GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: MOCHILA 01(KNAPSACK 01)

---

1. Introducción

Dados n elementos e_1 , e_2 , ..., e_n con pesos p_1 , p_2 , ..., p_n y beneficios b_1 , b_2 , ..., b_n , y dada una mochila capaz de albergar hasta un máximo de peso M (capacidad de la mochila), queremos encontrar las proporciones de los n elementos x_1 , x_2 , ..., x_n ($0 \leq x_i \leq 1$) que tenemos que introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima. Esto es, hay que encontrar valores (x_1 , x_2 , ..., x_n) de forma que se maximice la: $\Sigma_{i=1}^{n} b_i * x_i$, sujeta a la restricción $\Sigma_{i=1}^{n} p_i * x_i \leq M$.

2. Conocimientos previos

2.1. Programación Dinámica

Es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.

La programación dinámica toma normalmente uno de los dos siguientes enfoques:

1. **Top-down:** El problema se divide en subproblemas, y estos se resuelven recordando las soluciones por si fueran necesarias nuevamente. Es una combinación de memoización y recursión.
2. **Bottom-up:** Todos los problemas que puedan ser necesarios se resuelven de antemano y después se usan para resolver las soluciones a problemas mayores. Este enfoque es ligeramente mejor en consumo de espacio y llamadas a funciones, pero a veces resulta poco intuitivo encontrar todos los subproblemas necesarios para resolver un problema dado.

3. Desarrollo

Para encontrar un algoritmo de Programación Dinámica que lo resuelva, primero hemos de plantear el problema como una secuencia de decisiones que verifique el principio de óptimo. De aquí seremos capaces de deducir una expresión recursiva de la solución. Por último habrá que encontrar una estructura de datos adecuada que permita la reutilización de los cálculos de la ecuación en recurrencia, consiguiendo una complejidad mejor que la del algoritmo puramente recursivo.

Siendo M la capacidad de la mochila y disponiendo de n elementos, llamaremos V(i,p) al valor máximo de la mochila con capacidad p cuando consideramos i objetos, con $0 \leq p \leq M$ y $1 \leq i \leq n$. La solución viene dada por el valor de V(n,M). Denominaremos d_1 , d_2 , ..., d_n a la secuencia de decisiones que conducen a obtener V(n,M), donde cada d_i podrá tomar uno de los valores 1 ó 0, dependiendo si se introduce o no el i-ésimo elemento. Podemos tener por tanto dos situaciones distintas:

- Que $d_n = 1$. La subsecuencia de decisiones d_1 , d_2 , ..., d_{n-1} ha de ser también óptima para el problema V(n–1,M–p_n), ya que si no lo fuera y existiera otra subsecuencia e_1 , e_2 , ..., e_{n-1}

óptima, la secuencia e_1 , e_2 , ..., e_{n-1} , d_n también sería óptima para el problema V(n,M) lo que contradice la hipótesis.

- Que $d_n = 0$. Entonces la subsecuencia decisiones d_1 , d_2 , ..., d_{n-1} ha de ser también óptima para el problema V(n–1,M).

Podemos aplicar por tanto el principio de óptimo para formular la relación en recurrencia. Teniendo en cuenta que en la mochila no puede introducirse una fracción del elemento sino que el elemento i se introduce o no se introduce, en una situación cualquiera V(i,p) tomará el valor mayor entre V(i–1,p), que indica que el elemento i no se introduce, y V(i–1,p–p i)+b_i , que es el resultado de introducirlo y de ahí que la capacidad ha de disminuir en p_i y el valor aumentar en b_i , y por tanto podemos plantear la solución al problema mediante la siguiente ecuación:

$$V(i,p) = \begin{cases} 0 & \text{si } i = 0 \text{ y } p \geq 0 \\ -\infty & \text{si } p < 0 \\ Max\{V(i-1,p), V(i-1,p-p_i)+b_i\} & \text{en otro caso.} \end{cases}$$

4. Implementación

4.1. C++

```
#include <stdio.h>
#include <iostream>
#include <algorithm>
using namespace std;
int n, capacity;
int Weight[101], Value[101], Sol[1001];

inline int Knapsack01(){
   for (int i=0;i<=capacity;i++) Sol[i] = 0;
   for(int i=0;i<n;i++){
      for(int j=capacity;j>=1;j--){
         if(Weight[i] <= j){
            int x = Sol[j];
            int y = Sol[j-Weight[i]]+Value[i];
            Sol[j] = max(x,y);
         }
      }
    }
    return Sol[capacity];
}

int main(){
   n = 4, capacity = 6;
```

```
    Weight[0] = 1, Value[0] = 4;
    Weight[1] = 2, Value[1] = 6;
    Weight[2] = 3, Value[2] = 12;
    Weight[3] = 2, Value[3] = 7;
    printf(" %d\n",Knapsack01());
    return 0;
}
```

4.2. Java

```
public int knapSack(int W, int wt[], int val[], int n) {
    int[] dp = new int[W + 1];
    for(int i=1;i<n+1;i++){
        for(int w=W;w>=0;w--){
            if(wt[i-1]<=w){
                dp[w]=Math.max(dp[w],dp[w-wt[i-1]]+val[i-1]);
            }
        }
     }
     return dp[W];
}
```

5. Complejidad

La complejidad del algoritmo viene determinada por la construcción de una tabla de dimensiones n*M y por tanto su tiempo de ejecución es de orden de complejidad O(nM). Donde n es la cantidad de elementos disponibles y M la capacidad máxima de la mochila

6. Aplicaciones

Evidentemente este algoritmo permite conformar un subconjunto de un conjunto inicial de elementos donde tomar un elemento nos provoca un costo y a la vez un beneficio. El algoritmo funciona bajo el principio que el costo total que es la sumatoria de los costos de los elementos seleccionado no sobrepase un valor determinado pero siempre generando la máxima ganacia posible.

Es importante resaltar que cada elemento solo puede ser tomado una vez como parte de la solución una vez de ahí que parte del nombre este incluido el 01 donde 0 significa que el elemento no es tomado en la solución mientras 1 si lo es.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando este algoritmo:

- DMOJ - Prince
- DMOJ - Brazalete de Dijes