

Asesoría 4: Programación Dinámica 1

November 12, 2020

1 Programación Dinámica

1.1 Change Making Problem

Consider a money system consisting of n coins. Each coin has a positive integer value. Your task is to produce a sum of money x using the available coins in such a way that the number of coins is minimal.

For example, if the coins are $\{1, 5, 7\}$ and the desired sum is 11, an optimal solution is $5 + 5 + 1$ which requires 3 coins.

Input

The first input line has two integers n and x : the number of coins and the desired sum of money.

The second line has n distinct integers c_1, c_2, \dots, c_n : the value of each coin.

Output

Print one integer: the minimum number of coins. If it is not possible to produce the desired sum, print -1 .

$$OPT(x) = \begin{cases} x + 1 & \text{si } x < 0 \\ 0 & \text{si } x = 0 \\ \min_{c \in \text{coins}} \{ OPT(x - c) + 1 \} & \text{si } x > 0 \end{cases}$$

1.2 Explicación

El problema consta en que dada una suma de dinero x y un conjunto *coins* de monedas, cada una con un valor numérico, queremos hallar el mínimo número de monedas que tenemos que usar para poder llegar a esa suma x de dinero.

Definimos un $OPT(x)$ que devuelve el mínimo número de monedas necesarias para llegar a la suma x dado un conjunto $coins$ de monedas. Si esa suma es imposible con el conjunto de monedas que tenemos, $OPT(x)$ retornará $x + 1$.

Los casos base son cuando $x - c < 0$, en ese caso, hacemos que la solución óptima devuelva $x + 1$, lo cual interpretaremos como un flag que indica que llegar a esa suma es imposible con las monedas que tenemos. El otro caso base, es cuando la suma $x = 0$, en ese caso, el número de monedas para llegar a esa suma es 0, ya que ya hemos llegado a la suma que queremos. Finalmente, el caso recursivo es que para cada moneda c que pertenece al conjunto de monedas que tenemos, hacemos una llamada recursiva pasándole la suma original x menos el valor de la moneda actual c y le sumamos uno, ya que estamos usando la moneda c . Queremos el mínimo valor de todas estas llamadas recursivas. El número de llamadas recursivas será igual a $|coins|$.

1.3 Problema Resta de Dígitos

Ejercicio 6 (3 ptos). Describa un algoritmo $O(n \lg n)$ para el siguiente problema. Usted recibe un número n , en cada paso, usted puede restar de n cualquier cifra que aparece en n : ¿cual es el menor número de pasos para convertir n a 0? Por ejemplo, si usted recibe el número 27, necesita solo 5 pasos: $27 \rightarrow 20 \rightarrow 18 \rightarrow 10 \rightarrow 9 \rightarrow 0$.

$$OPT(num) = \begin{cases} 0 & \text{si } num = 0 \\ 1 & \text{si } 9 \geq num \geq 1 \\ \min_{digito \in num} \{ OPT(num - digito) + 1 \} & \text{si } num \geq 10 \end{cases}$$


1.4 Explicación

Definimos un $OPT(num)$ que devuelve el mínimo número de pasos (restas) que tenemos que hacer para convertir num a 0 si es que solo tenemos permitido restarle a num cualquiera de sus dígitos.

Son dos los casos base de este problema. El primero, es cuando el número ya es 0, entonces el número de pasos para convertir 0 a 0 es 0. El segundo, es que el número está en el intervalo desde 1 hasta 9. En ese caso, el mínimo número de pasos siempre será 1, ya que basta con restarle al número su único dígito. Finalmente, para el caso recursivo, si el número tiene dos o más dígitos, intentamos restarle al número cada dígito que tiene más uno (ya que el restarle uno de sus dígitos cuenta como un paso más) y elegimos el mínimo de esas llamadas recursivas.

1.5 Problema Max Subarray Sum en $\mathcal{O}(n)$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7



maximum subarray

$$OPT(A, i) = \begin{cases} A[i] & \text{si } i = 1 \\ \max_{i=2}^n \{ A[i], OPT(A, i-1) + A[i] \} & \text{caso contrario} \end{cases}$$

Definimos un $OPT(A, i)$ que devuelve el max subarray sum del arreglo A hasta el índice i . Acuérdense que dp se basa en subproblemas. Sirve bastante definir OPT 's que van hasta el índice i .

En este caso, tenemos solo un caso base, cuando estamos en el primero elemento del arreglo. Ahí, el max subarray sum es igual al valor de ese primer elemento. Caso contrario, hacemos una pregunta: ¿empezamos un nuevo max subarray sum? o ¿extendemos el max subarray sum que ya tenemos? Elegimos el máximo de ambas respuestas.

1.6 Edit Distance

The *edit distance* between two strings is the minimum number of operations required to transform one string into the other.

The allowed operations are:

- Add one character to the string.
- Remove one character from the string.
- Replace one character in the string.

For example, the edit distance between LOVE and MOVIE is 2, because you can first replace L with M, and then add I.

Your task is to calculate the edit distance between two strings.

Input

The first input line has a string that contains n characters between A–Z.

The second input line has a string that contains m characters between A–Z.

Output

Print one integer: the edit distance between the strings.

$$OPT(n, m) = \begin{cases} 0 & \text{si } n = 0, m = 0 \text{ hago un } \textit{replace} \text{ trivial} \\ n.length & \text{si } n \geq 1, m = 0 \text{ hago } n \text{ } \textit{delete}'\text{s} \\ m.length & \text{si } n = 0, m \geq 0 \text{ hago } m \text{ } \textit{add}'\text{s} \\ \min\{ \\ \quad 1 + OPT(n, m - 1), \\ \quad 1 + OPT(n - 1, m), \\ \quad cost + OPT(n - 1, m - 1) \} & \text{si } n.length > 1 \text{ y } m.length > 1 \end{cases}$$

Sea $s1$ y $s2$ dos strings con tamaño n y m , respectivamente. Definimos un $OPT(n, m)$ que devuelve el mínimo número de ediciones, ya sea *add*, *delete* o *replace*, que tenemos que hacerle a $s1$ para poder convertirlo a $s2$.

Defino un $OPT(n, m)$ que retorna el mínimo número de ediciones que le tengo que hacer al string $s1$ de tamaño n para convertirlo al string $s2$ de tamaño m .

Este problema tiene tres casos bases. El primero, cuando ambos strings tienen tamaño 0, esto

quiere decir que su *edit distance* es 0, ya que no tenemos que hacer nada para transformar s_1 y s_2 . Para mantenernos el término de ediciones, diremos que se hace un *replace* trivial que cuesta 0. El segundo, es cuando s_2 tiene tamaño 0 y s_1 tiene un tamaño mayor o igual a 1. En este caso, tenemos que borrar todos los caracteres de s_1 para poder convertir s_1 a s_2 , es decir, tenemos que hacer n *delete*'s. El tercero, es cuando s_1 tiene tamaño 0 y s_2 tiene un tamaño mayor o igual a 1. En este caso, tenemos que añadirle a s_1 todos los caracteres de s_2 , es decir, hacer m *add*'s. Finalmente, para el caso recursivo, hacemos tres llamadas recursivas, una para cada operación y elegimos el que tenga *edit distance* mínimo: una para *add*, otra para *delete* y otra para *replace*. En otras palabras, probamos tres posibilidades: añadimos un carácter al final de s_1 , eliminamos un carácter al final de s_1 o hacemos *match* ($cost = 0$) o reemplazamos ($cost = 1$) el último carácter de s_1 con el último de s_2 . El *match* ocurre cuando ambos caracteres finales de s_1 y s_2 son iguales, en dicho caso no tenemos que editar.

2 Material Adicional

- **Link al Jamboard:** <https://jamboard.google.com/d/117K3i7-ul6VtgC5q0gYv9hULHuWt4uKlwe4ZnuHqGz8/edit?usp=sharing>
- **Playlist:** <https://www.youtube.com/watch?v=Zq4upTEaQyM&list=PLiQ766zSC5jM20KvR8sooOuGgZkvnOCTI>
- **Playlist:** https://www.youtube.com/watch?v=8LusJS5-AGo&list=PLrmLmBdmIlpsHaNTPP_jHHDx_os9ItYXr
- **Tardos:** Sección 6.1, 6.2 y 6.4.
- **Cormen:** Capítulo 15.