Objetivos

- Resolver problemas mediante el diseño de algoritmos voraces.

Conceptos

El método que produce algoritmos voraces es un método muy sencillo y que puede ser aplicado a numerosos problemas, especialmente los de optimización. Dado un **problema con n entradas**, el método consiste en **obtener un subconjunto de éstas que satisfaga una determinada restricción definida para el problema**. Cada uno de los **subconjuntos que cumplan las restricciones** diremos que son **soluciones prometedoras**. Una **solución prometedora que maximice o minimice una función objetivo** la denominaremos **solución óptima**. Como ayuda para identificar si un problema es susceptible de ser resuelto por un algoritmo voraz vamos a definir una serie de **elementos que han de estar presentes en el problema**:

- Un **conjunto de candidatos**, que corresponden a las n entradas del problema.
- Una **función de selección** que en cada momento determine el candidato idóneo para formar la solución de entre los que aún no han sido seleccionados ni rechazados.
- Una **función** que compruebe si un cierto subconjunto de candidatos **es prometedor**. Entendemos por prometedor **que sea posible seguir añadiendo candidatos y encontrar una solución**.
- Una **función objetivo** que determine el valor de la solución hallada. Es la **función que queremos maximizar o minimizar**.
- Una **función** que compruebe **si un subconjunto de estas entradas es solución al problema**, sea óptima o no.

Con estos elementos, podemos **resumir el funcionamiento** de los algoritmos voraces en los siguientes puntos:

1. Para resolver el problema, un algoritmo voraz tratará de **encontrar un subconjunto de candidatos** tales que, cumpliendo las restricciones del problema, constituya la solución óptima.
2. Para ello **trabajará por etapas**, tomando en cada una de ellas la decisión que le parece la mejor, sin considerar las consecuencias futuras, y por tanto escogerá de entre todos los candidatos el que produce un óptimo local para esa etapa, suponiendo que será a su vez óptimo global para el problema.
3. **Antes de añadir un candidato a la solución** que está construyendo **comprobará si es prometedora** al añadirlo. En caso afirmativo lo incluirá en ella y en caso contrario descartará este candidato para siempre y no volverá a considerarlo.
4. Cada vez que se incluye un candidato **comprobará si** el conjunto obtenido **es solución**.

Resumiendo, los algoritmos voraces construyen la solución en etapas sucesivas, tratando siempre de tomar la decisión óptima para cada etapa. A la vista de todo esto no resulta difícil plantear un **esquema general** para este tipo de algoritmos (el orden de las acciones puede variar según convenga):

```
public void algoritmoVoraz(Conjunto entrada, Conjunto solucion) {
       Elemento x;
       int enc=0;
       prepararSolucion(solucion);
       while (!esVacio(entrada) && (!enc)) {
              x=seleccionarCandidato(entrada);
              if (esPrometedor(x,solucion)) {
                     incluir(x,solucion)
                     if (esSolucion(solucion)) {
                            enc=1;
                     }
              }
       }
}
```

De este esquema se desprende que los algoritmos voraces son muy **fáciles de implementar** y producen **soluciones muy eficientes**. Entonces cabe preguntarse ¿por qué no utilizarlos siempre?

- No todos los problemas admiten esta estrategia de solución.
- La **búsqueda de óptimos locales no tiene por qué conducir siempre a un óptimo global**. La estrategia de los algoritmos voraces consiste en tratar de ganar todas las batallas sin pensar que, como bien saben los estrategas militares y los jugadores de ajedrez, **para ganar la guerra muchas veces es necesario perder alguna batalla**. Y aquí radica la dificultad de estos algoritmos. Encontrar la función de selección que nos garantice que el candidato escogido o rechazado

en un momento determinado es el que ha de formar parte o no de la solución óptima sin posibilidad de reconsiderar dicha decisión.

Debido a su eficiencia, este tipo de algoritmos es **muchas veces utilizado aun en los casos donde se sabe que no necesariamente encuentran la solución óptima**. En algunas ocasiones **la situación nos obliga a encontrar pronto una solución** razonablemente buena, aunque no sea la óptima, puesto que, **si la solución óptima se consigue demasiado tarde, ya no vale para nada** (piénsese en el localizador de un avión de combate, o en los procesos de toma de decisiones de una central nuclear).

Es decir, **la eficiencia de este tipo de algoritmos hace que se utilicen**, aunque no consigan resolver el problema de optimización planteado, sino que **sólo den una solución "aproximada"**.

El nombre de algoritmos voraces, **también conocidos como ávidos** (su nombre original **proviene del término inglés greedy**) se debe a su comportamiento: **en cada etapa "toman lo que pueden" sin analizar consecuencias**, es decir, son **glotones por naturaleza**.

Experimentos

E1. (60 min.). El Problema de la mochila

Se tiene una mochila que es capaz de soportar un peso máximo, así como un conjunto de objetos, cada uno de ellos con un peso y un beneficio, y solo se puede coger como máximo 1 de cada tipo. La solución pasa por conseguir introducir el máximo beneficio en la mochila, eligiendo los objetos adecuados. Establezca un peso máximo de 20 y la siguiente información de pesos y sus respectivos beneficios:

	Obj. 1	Obj. 2	Obj. 3
Peso	10	18	15
Beneficio	15	20	25

Probar en las dos situaciones siguientes:

1. Cada objeto puede tomarse completo o fraccionado.
2. Los objetos solo pueden tomarse completos.

¿Se alcanza la solución óptima en algún caso? ¿A qué es debido?

¿En el caso expuesto, habría algún modo en el que pudiera obtener una solución óptima usando el algoritmo que ha desarrollado?

Problemas

P1. (30 mins.). El cambio de moneda.

Suponiendo que el sistema monetario de un país está formado por monedas de valores v1,v2,...,vn, el problema del cambio de dinero consiste en descomponer cualquier cantidad dada M en monedas de ese país utilizando el menor número posible de monedas.
En primer lugar, es fácil implementar un algoritmo voraz para resolver este problema, siguiendo el proceso que usualmente utilizamos en nuestra vida diaria. Sin embargo, tal algoritmo va a depender del sistema monetario utilizado y por ello vamos a plantearnos dos situaciones para las cuales deseamos conocer si el algoritmo voraz encuentra siempre la solución óptima (consideraremos una cantidad ilimitada de monedas de cada tipo):

1. Disponemos de monedas con valores de 1, 2, 5, 10, 20 y 50 céntimos de euro, 1 y 2 euros (€).
2. Supongamos que tenemos monedas sólo de 100, 90 y 1.

a) Considerar una cantidad ilimitada de monedas de cada tipo.
b) Modificar el algoritmo para tratar la situación en la que no existe un número ilimitado de cada moneda

P2. (30 mins.) El fontanero diligente.

Un fontanero necesita hacer n reparaciones urgentes, y sabe de antemano el tiempo que le va a llevar cada una de ellas: en la tarea i-ésima tardará ti minutos. Como en su empresa le pagan dependiendo de la satisfacción del cliente, necesita decidir el orden en el que atenderá los avisos para minimizar el tiempo medio de espera de los clientes. En otras palabras, si llamamos Ei a lo que espera el cliente i-ésimo hasta ver reparada su avería por completo, necesita minimizar la expresión:

$$E(n)=\sum_{i=1}^{n} E_i$$

P3. (120 min.). El camionero con prisa.

Un camionero conduce desde Bilbao a Málaga siguiendo una ruta dada y llevando un camión que le permite, con el tanque de gasolina lleno, recorrer n kilómetros sin parar. El camionero dispone de un mapa de carreteras que le indica las distancias entre las gasolineras que hay en su ruta. Como va con prisa, el camionero desea pararse a repostar el menor número de veces posible.

Supondremos además que disponemos de un vector con la información que tiene el camionero sobre las distancias entre ellas, de forma que el i-ésimo elemento del vector indica los kilómetros que hay entre las gasolineras i–1 e i. Para que el problema tenga solución hemos de suponer que ningún valor de ese vector es mayor que el número n de kilómetros que el camión puede recorrer sin repostar. Deseamos diseñar un algoritmo voraz para determinar en qué gasolineras tiene que parar.

P4. (120 min.). La asignación de tareas.

Supongamos que disponemos de n trabajadores y n tareas. Sea bij > 0 el coste de asignarle el trabajo j al trabajador i. Una asignación de tareas puede ser expresada como una asignación de los valores 0 ó 1 a las variables xij, donde xij = 0 significa que al trabajador i no le han asignado la tarea j, y xij = 1 indica que sí. Una asignación válida es aquella en la que a cada trabajador sólo le corresponde una tarea y cada tarea está asignada a un trabajador. Dada una asignación válida, definimos el coste de dicha asignación como:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}b_{ij}$$

Diremos que una asignación es óptima si es de mínimo coste. Cara a diseñar un algoritmo voraz para resolver este problema podemos pensar en dos estrategias distintas: asignar cada trabajador la mejor tarea posible, o bien asignar cada tarea al mejor trabajador disponible. Sin embargo, ninguna de las dos estrategias tiene por qué encontrar siempre soluciones óptimas. ¿Es alguna mejor que la otra?

Nota: Este es un problema que aparece con mucha frecuencia, en donde los costes son o bien tarifas (que los trabajadores cobran por cada tarea) o bien tiempos (que tardan en realizarlas). Para implementar ambos algoritmos vamos a definir la matriz de costes (bij), que forma parte de los datos de entrada del problema, y la matriz de asignaciones (xij), que es la que buscamos.

Bibliografía Básica

1. **Documentación Oficial Oracle JAVA**. https://docs.oracle.com/en/java/index.html
2. **Fundamentos de Algoritmia**. G. Brassard, P. Bratley. Prentice Hall.
3. **Introduction to Algorithms**. TH. Cormen, CE. Leiserson, RL. Rivest, C. Stein. MIT Press, 2001.