac{v_i}{p_i}$	2	1.5	2.2	1	1.2

Seleccionar	x_i					valor
máx v_i	0	0	1	0.5	1	146
mín p_i	1	1	1	1	0	156
máx $\frac{v_i}{p_i}$	1	1	1	0	0.8	164

Problema de la mochila real: Demostración de optimalidad

Suponemos que los objetos están ordenados en forma decreciente respecto a su valor por unidad de peso:

$$\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \dots \geq \frac{v_n}{p_n},$$

y sea $X = (x_1, x_2, \dots, x_n)$ la solución construida por el algoritmo voraz. Si todos los x_i son 1, la solución es claramente óptima. En caso contrario, sea j el menor índice tal que $x_j < 1$. Debido a como funciona el algoritmo, sabemos que

$$\begin{aligned} x_i &= 1 && \text{si } i < j \\ 0 \leq x_j &< 1 \\ x_i &= 0 && \text{si } i > j. \end{aligned}$$

Comparamos con una solución óptima $Y = (y_1, y_2, \dots, y_n)$. Si no todos los objetos caben, podemos asumir que $\sum_{i=1}^n y_i p_i = M$.

Sea k la menor posición diferente.

Por como funciona el algoritmo, se debe cumplir que $k \leq j$ y por tanto $y_k < x_k$:

- $k < j$, entonces $x_k = 1$, y por tanto, $y_k < x_k$.
- $k = j$, entonces $y = 1$ para $1 \leq i < k$, por lo que $y_k > x_k$ implicaría $\sum_{i=1}^n y_i p_i > M$, lo que es imposible.
- $k > j$ es imposible porque si no $\sum_{i=1}^n y_i p_i > M$.

Modificamos la solución óptima *aumentando* y_k para hacerlo igual a x_k y *decrementando* tantos y_{k+1}, \dots, y_n como haga falta para hacer que la capacidad total usada sea M .

La nueva solución $Z = (z_1, z_2, \dots, z_n)$ cumple que $z_i = x_i$ para $1 \leq i \leq k$ y

$$\sum_{i=k+1}^n p_i(y_i - z_i) = p_k(z_k - y_k).$$

Y Z sigue siendo óptima:

$$\begin{aligned} \sum_{i=1}^n v_i z_i &= \sum_{i=1}^n v_i y_i + v_k(z_k - y_k) - \sum_{i=k+1}^n v_i(y_i - z_i) \\ &= \sum_{i=1}^n v_i y_i + \frac{v_k}{p_k} p_k(z_k - y_k) - \sum_{i=k+1}^n \frac{v_i}{p_i} p_i(y_i - z_i) \\ &\geq \sum_{i=1}^n v_i y_i + \underbrace{\left(p_k(z_k - y_k) - \sum_{i=k+1}^n p_i(y_i - z_i) \right)}_0 \frac{v_k}{p_k} \\ &= \sum_{i=1}^n v_i y_i. \end{aligned}$$

```
void resolver(vector<Objeto> const& objetos, double M,
             vector<double> & sol, double & valor) { //  $O(N \log N)$ 
    size_t N = objetos.size();
    vector<Densidad> D(N);
    for (size_t i = 0; i < N; ++i) {
        D[i].densidad = objetos[i].valor / objetos[i].peso;
        D[i].id = i; // para saber a qué objeto corresponde
    }
    sort(D.begin(), D.end(), greater<Densidad>());

    double peso = 0; valor = 0;
    size_t i;
    for (i = 0; i < N && peso + objetos[D[i].id].peso <= M; ++i) {
        sol[D[i].id] = 1; // el objeto D[i].id cabe completo
        peso += objetos[D[i].id].peso;
        valor += objetos[D[i].id].valor;
    }
    if (i < N) { // partir el objeto D[i].id
        sol[D[i].id] = (M - peso) / objetos[D[i].id].peso;
        valor += sol[D[i].id] * objetos[D[i].id].valor;
    }
}
```