GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: CAMBIO DE MONEDAS (*COIN CHANGE*)

1. Introducción

Cuando no dirigimos hacia un cajero automático y solicitamos un cantidad N el cajero puede que nos de dos variantes de respuesta, la primera es que no nos puede entregar la cantidad N solicitada por que no tiene como dar esa cantidad bien sea porque con la denominaciones de los billetes con que cuenta el cajero no puede conformar la cantidad requerida o N es tan grande que con todo los billetes disponibles en el cajero no alcanza la cifra solicitada. La segunda es una cantidad de billetes que sumados dan la cantidad N solicitada inicialmente.

Ahora vamos a pensar en un cajero ideal el cual no tendrá nunca cola, ni estará fuera de servicio y además contará con un sistema de impresión de billetes al instante que utilizará en caso que necesite generar un billete de un determinado valor por lo que nunca se quedará sin billete de cualquier valor.

Con este nuevo cajero a pesar de todas sus bondades nos surgen dos interrogantes:

- Dado un valor de N y conocidas la denominaciones de los billetes el cajero podrá devolver esa cantidad N y poder de cuantas formas podrá hacerlo ?.
- Dado un valor de N y conocidas la denominaciones de los billetes cual es la cantidad de mínima de billetes que se necesita para obtener el valor de N ?.

A como resolver estas interrogantes esta enfocada la siguiente guía de aprendizaje.

2. Conocimientos previos

2.1. Recursivad

La recursividad es una técnica de programación que se utiliza para realizar una llamada a una función desde ella misma, de allí su nombre. Un algoritmo recursivo es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente.

2.2. Algoritmos golosos (*Greedy*)

Un algoritmo goloso construye una solución al problema al tomar siempre una decisión que se ve mejor en este momento. Un algoritmo goloso nunca recupera sus opciones, pero construye directamente la solución final. Por esta razón, los algoritmos golosos suelen ser muy eficientes.

La dificultad para diseñar algoritmos golosos es encontrar una estrategia codiciosa que siempre produzca una solución óptima al problema. Las opciones localmente óptimas en un algoritmo goloso también deben ser globalmente óptimos. A menudo es difícil argumentar que funciona un algoritmo goloso.

2.3. Memorización

La memorización es una técnica de optimización utilizada principalmente para hacer que los programas informáticos sean más rápidos al almacenar los resultados de las llamadas a funciones en la memoria caché y devolver los resultados almacenados en la memoria caché la próxima vez que se necesiten en lugar de volver a calcularlos.

En el caso de la programación dinámica consiste en almacenar las soluciones de los subproblemas en alguna estructura de datos para reutilizarlas posteriormente. De esa forma, se consigue un algoritmo más eficiente que la fuerza bruta, que resuelve el mismo subproblema una y otra vez.

3. Desarrollo

Antes de adentrarnos en como solucionar cada interrogante vamos a dejar claro cual es la situación inicial que para ambos casos es la misma y solo cambia lo que se quiere hallar.

Se cuenta con k denominaciones de billetes o monedas ($c_1, c_2, \ldots, c_k$) cuya cantidad de cada denominación es ilimitada y un valor N.

3.1. Mínima cantidad de billetes que sumen N

Este caso nuestra tarea es formar la suma N usando la menor cantidad de monedas posible.

Podemos resolver el problema usando un algoritmo goloso (greedy) que siempre elige la moneda más grande posible. El algoritmo goloso (greedy) no siempre no produce necesariamente una solución óptima sobre todo para algunos casos con ciertas denominaciones de monedas.

Ahora es el momento de resolver el problema de manera eficiente utilizando la programación dinámica, de modo que el algoritmo funcione para cualquier juego de monedas. El algoritmo de programación dinámica se basa en una función recursiva que pasa por todas las posibilidades de cómo formar la suma, como un algoritmo de fuerza bruta. Sin embargo, el algoritmo de programación dinámica es eficiente porque utiliza la memorización y calcula la respuesta a cada subproblema una sola vez.

3.1.1. Formulación recursiva

La idea en la programación dinámica es formular el problema recursivamente para que la solución al problema pueda calcularse a partir de soluciones a subproblemas más pequeños.

Entonces $solve(x)$ calcula el número mínimo de monedas necesarias para una suma x. Los valores de la función dependen de las denominaciones de las monedas. Por ejemplo, si las denominaciones de las monedas fueran = 1, 3, 4, los primeros valores de la función son los siguientes:

$$
\begin{aligned}
solve(0) &= 0 \\
solve(1) &= 1 \\
solve(2) &= 2 \\
solve(3) &= 1 \\
solve(4) &= 1 \\
solve(5) &= 2 \\
solve(6) &= 2 \\
solve(7) &= 2 \\
solve(8) &= 2 \\
solve(9) &= 3 \\
solve(10) &= 3
\end{aligned}
$$

Por ejemplo, $solve(10) = 3$, porque se necesitan al menos 3 monedas para formar la suma 10. La solución óptima es 3 + 3 + 4 = 10 respuesta muy diferente a la dada por un algoritmo goloso en la misma situación la cual sería 4 ya que escogiendo la moneda más grande posible la solución sería 4 + 4 +1 +1.

La propiedad esencial de $solve$ es que sus valores se pueden calcular recursivamente a partir de sus valores más pequeños. La idea es centrarnos en la primera moneda que elegimos para la suma. Por ejemplo, en el escenario anterior, la primera moneda puede ser 1, 3 o 4. Si primero elegimos la moneda 1, la tarea restante es formar la suma 9 usando el número mínimo de monedas, que es un subproblema del problema original. Por supuesto, lo mismo se aplica a las monedas 3 y 4. Por lo tanto, podemos usar lo siguiente fórmula recursiva para calcular el número mínimo de monedas:

$$
\begin{aligned}
solve(x) = \text{mín} \quad & (solve(x-1)+1, \\
& solve(x-3)+1, \\
& solve(x-4)+1)
\end{aligned}
$$

El caso base de la recursión es $solve(0) = 0$, porque no se necesitan monedas para formar una suma vacía. Por ejemplo,

$$
solve(10) = solve(7) + 1 = solve(4) + 2 = solve(0) + 3 = 3.
$$

Ahora estamos listos para dar una función recursiva general que calcule el número mínimo de monedas necesarias para formar una suma x:

$$
solve(x) = \begin{cases}
\infty & x < 0 \\
0 & x = 0 \\
\text{mín}_{c \in monedas}\, solve(x-c) + 1 & x > 0
\end{cases}
$$

Primero, si $x < 0$, el valor es ∞ , porque es imposible formar una suma negativa de dinero. Entonces, si $x = 0$, el valor es 0, porque no se necesitan monedas para formar una suma vacía.

Finalmente, si $x > 0$, la variable c pasa por todas las posibilidades de elegir la primera moneda de la suma.

Aún así, esta función no es eficiente, porque puede haber una exponencial número de maneras de construir la suma. Sin embargo, a continuación veremos cómo hacer el función eficiente usando una técnica llamada **memorización**.

3.1.2. Usando memorización

La idea de la programación dinámica es usar la memorización para calcular de manera eficiente los valores de una función recursiva. Esto significa que los valores de la función se almacenan en una matriz después de calcularlos. Para cada parámetro, el valor de la función se calcula recursivamente solo una vez y, después de esto, el valor se puede recuperar directamente de la matriz.

En este problema, usamos arreglos donde $ready[x]$ indica si el valor de $solve(x)$ ha sido calculado, y si es así, $value[x]$ contiene este valor. La constante N ha sido elegida para que todos los valores requeridos quepan en los arreglos.

La función maneja los casos base $x < 0$ y $x = 0$ como antes. Luego, la función verifica desde $ready[x]$ es verdadero si lo es implica que $solve(x)$ ya se ha almacenado en el $value[x]$, y si es así, la función lo devuelve directamente. De lo contrario, la función calcula el valor de $solve(x)$ de forma recursiva y lo almacena en el $value[x]$.

Esta función funciona de manera eficiente, porque la respuesta para cada parámetro x se calcula recursivamente solo una vez. Después de que un valor de $solve(x)$ se haya almacenado en $value[x]$, se puede recuperar de manera eficiente cada vez que se vuelva a llamar a la función con el parámetro x.

3.1.3. Construyendo una solución

A veces se nos pide que hallemos el valor de una solución óptima y que demos un ejemplo de cómo se puede construir dicha solución. En el problema de la moneda, por ejemplo, podemos declarar otro arreglo que indique para cada suma de dinero la primera moneda en una solución óptima y de esta forma podemos construir una solución optima que se construye de N a 0

3.2. Cantidad de formas de devolver N

Vamos a considerar otra versión del problema de las monedas en donde la tarea es calcular el total de maneras de producir un total N utilizando las monedas. Para este ejemplo donde las denominaciones de las monedas son $1, 3, 4$ y $N = 5$ existen un total de 6 maneras:

1. 1 + 1 + 1 + 1 + 1 = 5
2. 3 + 1 + 1 = 5
3. 1 + 1 + 3 = 5
4. 1 + 3 + 1 = 5

5. 1 + 4 = 5

6. 1 + 4 = 5

Otra vez vamos a resolver el problema recursivamente. La función $solve(x)$ calcula la cantidad de formas que se obtiene una suma igual x. Por ejemplo, si la denominaciones de las monedas son $c = 1, 3, 4$, entonces $solve(5) = 6$ y su fómula recursiva es:

$$\begin{aligned} solve(x) = \quad & solve(x-1)+ \\ & solve(x-3)+ \\ & solve(x-4) \end{aligned}$$

Entonces la fórmula general recursiva es la siguiente:

$$solve(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in monedas} solve(x-c) + 1 & x > 0 \end{cases}$$

Si $x < 0$, el valor es 0, porque no hay soluciones. Si $x = 0$, el valor es 1, porque solo hay una forma de formar una suma vacía. De lo contrario, calculamos la suma de todos los valores de la forma $solve(x - c)$ donde c está en monedas.

A menudo, el número de soluciones es tan grande que no es necesario calcular el número exacto, pero es suficiente para dar la respuesta módulo m donde, por ejemplo, $m = 10^9 + 7$. Esto se puede hacer cambiando el código para que todos los cálculos se realicen módulo m.

Aún así, esta función no es eficiente, porque puede haber una exponencial número de maneras de devolver la suma. Sin embargo, aplicando el mismo proceder del caso anterior veremos cómo hacer el función eficiente con el uso de la memorización.

En este problema, usamos arreglos donde $ready[x]$ indica si el valor de $solve(x)$ ha sido calculado, y si es así, $counts[x]$ contiene este valor. La constante N ha sido elegida para que todos los valores requeridos quepan en los arreglos.

La función maneja los casos base $x < 0$ y $x = 0$ como antes. Luego, la función verifica desde $ready[x]$ es verdadero si lo es implica que $solve(x)$ ya se ha almacenado en el $count[x]$, y si es así, la función lo devuelve directamente. De lo contrario, la función calcula el valor de $solve(x)$ de forma recursiva y lo almacena en el $value[x]$.

Esta función funciona de manera eficiente, porque la respuesta para cada parámetro x se calcula recursivamente solo una vez. Después de que un valor de $solve(x)$ se haya almacenado en $count[x]$, se puede recuperar de manera eficiente cada vez que se vuelva a llamar a la función con el parámetro x.

Tenga en cuenta que también podemos construir iterativamente en ambos casos el valor del arreglo usando un ciclo que simplemente calcula todos los valores de solve para los parámetros

$0 \ldots n$. De hecho, la mayoría de los programadores competitivos prefieren esta idea porque es más corta y tiene factores constantes más bajos.

4. Implementación

4.1. C++

4.1.1. Mínima cantidad de billetes

```
int minimunCoins(vector<int> coins, int x) {
   vector<int> values(x+10,INT_MAX);
   values[0] = 0;
   for (int i = 1; i <= x; i++) {
      for (int c : coins) {
         if (i - c >= 0 && values[i - c] != INT_MAX) {
            values[i] = min(values[i], values[i - c] + 1);
         }
      }
   }
   return values[x] == INT_MAX ? -1 : values[x];
}
```

Notar que en esta implementación se retorna -1 si no se puede devolver la cantidad solicitada (x) con las denominaciones de las monedas disponibles ($coins$).

```
int minimunCoinsWithSolution(vector<int> coins,int x,vector<int> * solution){
   vector<int> values(x+10,INT_MAX);
   vector<int> s(x+10,0);
   values[0] = 0;
   solution->assign(x+10,0);
   for (int i = 1; i <= x; i++) {
      for (int c : coins) {
         if(i-c>=0 && values[i-c]!=INT_MAX && values[i]>values[i-c]+1){
            values[i] = values[i - c] + 1;
            s[i] = c;
         }
      }
   }
   solution->assign(s.begin(),s.end());
   return values[x] == INT_MAX ? -1 : values[x];
}

void printSolution(vector<int>solution, int x) {
   while (x > 0) {
      cout<<solution[x]<<endl;
      x -= solution[x];
   }
}
```

Esta implementación es muy similar a la anterior con la adecuación que se construye una posible solución para eso la función *minimunCoinsWithSolution* recibe un tercer parámetro *solution* vector donde se almacenará una de las posibles soluciones óptimas al problema. Una vez invocada la función *minimunCoinsWithSolution* y comprobar que existe al menos una solución solo bastará con invocar la funcionalidad *printSolution* pasando el arreglo $solution$ y x para imprimir una posible solución optima.

4.1.2. Cantidad de formas

```
ULL countWays(vector<int> coins, int x) {
   vector<ULL> counts(x + 10, 0);
   counts[0] = 1L;
   for (int i = 0; i <= x; i++) {
      if (counts[i] == 0L) continue;
      for (int c : coins) {
         if (i + c <= x) {
            counts[i + c] += counts[i];
            //counts[i + c]  %= MOD;
         }
      }
   }
   return counts[x];
}
```

Esta es una implementación que utiliza el enfoque de ***Bottom Up + Memorization***. Como se dijo anteriormente para este tipo de problema la solución tiende a crecer muy rapido por lo cual es muy normal que se pida una solución modulada para lo cual lo hemos puesto en el código de forma comentada para cuando la necesiten.

4.2. Java

4.2.1. Mínima cantidad de billetes

```
private int minimunCoins(int[] coins, int x) {
  int[] values = new int[x + 10];
  Arrays.fill(values, Integer.MAX_VALUE);
  values[0] = 0;
  for (int i = 1; i <= x; i++) {
     for (int c : coins) {
        if (i - c >= 0 && values[i - c]!=Integer.MAX_VALUE) {
           values[i] = Math.min(values[i], values[i - c] + 1);
        }
     }
  }
  return values[x]==Integer.MAX_VALUE ? -1 : values[x];
```

```
}
```

Notar que en esta implementación se retorna -1 si no se puede devolver la cantidad solicitada (x) con las denominaciones de las monedas disponibles ($coins$).

```
private int minimunCoinsWithSolution(int[] coins, int x, int[] solution) {
   int[] values = new int[x + 10];
   Arrays.fill(values, Integer.MAX_VALUE);
   values[0] = 0;
   for (int i = 1; i <= x; i++) {
      for (int c : coins) {
         if (i - c >= 0 && values[i - c] != Integer.MAX_VALUE && values[i] >
            values[i - c] + 1) {
            values[i] = values[i - c] + 1;
            solution[i] = c;
         }
      }
   }
   return values[x] == Integer.MAX_VALUE ? -1 : values[x];
}

private void printSolution(int[] solution, int x) {
   while (x > 0) {
      System.out.println(solution[x]);
      x -= solution[x];
   }
}
```

Esta implementación es muy similar a la anterior con la adecuación que se construye una posible solución para eso la función *minimunCoinsWithSolution* recibe un tercer parámetro *solution* arreglo donde se almacenará una de las posibles soluciones óptimas al problema. Este parámetro debe estar inicializado previamente con una capacidad igual a x. Una vez invocada la función *minimunCoinsWithSolution* y comprobar que existe al menos una solución solo bastará con invocar la funcionalidad *printSolution* pasando el arreglo $solution$ y x para imprimir una posible solución optima.

4.2.2. Cantidad de formas

```
private long countWays(int[] coins, int x) {
   long[] counts = new long[x + 10];
   Arrays.fill(counts, 0L);
   counts[0] = 1L;
   for (int i = 0; i <= x; i++) {
      if(counts[i]==0L)continue;
      for (int c : coins) {
         if (i + c <= x) {
            counts[i+c] += counts[i];
```

```
                //counts[i+c]  %=MOD;
            }
        }
    }
    return counts[x];
}
```

Esta es una implementación que utiliza el enfoque de ***Bottom Up + Memorization***. Como se dijo anteriormente para este tipo de problema la solución tiende a crecer muy rapido por lo cual es muy normal que se pida una solución modulada para lo cual lo hemos puesto en el código de forma comentada para cuando la necesiten.

5. Aplicaciones

Este es de los problemas de la programación dinámica, uno de los denomindados como clásicos. Para la solución de las dos variantes se llega a un enfoque ***Bottom Up + Memorization*** (implementaciones) aunque en el proceso de análisis se trabajo con el enfoque ***Top Down + (Memorization)***.

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.

6. Complejidad

La complejidad de ambos algoritmos es de O(N*K) donde N es el valor máximo y K el número de denominaciones diferentes de billetes disponibles tener en cuenta que solo puede aplicar cuando no nos ponen restricciones en cuanto la cantidad de billetes por denominaciones además, el costo de memoria puede ser peligroso mientras N sea mas grande.

7. Ejercicios

A continuación una lista de ejercicios que se pueden resolver aplicando los algoritmos abordados en esta guía:

- CSES - Dice Combinations
- UVA - 00674 - Coin Change
- CSES - Minimizing Coins

- CSES - Coin Combinations I