

2. Algoritmos Voraces (Ávidos)

Los algoritmos voraces se caracterizan por buscar una aproximación rápida hacia la solución de un problema, sin tener en cuenta si la solución obtenida es óptima o no. Por tal motivo, al elegir una alternativa que presenta un camino más rápido hacia la solución o un mayor beneficio inmediato, estos algoritmos no pueden determinar si la alternativa elegida lleva a la solución óptima.

Los algoritmos voraces se utilizan con frecuencia para encontrar soluciones iniciales o aproximadas en problemas de optimización, en los cuales existe un parámetro a maximizar (o minimizar) y una serie de restricciones iniciales. Un problema se puede solucionar por medio de un algoritmo voraz si cuenta con determinadas características:

- Se tiene un conjunto de datos (inicialmente conformado por la entrada del problema) a partir del cual se construirá la solución del problema, una o más restricciones que se deben tener en cuenta al construir la solución y un valor o función objetivo que se trata de alcanzar.
- A partir del conjunto de datos, se puede determinar cuál es el “mejor candidato” inmediato para aproximarse a la solución.
- Una vez seleccionado el mejor candidato, es posible determinar si al tomar este candidato junto con los candidatos seleccionados anteriormente y al aplicar las restricciones del problema, se tiene un camino hacia la solución.

A continuación se presentan algunos ejemplos en los cuales se hace uso de algoritmos voraces para hallar la solución a un problema.

2.1. El problema del cambio de monedas

Dada una cantidad de dinero M y un conjunto d de denominaciones de billetes, desarrollar el algoritmo que calcule el mínimo número de billetes requeridos para completar la cantidad M . Suponga que existe un número infinito de billetes de cada denominación y que existe cambio exacto para M

La solución intuitiva a este problema se puede obtener mediante el siguiente algoritmo voraz: Dado que se quiere minimizar la cantidad de billetes que completan la cantidad M , es lógico que se deberá seleccionar el billete de mayor denominación (que sea menor que M) y restar su valor a la cantidad M . Sólo se debe entregar un billete de menor denominación si el valor de la denominación actual es mayor que M . Si existe cambio exacto, al final del algoritmo el valor de M deberá ser cero.

El algoritmo voraz para solucionar el problema del cambio se muestra a continuación.

```
//Algoritmo para resolver el problema del cambio
//Entrada:
//  m: cantidad de dinero
//  d: arreglo de denominaciones
//  n: numero de denominaciones
//Salida:
//  arreglo s con el numero de billetes de cada
//  denominación
algoritmo cambio(m, d[], n)
  k=1
  //Pre: s[i]=0 para i=1..n
```

```

    mientras k<=nd and m > 0
        si m >= d[k]
            m = m - d[k]
            s[k] = s[k] + 1
        sino
            k = k + 1
        fin_si
    fin_mientras
    retornar s
fin_algoritmo

```

2.2. El problema de la mochila

Dada una mochila con capacidad M y una serie de elementos cada uno con un peso w_i y un valor v_i , desarrollar un algoritmo que determine la combinación de elementos que maximizan el valor almacenado en la mochila.

Este problema puede ser resuelto mediante un algoritmo voraz de diferentes formas:

- Voraz por valor: Colocar en la mochila los elementos de mayor valor, sin importar su peso.
- Voraz por peso: Colocar en la mochila los elementos de menor peso, con el fin de maximizar el número de elementos almacenados.

Es sencillo verificar que ninguna de estas dos estrategias ofrece una solución aceptable. Si se toman primero los elementos de mayor valor, también es posible que estos elementos sean más pesados, con lo cual se disminuye la capacidad disponible para los demás elementos. Por otro lado, si se eligen primero los elementos con menor peso, es probable que estos elementos también sean los de menor valor.

Entonces, un algoritmo voraz que se acerca mejor a la solución óptima consiste en elegir los elementos que posean una mejor relación valor/peso. Con ello, si dos o más elementos tienen igual valor, se elegirá primero aquel que tenga menor peso. Tomando como base la solución obtenida en el problema del cambio:

```

// Algoritmo para solucionar el problema de la mochila
// Entrada:
// m: capacidad de la mochila
// v: Arreglo que contiene los valores de cada item
// w: Arreglo que contiene los pesos de cada item
// Salida:
// s: Arreglo que contiene el numero de items
// que se colocan en la mochila.

algoritmo mochila (m, v[], w[], n) {

    // vp: Arreglo que contiene la relacion peso / valor (v[i]/w[i])

    // Pre: vp[i] = v[i]/wi, para i=1..n
    // Pre: vp[i] >= vp[i-1], para i=2..n

    i=1
    total = 0

```

```

//s: arreglo que contiene el numero de items que se
// colocan en la mochila

// Pre: s[i]=0, para i=1..n

mientras m > 0 and i <= n
    si m>= vp[i] //La mochila tiene capacidad para este item
        m = m - vp[i].peso
        s[i]=s[i]+1
        total = total + vp[i]
    sino
        i = i + 1
    fin_si
fin_mientras
retornar s
fin algoritmo

```

Sin embargo, el algoritmo anterior supone que se tiene una cantidad infinita de cada item (problema del cambio). Para la solución del problema de la mochila se debe tener en cuenta que se tiene **un solo ítem de cada clase**, con lo cual se debe modificar el algoritmo base de la siguiente forma:

```

....

mientras m > 0 and i <= n
    //Validar si la mochila tiene capacidad para este item
    si m>= vp[i] //Si hay capacidad, incluir el item
        m = m - vp[i].peso
        s[i]=s[i]+1
        total = total + vp[i]
    fin_si
    i = i + 1 //Pasar al siguiente item
fin_mientras

....

```

No obstante, esta mejora, al final del algoritmo es posible que la mochila aún tenga capacidad disponible. Se deja como ejercicio que se desarrolle un algoritmo en el cual se puedan incluir porciones (fracciones) de los ítems.