GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: MOCHILA ILIMITADA (*UNBOUNDED KNAPSACK*)

---

1. Introducción

El problema de la mochila sin límites, también conocido como *Unbounded Knapsack* en inglés, es un problema clásico de optimización combinatoria. En este problema, se busca determinar la forma óptima de llenar una mochila con capacidad limitada (W), maximizando el valor total de los objetos que se pueden colocar en ella.

A diferencia del problema de la mochila $0/1$, en el cual cada objeto puede ser seleccionado una sola vez, en el problema de la mochila sin límites, se permite seleccionar un número ilimitado de veces cada objeto.

2. Conocimientos previos

2.1. Mochila 0/1 (*Knapsack 0/1*)

El problema de la mochila $0/1$, también conocido como *Knapsack 0/1* en inglés, es otro problema clásico de optimización combinatoria. En este problema, se busca determinar la forma óptima de llenar una mochila con capacidad limitada, maximizando el valor total de los objetos que se pueden colocar en ella, teniendo en cuenta que cada objeto se puede seleccionar una sola vez.

2.2. Memorización

En Informática, el término memorización (del inglés memoization) es una técnica de optimización que se usa principalmente para acelerar los tiempos de cálculo, almacenando los resultados de la llamada a una subrutina en una memoria intermedia o búfer y devolviendo esos mismos valores cuando se llame de nuevo a la subrutina o función con los mismos parámetros de entrada.

2.3. Programación Dinámica

La Programación Dinámica la cual es una técnica que combina la corrección de la búsqueda completa y la eficiencia de los algoritmos golosos.

3. Desarrollo

Este problema se basa en el de la Mochila $0/1$ pero en vez de existir n objetos distintos, de lo que disponemos es de n tipos de objetos distintos. Con esto, de un objeto cualquiera podemos escoger tantas unidades como deseemos.

Este problema se puede formular también como una modificación al problema de la Mochila $0/1$, en donde sustituimos el requerimiento de que x_i=0 ó x_i=1, por el que x_i sean números naturales. Como en el problema original, deseamos maximizar la suma de los beneficios de los elementos introducidos, sujeta a la restricción de que éstos no superen la capacidad de la mochila.

Para encontrar un algoritmo de Programación Dinámica que resuelva el problema, primero hemos de plantearlo como una secuencia de decisiones que verifiquen el principio del 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.

3.1. Enfoque recursivo

Una solución simple es considerar todos los subconjuntos de artículos y calcular el peso y el valor total de todos los subconjuntos. Considere los únicos subconjuntos cuyo peso total es menor que W. De todos esos subconjuntos, elija el subconjunto de valor máximo.

Subestructura óptima: para considerar todos los subconjuntos de elementos, puede haber dos casos para cada elemento.

- Caso 1: El artículo está incluido en el subconjunto óptimo.
- Caso 2: El artículo no está incluido en el conjunto óptimo.

Por lo tanto, el valor máximo que se puede obtener de n elementos es el máximo de los dos valores siguientes.

Valor máximo obtenido por $n-1$ artículos y peso W (excluyendo el enésimo artículo). Valor del enésimo artículo más el valor máximo obtenido por n (debido a la oferta infinita) artículos y W menos el peso del enésimo artículo (incluido el enésimo artículo). Si el peso del enésimo elemento es mayor que W, entonces el enésimo elemento no se puede incluir y el Caso 1 es la única posibilidad.

Con esto en mente, llamaremos $V(i,p)$ al valor máximo de una mochila de capacidad p y con i tipos de objetos. Iremos decidiendo en cada paso si introducimos o no un objeto de tipo i. Por consiguiente, para calcular $V(i,p)$ existen dos opciones en cada paso:

- No introducir ninguna unidad del tipo i, con lo cual el valor de la mochila $V(i,p)$ es el calculado para $V(i\breve{}1,p)$.
- Introducir una unidad más del objeto i lo cual indica que el valor de $V(i,p)$ será el resultado obtenido para $V(i,p-p_i)$ más el valor del objeto v_i , con lo cual se verifica que $V(i,p) = V(i,p-p_i\,)+b_i$.

Esto nos permite establecer la siguiente relación en recurrencia para $V(i,p)$:

$$V(i,p) = \begin{cases} 0 & p = 0 \\ (p \div p_i)b_i & i = 1 \\ V(i-1,p) & p < p_i \\ \text{máx}\{V(i-1,p), V(i,p-p_i)+b_i\} & \text{en otro caso} \end{cases}$$

3.2. Memorización

Al igual que otros problemas típicos de programación dinámica (DP), se puede evitar volver a calcular los mismos subproblemas construyendo una matriz temporal $K[][]$ de manera ascendente en la cual se guarde los resultados de los subproblemas.

3.3. Programación Dinámica

Es un problema de mochila ilimitado ya que podemos usar 1 o más instancias de cualquier recurso. Se puede usar una matriz unidimensional simple, digamos $dp[W+1]$, de modo que $dp[i]$ almacene el valor máximo que se puede lograr usando todos los elementos y la capacidad de la mochila. Tenga en cuenta que aquí usamos una matriz unidimensional, que es diferente de la mochila clásica donde usamos una matriz bidimensional. Aquí el número de elementos nunca cambia. Siempre tenemos todos los artículos disponibles. Podemos calcular recursivamente $dp[]$ usando la siguiente fórmula: $dp[i] = 0$, $dp[i] = \max(dp[i], dp[i - wt[j]] + val[j])$ donde j varía de 0 a $n - 1$ tal que $peso[j] \leq i$. De esta forma la respuesta queda en $d[W]$.

3.4. Enfoque eficiente

El enfoque anterior se puede optimizar en función de las siguientes observaciones:

1. Supongamos que el índice i nos da el valor máximo por unidad de peso en los datos dados, que se puede encontrar fácilmente en $O(n)$.
2. Para cualquier peso X, mayor o igual a $wt[i]$, el valor máximo alcanzable será $dp[X - wt[i]] + val[i]$.
3. Podemos calcular los valores de $dp[]$ de 0 a $wt[i]$ usando el algoritmo tradicional y también podemos calcular el número de instancias del i-ésimo elemento que podemos incluir en el peso W.
4. Entonces la respuesta requerida será $val[i] * (W/wt[i]) + dp[W \,\%wt[i]]$.

4. Implementación

4.1. C++

4.1.1. Enfoque recursivo

```
int unboundedKnapsack(int W, int wt[], int val[], int idx){
   if (idx == 0) return (W / wt[0]) * val[0];
   int notTake = 0 + unboundedKnapsack(W,wt,val,idx-1);
   int take = INT_MIN;
   if (wt[idx] <= W) take = val[idx]+unboundedKnapsack(W-wt[idx],wt,val,idx);
   return max(take, notTake);
}
```

4.1.2. Memorización

```
int unboundedKnapsackMemo(int W, int wt[], int val[], int idx,vector<vector<
    int> >& dp){
   if (idx == 0) return (W / wt[0]) * val[0];
   if (dp[idx][W] != -1) return dp[idx][W];
   int notTake = 0 + unboundedKnapsackMemo(W, wt, val, idx - 1, dp);
   int take = INT_MIN;
   if (wt[idx] <= W)
      take = val[idx] + unboundedKnapsackMemo(W-wt[idx],wt,val,idx,dp);

    return dp[idx][W] = max(take, notTake);
}

int unboundedKnapsack(int W, int wt[], int val[]){
   int n = sizeof(val) / sizeof(val[0]);
   vector<vector<int> > dp(n, vector<int>(W + 1, -1));
   return unboundedKnapsackRecursive(W, wt, val, n - 1, dp);
}
```

4.1.3. Programación Dinámica

```
int unboundedKnapsackDP(int W, int val[], int wt[]){
   int n = sizeof(val)/sizeof(val[0]);
   vector<int> dp(W+1,0);
   for (int i=0; i<=W; i++)
      for (int j=0; j<n; j++)
        if (wt[j] <= i)
           dp[i] = max(dp[i], dp[i-wt[j]] + val[j]);
   return dp[W];
}
```

4.1.4. Enfoque eficiente

```
int unboundedKnapsackBetter(int W, vector<int> val, vector<int> wt){
   int maxDenseIndex = 0;

   for (int i = 1; i < val.size(); i++) {
      if (val[i] / wt[i] > val[maxDenseIndex] / wt[maxDenseIndex] ||
         (val[i] / wt[i] == val[maxDenseIndex] / wt[maxDenseIndex] &&
         wt[i] < wt[maxDenseIndex])) {
         maxDenseIndex = i;
      }
   }

   int dp[W + 1] = { 0 };
   int counter = 0;
```

```
   bool breaked = false;
   int i = 0;
   for (i = 0; i <= W; i++) {
      for (int j = 0; j < wt.size(); j++) {
         if (wt[j] <= i) {
            dp[i] = max(dp[i], dp[i - wt[j]] + val[j]);
         }
      }
      if (i - wt[maxDenseIndex] >= 0 &&
          dp[i] - dp[i - wt[maxDenseIndex]] == val[maxDenseIndex]) {
         counter += 1;
         if (counter >= wt[maxDenseIndex]) {
            breaked = true; break;
         }
      }
      else counter = 0;
   }

   if (!breaked) return dp[W];
   else {
      int start = i - wt[maxDenseIndex] + 1;
      int times = (floor)((W - start) / wt[maxDenseIndex]);
      int index = (W - start) % wt[maxDenseIndex] + start;
      return (times * val[maxDenseIndex] + dp[index]);
   }
}
```

4.2. Java

4.2.1. Enfoque recursivo

```
static int unboundedKnapsack(int W, int wt[], int val[], int idx){
   if (idx == 0) return (W / wt[0]) * val[0];
   int notTake = 0 + unboundedKnapsack(W, wt, val, idx - 1);
   int take = Integer.MIN_VALUE;
   if (wt[idx] <= W) take = val[idx]+unboundedKnapsack(W-wt[idx],wt,val,idx);
   return Math.max(take, notTake);
}
```

4.2.2. Memorización

```
public static int unboundedKnapsackRecursive(int W,int wt[],int val[],int idx,
    int dp[][]){
   if (idx == 0) return (W / wt[0]) * val[0];

   if (dp[idx][W] != -1) return dp[idx][W];
```

```
    int notTake = 0 + unboundedKnapsackMemo(W,wt,val,idx-1,dp);
    int take = Integer.MIN_VALUE;
    if (wt[idx] <= W)
        take = val[idx] + unboundedKnapsackMemo(W-wt[idx],wt,val,idx,dp);

    return dp[idx][W] = Math.max(take, notTake);
}

public static int unboundedKnapsack(int W, int wt[], int val[]){
    int n = val.length;
    int[][] dp = new int[n][W + 1];
    for (int row[] : dp) Arrays.fill(row, -1);
    return unboundedKnapsackMemo(W,wt,val,n-1,dp);
}
```

4.2.3. Programación Dinámica

```
public static int unboundedKnapsackDP(int W,int[] val, int[] wt){
    int n = val.length;
    int dp[] = new int[W + 1];

    for(int i = 0; i <= W; i++){
        for(int j = 0; j < n; j++){
            if(wt[j] <= i)
                dp[i] = Math.max(dp[i],dp[i-wt[j]]+val[j]);
        }
    }
    return dp[W];
}
```

4.2.4. Enfoque eficiente

```
public static int unboundedKnapsackBetter(int W,int[] val,int[] wt){
    int maxDenseIndex = 0;
    for (int i = 1; i < val.length; i++) {
        if (val[i] / wt[i] > val[maxDenseIndex] / wt[maxDenseIndex] ||
            (val[i] / wt[i] == val[maxDenseIndex] / wt[maxDenseIndex] &&
            wt[i] < wt[maxDenseIndex])) {
            maxDenseIndex = i;
        }
    }

    int[] dp = new int[W + 1];
    int counter = 0;
    boolean breaked = false;
    int i = 0;
```

```
    for (i = 0; i <= W; i++) {
        for (int j = 0; j < wt.length; j++) {
            if (wt[j] <= i) {
                dp[i] = Math.max(dp[i], dp[i - wt[j]] + val[j]);
            }
        }
        if (i - wt[maxDenseIndex] >= 0 && dp[i] - dp[i - wt[maxDenseIndex]] ==
            val[maxDenseIndex]){
            counter += 1;
            if (counter >= wt[maxDenseIndex]) {
                breaked = true; break;
            }
        }
        else {
            counter = 0;
        }
    }

    if (!breaked) return dp[W];
    else {
        int start = i - wt[maxDenseIndex] + 1;
        int times = (int)((W - start) / wt[maxDenseIndex]);
        int index = (W - start) % wt[maxDenseIndex] + start;
            return (times * val[maxDenseIndex] + dp[index]);
    }
}
```

5. Aplicaciones

Este es de los problemas de la programación dinámica, uno de los denomindados como clásicos. Aunque los problemas de tipo programación dinámica son muy popular con una alta frecuencia de aparición en concursos de programación recientes, los problemas clásicos de programación dinámica en su forma pura (como el presentado en esta guía) por lo general ya no aparecen en los IOI o ICPC modernos. A pesar de esto es necesario su estudio ya que nos permite entender la programación dinámica y como poder resolver aquellos problema de programación dinámica clasificados como no-clásicos e incluso nos permite desarrollar nuestras habilidades de programación dinámica en el proceso.

El problema de la mochila sin límites tiene diversas aplicaciones en la vida real, entre las cuales se incluyen:

1. **Gestión de inventario:** La solución de Unbounded Knapsack se puede utilizar para optimizar la gestión de inventario en empresas. Por ejemplo, si una tienda tiene varios productos con diferentes valores y cantidades disponibles, se puede utilizar este algoritmo para determinar la combinación óptima de productos que generen el mayor beneficio.

2. **Planificación de proyectos:** En la gestión de proyectos, es común tener recursos limitados

y tareas que requieren diferentes cantidades de esos recursos. *Unbounded Knapsack* puede ayudar a determinar la asignación óptima de recursos a las tareas para maximizar el valor del proyecto.

3. **Optimización de publicidad en línea:** En la publicidad en línea, los anunciantes a menudo tienen un presupuesto limitado y diferentes opciones de anuncios con diferentes tasas de conversión y costos. *Unbounded Knapsack* se puede utilizar para seleccionar la combinación de anuncios que maximice el retorno de la inversión dentro del presupuesto asignado.

4. **Selección de cartera de inversiones:** En el ámbito de las finanzas, los inversores a menudo necesitan seleccionar una cartera de inversiones que maximice el rendimiento esperado dado un conjunto de activos disponibles. *Unbounded Knapsack* puede ayudar a determinar la combinación óptima de activos para maximizar el rendimiento esperado.

Estas son solo algunas de las aplicaciones comunes de *Unbounded Knapsack*, pero existen muchas otras áreas donde este problema de optimización puede ser utilizado para mejorar la eficiencia y la toma de decisiones.

6. Complejidad

Veamos un análisis tanto temporal como espacial de los diferentes enfoques analizados para resolver el problema:

	Complejidad temporal	**Complejidad espacial**
Enfoque recursivo	La complejidad temporal de este enfoque recursivo es exponencial $O(2^W)$ ya que existen un casos de subproblemas superpuestos	No se utiliza ningún espacio externo para almacenar valores aparte del espacio de la pila interna por tanto $O(1)$
Memorización	En el peor de los casos se tendría que calcular todos los posibles estados que sería como recorrer todas las posiciones de la matriz por lo que la complejidad es $O(W \times n)$ donde W es el peso máximo de la mochila y n la cantidad de diferentes de tipos objetos	Es evidente que el uso de una matriz para almacenar los cálculos de los diferentes estados hace que la complejidad espacial para esta variante es $O(W \times n)$

Programación Dinámica	Es evidente que se calcula todos los posibles estados que sería como recorrer todas las posiciones de la matriz por lo que la complejidad es $O(W \times n)$ donde W es el peso máximo de la mochila y n la cantidad de diferentes de tipos objetos	El uso de un arreglo para almacenar lo mejor para cada posible peso de 1 a W hace que esta complejidad sea $O(W)$
Enfoque eficiente	Esta optimización o enfoque mas eficiente hace que la complejidad sea igual $O(N + \text{mín}(wt[i], W) \times N)$	El uso de un arreglo para almacenar lo mejor para cada posible peso de 1 a W hace que esta complejidad sea $O(W)$

7. Ejercicios

A continuación una lista de ejercicios que su solución se basan en la utilización del algoritmo de mochila ilimitada

- DMOJ - Comprando Heno