Algoritmos Voraces (1)

Método Voraz

El método voraz construye una solución a través de una secuencia de pasos, hasta que se alcanza una solución completa al problema.

En cada paso, la elección debe ser:

1. Factible, es decir, tiene que satisfacer las restricciones del problema.
2. Óptima localmente, es decir, la mejor opción local entre todas las disponibles en ese paso.
3. Irrevocable, es decir, no se puede cambiar en los pasos posteriores del algoritmo.

Las características generales de los algoritmos voraces son:

1. Para construir la solución se dispone de un **conjunto de candidatos**. Se van formando dos conjuntos: los candidatos **seleccionados** (formarán parte de la solución), y los **rechazados** definitivamente
2. Existe una **función de selección** que indica cuál es el candidato más prometedor de entre los aún no considerados
3. Existe un test de factibilidad que comprueba si un candidato es compatible con la solución parcial construida hasta el momento.
4. Existe un test de solución que determina si una *solución parcial* forma una *solución completa*.
5. Se tiene que obtener una solución óptima según una función objetivo que asocia un valor a cada solución.

```
fun voraz (datos : conjunto):
    var candidatos : conjunto
```

```
    S = ConjuntoVacio {en S se va construyendo la solución}
    candidatos = datos
    while !candidatos.empty() && !esSolucion(S):
        x = seleccionar(candidatos)
        candidatos = candidatos - {x}
        if( esFactible(S+x) ) :
            S = S + x
 return S
```

El coste promedio de un algoritmo voraz es de O(n) si las entradas están ordenadas. Si no, serían O(n log(n)) que es el coste de la ordenación.

Demostración de Corrección

Los algoritmos voraces construyen una solución de manera directa y eficiente, pero necesitan demostrar que la estrategia seguida es correcta, es decir, que siempre encuentra una solución óptima.

El método de reducción de diferencias compara una solución óptima con la solución obtenida por el algoritmo voraz. Si ambas soluciones no son iguales, se a transformando la solución óptima de partida de forma que continue siendo óptima, pero siendo más parecida a la del algoritmo voraz.

Si el resultado de aplicar el método un número finito de veces resulta en que la solución del algoritmo y la solución óptima son idénticos, significará que el algoritmo es correcto.

Ejemplo

En el problema de maximizar el número de ficheros en el disco.

En nuestro algoritmo voraz elegiríamos los ficheros por tamaño (de menor a mayor). Compararíamos la solución del algoritmo con una solución óptima cualquiera (ordenada de menor a mayor que es como estará el conjunto resultante de nuestro algoritmo):

- La solución de nuestro algoritmo es igual a la óptima hasta Fi donde la solución óptima ha cogido el Fj.
- Teniendo j > i ⇒ si ≤ sj (porque el algoritmo va de menor a mayor)
- Entonces deberíamos de cambiar Fj por Fi

- Estas transformaciones se pueden seguir haciendo con los k ficheros de la solución del algoritmo, hasta que la solución óptima sea igual que la del algoritmo

Contraejemplo

Existen ocasiones en las que los algoritmos voraces no son una solución al problema. Para llegar a esta afirmación la mejor forma es con contraejemplos.

Ej. Queremos llenar un disco de memoria D con ficheros y llegar a su máximo.

1. Si el D = 45 y los ficheros son 30, 25 y 20 un algoritmo voraz de mayor a menor cogería el fichero de 30 y esa seria su solución. Sin embargo la solución es coger los de 25 y los de 20.
2. Si el D = 30 y los ficheros son 10, 15, 20 un algoritmo voraz de menor a mayor cogeria el fichero 10 y 15, si embargo la solución real seria 10 y 20.

> El algoritmo voraz solucionaría la maximización del número de ficheros

Problemas

Mínimo teniendo Cota

Problemas en los que tenemos que gastar lo mínimo teniendo una cota máxima. Ejemplos:

- Viajar de Valencia a Lisboa con los mínimos repostajes posibles sabiendo que el coche aguanta k kilómetros.
- Cuantos parches tengo que poner como mínimo sabiendo las dimensiones de los parches.

En los parches (al igual que en el viaje) la solución se basa en cuantos puedo ocupar si lo pongo aquí. Y la pregunta es, La distancia que lleva ocupada el parche sumada a la que esta la actual, ¿ocuparía también este agujero?,

```
int numParches(vector<int> datos, int K, int N) {
    int contador = 1, anterior = datos[0], distActual = 0;
    for (int i = 1; i < N; i++)
    {
        int dis = datos[i] - anterior;
```

```
        if (distActual + dis <= K) {
            distActual += dis;
        }
        else {
            contador++;
            distActual = 0;
        }
        anterior = datos[i];
    }
    return contador;
}
```

Máximo compatibles

Primera Variante

Teniendo dos vectores ordenados, numero de pares compatibles. Ejemplo:

- Teniendo equipos con un numero de soldados, y sabiendo el número de soldados que necesito en una ciudad para que esté protegida, ¿cuantas podría proteger?

La solución pasa por (en dos vectores ordenados de mayor a menor), asignar a cada ciudad el equipo mas compatible. Es decir,
A mi ciudad me vale este:

- No ⇒ paso a la siguiente ciudad, porque yo no puedo ser protegida (el actual equipo es menor que lo que yo necesito, por lo que todos los demás también lo serán)
- Si ⇒ me asigno el equipo y la sig ciudad podrá elegir su siguiente mayor (me asigno el mayor de los equipos, y le paso a la siguiente ciudad que es menor que yo el siguiente mayor equipo)

```
int numVictorias(vector<int> ciudad, vector<int> equipos, int N) {
    int contador = 0, ultimoIndice=0;
    for (int i = 0; i < N; i++)
    {
        if (ciudad[i] <= equipos[ultimoIndice]) {
            ultimoIndice++;
            contador++;
        }
    }
    return contador;
}
```

Segunda Variante

Teniendo dos vectores ordenados, número de pares compatibles, teniendo una cota superior. Ejemplo:

- Tenemos un numero M de camisetas, cada una con una talla, y queremos saber cuantas habrá que comprar, sabiendo las N tallas que necesitamos. Tener en cuenta que una talla es valida para un jugador cuando es igual o es solo una talla más.

La solución, parecida a la anterior. Sin embargo hay que iterar sobre ambos vectores ordenados a la vez:

- La iteración termina cuando alguno de los no tiene más elementos
- Si la camiseta es valida, avanzamos en ambos vectores y asignamos
- Si la camiseta que tenemos es más pequeña que la que necesitamos ⇒ avanzamos en el vector de las que necesitamos, pues no se podrá asignar nunca
- Por el contrario si la que necesitamos es más pequeña que la que tenemos ⇒ avanzamos en el vector de las que tenemos, la siguiente que tenemos es menor que la actual

```
int numNecesitamos(vector<int> necesitamos, vector<int> tenemos, int N, int M)
    int contador = N, n=0, m=0;
    while (m < M && n < N)
    {
        if (tenemos[m] == necesitamos[n] || tenemos[m] == necesitamos[n] + 1) {
            m++; n++;
            contador--;
        }
        else if (tenemos[m] < necesitamos[n]) {
            n++;
        }
        else {
            m++;
        }
    }
    return contador;
}
```

Tiempo en el Sistema Mínimo

- Tenemos n tareas, cada una de las cuales requiere un tiempo de ejecución ti.
- Todas están disponibles para ser ejecutadas, por un único procesador, en secuencia.
- Queremos 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.
- Ti es el tiempo en el sistema de la tarea i, que es la suma de su tiempo de ejecución ti, más los tiempos de ejecución de las tareas que se ejecutan antes de ella.

- El problema consiste en minimizar el tiempo medio del sistema. Pero como n está fijo, esto equivale a minimizar el tiempo total.

> La planificación óptima consiste en atender a las tareas por orden creciente de tiempo de ejecución (FIFO).

Tareas con Plazo y Beneficio

- Tenemos n tareas, cada una requiriendo una unidad de tiempo en ejecutarse, y con un plazo pi y un beneficio bi
- Si la tarea se realiza dentro de plazo, se obtiene beneficio.
- No todas las tareas tiene por qué realizarse. Solamente las que puedan hacerse antes de que venza el plazo.
- El objetivo es planificar las tareas de forma que se maximice el beneficio total obtenido.

Ejemplo: teniendo dos unidades de tiempo y las siguientes tareas:

i	pi	bi
1	2	30
2	1	35
3	2	25
4	1	40

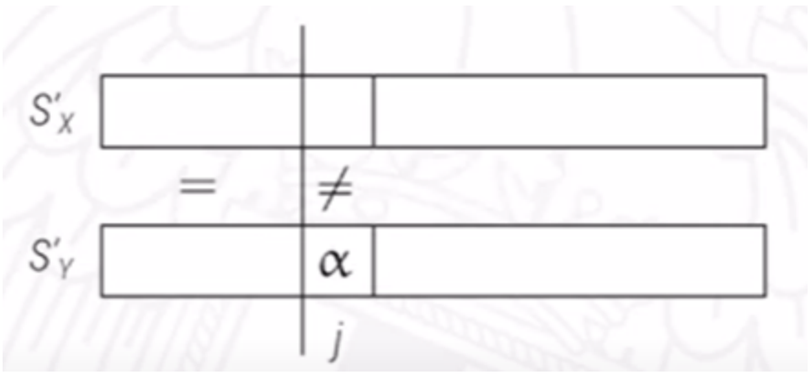
La planificación mas beneficiosa [4, 1]. Un enfoque voraz seria ordenar las tareas por orden no decreciente de beneficios y añadirlas a la planificación si es posible:

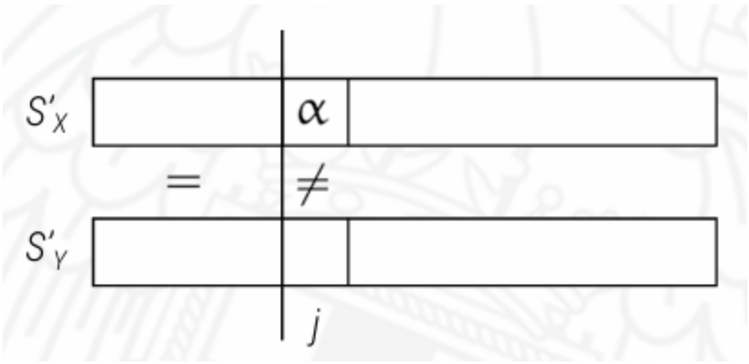
- Un conjunto de tareas es factible si existe alguna secuencia de ejecución admisible, que permita realizar todas las tareas dentro de sus plazos.
- Una permutación (i1, i2, ..., ik) es admisible si
  - Para todo: $1 \leq j \leq k : pij \geq j$
- La estrategia voraz considera las tareas de mayor a menor beneficio, y cada tarea se selecciona si al añadirla al conjunto de tareas seleccionadas, este sigue siendo factible.

Demostración de Optimalidad

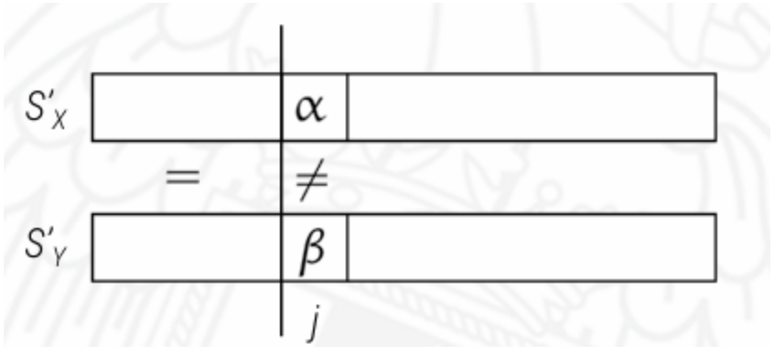
- Llamamos X a la solución voraz, e Y a una solución óptima cualquiera.
- Sean Sx y Sy secuencias admisibles de las tareas de X e Y.
- Primero se transforman las secuencias de forma que las tareas comunes se realicen en el mismo momento.
  - Ej, la tarea h se hace en el momento c de Sx y en el momento j de Sy. Se retrasaría la tarea h en Sx

- Después de esto, este tipo de diferencias no puede existir, donde la tarea alpha este en una solución y no en la otra.





- Tampoco es posible que haya una tarea h en Sx y otra c en Sy en la posición j donde
  - bh > bc ⇒ es imposible porque habria que sustituir h por c y sabemos que Sy es óptima
  - bh < bc ⇒ tampoco, porque la estartegia voraz habria seleccionado c antes que a h
  - bh == bc ⇒ es la unica posibilidad, donde el beneficio de ambas tareas es el mismo



Test de factibilidad 1

- Un conjunto de tareas T es factible si y solo si la secuencia que ordena las tareas por orden creciente de plazos es admisible.
- Sea $T = \{t_1, \ldots, t_k\}$ con $p_1 \leq p_2 \leq \ldots \leq p_k$.
- Si la secuencia $t_1, t_2, \ldots, t_k$ no es admisible, existe alguna tarea t_r tal que $p_r < r$. Pero entonces se cumple

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

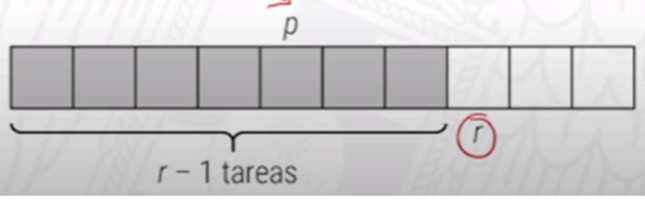
- T no es factible.

Test de factibilidad 2

- Un conjunto de tareas T es factible si y solo si la secuencia que planifica las tareas *lo más tarde posible* es admisible.

$$d(i) = \max\{d \mid 1 \leq d \leq \min(n, p_i) \wedge (\forall j : 1 \leq j < i : d \neq d(j))\}$$

- Tarea en T con plazo p, que al ir a planificarla no "cabe":




```
struct Tarea {
 int plazo;
 int beneficio;
 int id;
 };

 bool operator>(Tarea a, Tarea b) {
    return a.beneficio > b.beneficio;
 }
/*
COSTE
   - inicializaciones ⇒ O(N)
   - recorrido de tareas ⇒ O(N log*N)
   - compactar ⇒ O(N)
*/
// las tareas están ordenadas de mayor a menor beneficio
int resolver(vector<Tarea> const& tareas, vector<int> & sol) {
   //init
```

```
    int N = tareas.size(); // número de tareas
    vector<int> libre(N + 1, 0);

    for (int i = 0; i <= N; ++i)
        libre[i] = i;

    vector<int> plan(N + 1); // 0 es que no está usado
    ConjuntosDisjuntos particion(N + 1);
    int beneficio = 0;

   //recorrer las tareas
   for (int i = 0; i < N; ++i) {
        int c1 = particion.buscar(min(N, tareas[i].plazo)); // busca clase de equiv do
        int m = libre[c1];
        if (m != 0) { // podemos colocar la tarea i
            plan[m] = tareas[i].id;
            beneficio += tareas[i].beneficio;
            int c2 = particion.buscar(m-1);
            particion.unir(c1, c2);
            libre[c1] = libre[c2];
        }
   }
    // compactamos la solución
    for (int i = 1; i <= N; ++i) {
        if (plan[i] > 0)
            sol.push_back(plan[i]);
    }
   return beneficio
}
```