Problema de las monedas usando programación dinámica.

Ahora abordaremos una ejemplo de programación dinámica para que quede más claro.

El problema es, dado un conjunto de monedas $C = \{c_0, c_1, \ldots, c_{n-1}\}$ y queremos entregar una cantidad de dinero S con la **mínima** cantidad de monedas posibles. Por ejemplo con $C = \{5, 10, 20, 50\}$ queremos entregar $S = 180$ nuestro sentido común seria tomar 3 de 50 y uno de 20 y uno de 10, y si esta es la cantidad mínima.

¿ Cómo llegamos a esa solución? en este caso una solución greedy(siempre tomar la mejor solución en este caso siempre tomar la mayor), en este caso siempre tomar la moneda más grande funciona. ¿Pero esto siempre funciona? la respuesta es no y aquí un ejemplo.

$C = \{1, 3, 3, 4\}$ y $S = 6$ si siempre tomaramos la moneda más grande llegariamos a la solución $Sol = \{4, 1, 1\}$ ¿es esta la solución optima? no, hay una solución con menor cantidad de monedas. $Sol = \{3, 3\}$ la cual solo contiene dos monedas y por lo tanto es mejor. Por lo tanto siempre tomar la moneda mayor no conviene. ¿Cómo lo resolvemos?

Podemos hacer una fuerza bruta inteligente es decir Programación Dinámica. Primero veamos si pasa el filtro, es decir, si tiene subproblemas con traslapes.

Sea $Cantidad(x) = y$ donde y es la mínimida cantidad de monedas en C para entregar el cambio x. Dada $c \in C$ si usamos la moneda c nos resta por entregar $x - c$ de cambio. Entonces si hay subproblemas con traslapes.

¿Cómo expresamos la recurrencia? Podemos hacerlo de esta forma.

Opción 1.

$$cantidad(x) = \begin{cases} \min_{c \in C} cantidad(x - c) + 1 & si \quad x \geq 0 \\ 0 & si \quad x = 0 \\ \infty & si \quad x < 0 \end{cases}$$

En esta variante probamos cada moneda y tomamos la mínima.

Opción 2.

Con una pequeña modificación a la función $Cantidad(x, k) = y$ donde k es la moneda actual con la que se entrega.

Esta versión ocupa más memoria que la anterior

$$cantidad(x,k) = \begin{cases} \min(cantidad(x-c,k)+1, cantidad(x,k+1)) & si \quad x \geq 0 \\ 0 & si \quad x = 0 \\ \infty & si \quad x < 0 \\ \infty & si \quad k > |C| \end{cases}$$

Esta variante tiene una característica muy especial que si desearamos únicamente usar una monedaen esta forma es más fácil.

Veamos los 5 puntos que vimos en el primer documento de programación Dinámica.

1. Identificamos que distigue al estado (x o x y k)
2. Dividimos el problema de en subproblemas(función pasada.)
3. Identificamos casos base (los de la función)
4. resolvemos por separados los casos.
5. Memorizamos subproblemas.

Implementación de cada una das funciones (Top down).

Primera forma.

```
int cantidad(int x){
        if( x < 0) return INF;
        if( x == 0) return 0;
        if( memo[x] != -1) return memo[x];
        int minimo = INF;
        for(int i=0; i<n; i++){// n es la longitud de coins
                minimo = min(cantidad(x-coins[i]) + 1,minimo);
        }
        return memo[x] = minimo;
}
```

Segunda forma

```
int cantidad2(int x, int k){
        if( x < 0) return INF;
        if( x == 0) return 0;
        if( k >= n) return INF;
        if( memo2[x][k] != -1) return memo2[x][k];
        int minimo = min(cantidad2(x - coins[k], k) + 1, cantidad2(x,k+1));
        return memo2[x][k] = minimo;
}
```

Definiciones generales para cada uno:

```
const int INF = 1e8;
vector<int> memo(100000, -1);
int memo2[100000][2000];
vector<int> coins;
int n, x;
```

implementación Bottom-Up primera forma

Acabamos ver la implementació Top-Down ahora veramos la implementación Bottom-Up es decir desde abajo constriiremos la solución.

```
dp_coins[0] = 0; //entregar cero
for(int i=1 ; i <= x ; i++){
    dp_coins[i] = INF;
    for(auto c: coins){//alternativo (int j=0; j<n; j++) donde n es la
cantidad de monedas.
        if(i - c  >= 0){
            dp_coins[i] = min(dp_coins[i], dp_coins[i - c] + 1);
        }
    }
}
```

Implementación Bottom-Up segunda forma

En este caso `dp_2[i][j]` guarda la minima cantidad de de entregar `i` uidades empezando desde la moneda `j`, por lo que la respuesta final se encuentra `dp_2[x][0]`

```
for(int i=0 ; i<=n; i++) dp_2[0][i] = 0;
for(int i=0; i<=x; i++) dp_2[i][n] = INF;

for(int i=1; i<= x ; i++){
    for(int j = n - 1; j >=0; j--){
        int x = INF;
        if( i - coins[j] >= 0){
            x = dp_2[i - coins[j]][j];
        }
        int y = INF;
        if( j + 1 < n){
            y = dp_2[i][j + 1];
        }
        dp_2[i][j] = min(x + 1,y);
    }
 }
```

Complejidad: $\mathbb{O}(x|C|)$

Contando las maneras de entregar el dinero.

Ahora abordaremos un problema con una descripción similar pero un problema escencialmente distinto en el cual ahora resolveremos un problema de contar las maneras de entregar el cmbio, más formalmente podemos decir:

Dado un conjunto $C = \{c_0, c_1, \ldots, c_{n-1}\}$ y una cantidad S.

¿De cuántas maneras podemos entregar el cambio?.

Ejemplo:

Sea $C = \{1,3,5\}, S = 6$ tenemos las siguientes formas de entregar el dinero: $\{1,1,1,1,1,1\}, \{1,1,1,3\}, \{3,3\}, \{1,5\}$ si no se me paso alguna la respuesta es 4 **si no permitimos contar en diferentes ordenes**, claramente con esa observación de este problema se desglosan dos problemas uno que permite contar formas repetidas y uno que no. El problema se presenta cuando no se permiten contar formas repetidas pues se puede llegar a sobre contar las formas y se requiere un análisis distinto.

Aceptando Formas distintas.

Este problema es simple

Usando una recurrencia muy parecida a la anterior de la siguiente forma: donde $formas(x)$ es la cantidad de maneras de representar a x usando C

$$formas(x) = \begin{cases} \sum_{c \in C} formas(x-c) & si \quad x > 0 \\ 1 & si \quad x = 0 \\ 0 & si \quad x < 0 \end{cases}$$

Se puede verificar que se cumplen los 5 puntos que hemos mencionado. Implementación.

```
// TOP DOWN
int formas(int x){
        if( x == 0) return 1;
        if( x < 0) return 0;
        if(memo[x] != -1) return memo[x];
        int cuantos = 0;
        for(auto k : coins){
                cuantos += formas(x - k);
        }
        return memo[x] = cuantos;
}
```

Ahora en la versión Bottom-Up tendremos un arreglo que se llame `dp_formas[x]` con la misma finalidad quela función y las mismas rtansiciones

```
//BOTTOM-UP
dp_formas[0] = 1;
for(int i=1 ; i<= q; i++){
    for(auto c: coins){
        if( i -c >= 0){
            dp_formas[i] += dp_formas[i - c];
        }
    }
}
```

Este problema no tuvo mayor dificultad pues el problema de sobrecontar las formas no esta presente, pero ahora analicemos el caso cuando sobre contar las formas si importa.

complejidad $\mathbb{O}(x|C|)$

Sólo permitir formas únicas

Ahora ejemplifiquemos mejor a que nos referimos con formas únicas pensemos tenemos $C = \{1,2,3,4,5\}$ y $S = 5$ una forma de reperesentarlo es $\{1,1,2,1\}$ y $\{2,1,1,1\}$ pero estas son escencialemnte las únicas si contamos el orden lexicográfico solo sería una y sería $\{1,1,1,2\}$ con lo cual solo se contaría una única vez, nuestra meta es contar cuantas representaciones únicas existes para nuestro S.

Primero analicemos, ¿Es suficiente definir un estado únicamente con el valor x? es decir para obtener los subproblemas, ¿es suficiente esa información?.

Por un momento pensemos que si es suficiente (**spoiler**, *no lo es*), por ejemplo del estado $x = 5$ podemos ir a un estado $x = 3$(tomando la moneda de 2) y de ese estado toammos tres monedas de a peso y queda la respuesta $\{2,1,1,1\}$ pero del estado $x = 5$ se deriva otro estado el que es tomar una moneda de a peso y movernos a $x = 4$ y de ahi tomamos dos de de a 1 y uno de dos teniendo una solución $\{1,1,1,2\}$ la cual es escencialemte la misma de la pasada por lo cual estamos sobrecontando por lo que necesitamos más información por cada estado.

Podemos representar el estado ahora de la forma $formas(x,k)$ donde x es el monto a entregar y k la moneda que actualmente estamos usando.

Ahora veamos como se podrían hacer las transiciones. Vamos a pensar las transiciones de la forma **tomar o no tomar**

$$formas(x,k) = \begin{cases} formas(x-c_k,k) + formas(x,k+1) & si \quad x > 0 \\ 1 & si \quad x = 0 \\ 0 & si \quad x < 0 \\ 0 & si \quad k >= n \end{cases}$$

donde $n = |C|$ y c_k la k-ésima moneda.

En el primer caso representa las formas de entregar el cambio tomando la moneda o no tomándola.

¿Por qué esto evade sobre contar? pues desde que decidimos evadir una moneda ya no la volvemos a considerar paorque ya no checamos monedas anteriores.

De esta forma podemos ya programar nuestra recurrencia con la técnicacaracterística de la DP que es la memorización.

En top down queda de la siguiente forma

```
long long formasuniq(int x,int k){
        if( x == 0) return 1;
        if( x < 0) return 0;
        if( k >= n) return 0;
        if(memouniq[x][k]!=-1) return memouniq[x][k];
        long long cuantas = formasuniq(x - coins[k], k) + formasuniq(x,
k+1);
        return memouniq[x][k] = cuantas;
}
```

En bottom up queda de la siguiente forma.

```
    for(int i=0 ; i<= n ; i++) dp[0][i] = 1;
    for(int i=1 ; i <= q; i++){
        for(int j = n - 1; j >=0 ; j--){
            if(i - coins[j] >=0) dp[i][j] += dp[ i - coins[j] ][j];
                dp[i][j] += dp[i][j+1];

        }
    }
```

Ambas de estas tiene complejidad $\mathbb{O}(x|C|)$