

Programación Dinámica

Wilmer Emiro Castrillón Calderón

12 de marzo de 2022

1. Introducción a la programación dinámica

La programación dinámica es una metodología utilizada para reducir la complejidad computacional a un algoritmo, es usada principalmente para resolver problemas de optimización, se basa en la estrategia *divide y vencerás*, consiste en tomar un problema complejo y dividirlo sucesivamente en sub-problemas mas pequeños hasta llegar a un caso base, y partir de ahí empezar a construir la solución de cada sub-problema, hasta llegar a una solución global. Durante la búsqueda de soluciones se utiliza tablas de memorización, en la cuales se irá guardando la solución óptima de cada sub-problema. Como abreviatura a programación dinámica vamos a usar dp, pues son sus siglas en ingles.

Los problemas que pueden ser resueltos utilizando programación dinámica presentan tres condiciones básicas:

1. El problema se puede dividir en sub-problemas, y estos a su vez en más sub-problemas, y así hasta llegar a uno o múltiples casos base.
2. La solución óptima de cada sub-problema depende de la solución óptima de cada uno de sus propios sub-problemas, entonces cumple con el principio de optimalidad de Bellman.
3. Se presenta superposición de problemas, es decir, existen sub-problemas que aparecen múltiples veces a lo largo de la búsqueda de la solución general. Aunque como tal no es obligatorio, es la razón principal que le permite tener menor complejidad computacional.

Ejemplo inicial.

El ejemplo básico más usado es con la sucesión de Fibonacci, esta comienza con los números 0 y 1, y a partir de estos dos números iniciales los siguientes son la suma de los dos anteriores, entonces los primeros números de Fibonacci son:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,

Ahora se buscará la manera de calcular y guardar en un vector los primeros n números de la sucesión eficientemente, de tal forma que la posición i -ésima del vector corresponda al número i -ésimo de la sucesión. Para solucionar el problema con dp primero debemos identificar los sub-problemas, es decir dividir el problema en partes mas pequeñas, en este caso obtenemos la siguiente formula recursiva:

$$fibo(n) = \begin{cases} n & \text{if } n \leq 1 \\ fibo(n-1) + fibo(n-2) & \text{if } n \geq 2 \end{cases}$$

Para implementar dp surgen dos enfoques: Top-Down y Buttom-Up, vamos a resolver este ejemplo usando los dos.

Buttom-Up

En este enfoque vamos a ir recorriendo la tabla de memorización mientras la llenamos, comenzando desde los casos base de la formula recursiva y a partir de ahí ir calculando y guardando los demás resultados en la tabla. En este enfoque primero se calcula el resultado de todos los subproblemas antes de dar alguna solución global. Para este ejercicio comenzamos llenando en un vector v los casos base: $v[0] = 0$; $v[1] = 1$; y luego llenamos el resto del vector según nos indica la formula recursiva: $v[i] = v[i-1] + v[i-2]$, de tal forma que $v[i-1]$ y $v[i-2]$ son valores que se han calculado antes y ademas $v[x] = \text{fibo}(x)$. A continuación se muestra el ejemplo en C++ calculando los primero 45 números de la sucesión:

```
1 void dp(){
2     v[0] = 0;
3     v[1] = 1;
4     for(int i = 2; i < 45; i++)
5         v[i] = v[i-1] + v[i-2];
6 }
```

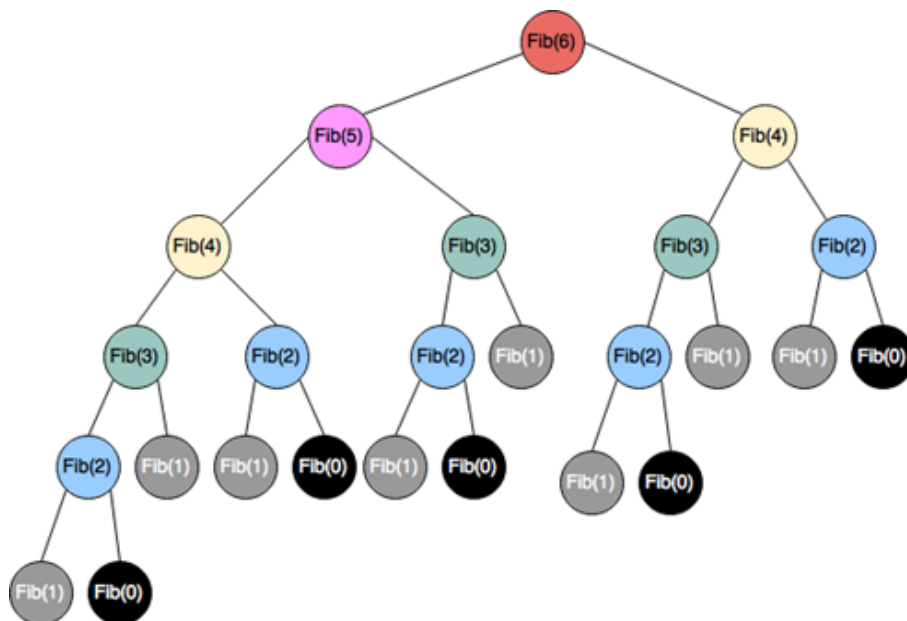
Aunque en este enfoque no utilizamos llamados recursivos, puede resultar mas difícil de implementar, pues por lo general es menos intuitivo.

Top-Down

A diferencia del anterior enfoque en este vamos a utilizar llamados recursivos, en los cuales se irá guardando los resultados en la tabla de memorización mientras se calculan, y así evitar que se realice dos veces el mismo llamado recursivo. Es decir se calcula el resultado solo de los subproblemas que sean necesarios para dar alguna solución global, a diferencia del Buttom-Up donde se calculaban primero todos los subproblemas antes de poder dar alguna solución. En este enfoque se hace mas notorio la importancia de la tabla de memorización, veamos la eficiencia del algoritmo si dejamos únicamente la formula recursiva, el código seria el siguiente:

```
1 int fibo(int x){
2     if(x <= 1)
3         return x;
4     else
5         return fibo(x-1) + fibo(x-2);
6 }
```

Si ejecutamos este código para $\text{fibo}(6)$ obtenemos el siguiente arbol de recursión:



Se puede observar que se realizan múltiples llamados repetidos, *fibo(4)* se repite 2 veces, *fibo(3)* 3 veces, *fibo(2)* 5 veces, *fibo(1)* 8 veces y *fibo(0)* 5 veces, resultan bastantes llamados recursivos solamente para encontrar *fibo(6)*, esta solución tiene una complejidad exponencial!, en números pequeños no es muy notorio pero cuando intentamos encontrar *fibo(42)* o *fibo(45)* el tiempo de ejecución se hace muy alto, entonces para mejorar el tiempo se debe implementar una tabla de memorización, en la cual se guardara el resultado de cada llamado recursivo, ademas se debe agregar un condicional al inicio del método preguntando si la solución de ese sub-problema ya fue encontrada, en tal caso se devuelve el valor guardado, sino se calcula y se guarda, en este ejemplo la tabla comienza llena con -1, un valor que nunca es usado en la solución, de tal forma que si una posición es igual a -1 entonces se debe calcular.

```

1  int fibo(int x){
2      if(v[x] != -1) return v[x];
3
4      if(x <= 1)
5          return v[x] = x;
6      else
7          return v[x] = fibo(x-1) + fibo(x-2);
8  }

```

Este ejercicio resulta muy trivial, se puede resolver facilmente sin pensar en dp, ahora se mostraran problemas en los cuales se hace necesario utilizar dp para dar una solución eficiente.

Recorrido óptimo en una matriz.

Teniendo una matriz de $n \times m$ con números enteros, se quiere conocer la suma maxima que se puede obtener recorriendo la matriz, comenzando desde la esquina superior-izquierda y terminando en la esquina inferior-derecha, realizando únicamente pasos hacia la derecha y abajo, por ejemplo:

| | | |
|---|----|----|
| 5 | 6 | 4 |
| 3 | 8 | 5 |
| 2 | 11 | 15 |
| 5 | 2 | 17 |

El camino óptimo es 5-6-8-11-15-17 y la suma es 62.

Este es un problema de optimización, a simple vista la solución consistiría de siempre ir a la casilla adyacente con mayor valor, pero no siempre es el mejor camino, por ejemplo con la siguiente matriz:

| | | | |
|----|----|----|----|
| 1 | 12 | 4 | 9 |
| 6 | 5 | 21 | 15 |
| 35 | 18 | 8 | 10 |
| 12 | 2 | 4 | 15 |

No es una solución óptima ir a la siguiente casilla con mayor valor, pues haciendo esto se tendría el siguiente camino 1-12-5-21-15-10-15 con un acumulado de 79, en cambio el camino 1-6-35-18-8-10-15 tiene un acumulado de 93, entonces con la anterior estrategia no siempre se obtiene el camino óptimo. En este problema es necesario realizar una toma de decisiones, pues existen múltiples opciones de caminos, y se necesita encontrar el conjunto de decisiones que permita llegar a un resultado óptimo, una solución inicial puede ser con BackTraking pero esta tendría complejidad exponencial $O(2^n)$, por lo tanto se necesita utilizar otro enfoque.

Este problema se puede resolver con programación dinámica, pues se puede encontrar la solución general a partir de la solución de sub-problemas más pequeños, supongamos que tenemos una matriz T de tamaño $n \times m$ a la cual se va a calcular la suma de su recorrido óptimo, para facilidad se indexara desde 1 tomando la esquina superior izquierda como $T[1][1]$, para encontrar el camino óptimo hasta $T[n][m]$ (ultima casilla) es necesario primero encontrar el óptimo de $T[n-1][m]$ y $T[n][m-1]$, pues para llegar a $T[n][m]$ hay dos opciones, llegar por arriba (equivalente a tomar la decisión ir-abajo desde $T[n-1][m]$) o llegar por la izquierda (equivalente a tomar la decisión ir-derecha desde $T[n][m-1]$), entonces la solución general sera $T[n][m] + \text{máximo entre}(\text{camino óptimo hasta } T[n-1][m], \text{camino óptimo hasta } T[n][m-1])$, y a su vez el óptimo para ellos se calcula de manera similar, el óptimo para llegar hasta $T[n-1][m]$ es $T[n-1][m] + \text{máximo entre}(\text{camino óptimo hasta } T[n-2][m], \text{camino óptimo hasta } T[n-1][m-1])$ y de manera similar para $T[n][m-1]$, de esta forma se puede dividir el problema en sub-problemas.

Ahora ya se puede empezar a construir una formula recursiva, llamaremos i a la posición en fila y j a la posición en columna, entonces $f(i, j)$ devolverá el valor del recorrido óptimo desde $T[1][1]$ hasta $T[i][j]$, ahora se deben establecer los casos base, entonces si i y j son iguales a 1 entonces el resultado es $T[1][1]$ (se encuentra en el punto inicial), si solo i es 1 entonces el resultado es $T[1][j] + f(1, j-1)$, solo se puede llegar a $T[1][j]$ desde su izquierda, pues desde arriba estaría afuera de la matriz, si solo j es 1 entonces de manera similar se tiene como resultado $T[i][1] + f(i-1, 1)$, ahora ya se tienen los casos base y se puede construir la formula recursiva:

$$f(i, j) = \begin{cases} T[1][1] & \text{if } i, j = 1 \\ T[1][j] + f(1, j-1) & \text{if } i = 1 \\ T[i][1] + f(i-1, 1) & \text{if } j = 1 \\ T[i][j] + \text{mayor}(f(i-1, j), f(i, j-1)) & \text{if } i, j \neq 1 \end{cases}$$

Realzando manualmente la función recursiva se puede tener mas claridad. El caso $f(1, 1)$ indica que la solución es el elemento en la primera casilla, la solución para la primera fila sera la sumatoria de los elementos desde la primera casilla hasta la columna correspondiente, ese es el único camino posible, de similar manera ocurre con la primera columna (figura 1), para los demás casos se abren dos opciones, llegar desde arriba o llegar desde la izquierda (figura 2), la formula indica tomar el de mayor valor, de esta manera mientras se avanza se ira seleccionando

el mejor camino hasta llegar a la ultima casilla.

| | | | |
|----|----|----|----|
| 1 | 13 | 17 | 26 |
| 7 | | | |
| 42 | | | |
| 54 | | | |

Figura 1:

| | | | |
|----|----|----|----|
| 1 | 13 | 17 | 26 |
| 7 | ? | | |
| 42 | | | |
| 54 | | | |

Figura 2:

| | | | |
|----|----|----|----|
| 1 | 13 | 17 | 26 |
| 7 | 18 | 39 | 54 |
| 42 | 60 | 68 | 78 |
| 54 | 62 | 72 | 93 |

Figura 3:

Para terminar el dp es necesario agregar la tabla de memorización, pues aplicar la formula directamente resultaría en un algoritmo de complejidad exponencial, como tabla de memorización se usara otra matriz de igual tamaño, esta se llamara memo y guardara las soluciones, es decir, que en $memo[i][j]$ se guardara el acumulado óptimo para ir desde $T[1][1]$ hasta $T[i][j]$, en otras palabras $memo[i][j] = f(i, j)$, ahora se puede solucionar el problema usando cualquiera de los dos enfoques de dp, para usar Top-Down solamente se agrega la tabla de memorización a la formula recursiva, resultando en la siguiente función (recordar que en C++ se indexa desde 0):

```

1 int f(int i, int j){
2     if(i==0 && j==0) return T[0][0];
3     if(memo[i][j] != -1) return memo[i][j];
4
5     if(i == 0) return memo[0][j] = T[0][j] + f(0, j-1);
6     if(j == 0) return memo[i][0] = T[i][0] + f(i-1, 0);
7
8     return memo[i][j] = T[i][j] + max(f(i-1, j), f(i, j-1));
9 }
```

para conocer el resultado solo se debe imprimir $f(n-1, m-1)$. para la solución con Bottom-Up se debe recorrer la matriz y llenarla según la formula recursiva, se debe comenzar con los casos base antes de calcular lo demás, esto resultaría en la siguiente función:

```

1 void dp(){
2     memo[0][0] = T[0][0]; //caso base
3
4     for(int i = 1; i < n; i++) //llenar filas
5         memo[i][0] += memo[i-1][0] + T[i][0];
6
7     for(int i = 1; i < m; i++) //llenar columnas
8         memo[0][i] += memo[0][i-1] + T[0][i];
9
10    for(int i = 1; i < n; i++)
11        for(int j = 1; j < m; j++)
12            memo[i][j] += max(memo[i-1][j], memo[i][j-1]) + T[i][j];
13 }
```

De esta forma para conocer la solución se debe imprimir $memo[n-1][m-1]$. La complejidad algorítmica resultante es $O(n*m)$, resultando mejor que un algoritmo de complejidad exponencial utilizando fuerza bruta.

2. Sub-SetSum

También conocido como suma de subconjuntos, el problema consiste en que a partir de un conjunto de números enteros V , encontrar si es posible que otro número n sea el resultado de la suma de los elementos de algún subconjunto de V , por ejemplo con el conjunto $V = \{1, 2, 5\}$ es posible sumar 7 usando el subconjunto $\{2, 5\}$, con este conjunto V sería posible sumar $\{1, 2, 3, 5, 6, 7, 8\}$.

Utilizando un algoritmo de BackTraking para probar todos los posibles subconjuntos se obtendría una complejidad $O(2^n)$, siendo bastante alta aun para rango pequeños. Este problema puede ser resuelto usando dp, se puede reescribir el numero n como: $n = V[k_1] + x_1$, con $V[k_1]$ como algún numero en V , de manera similar se puede reescribir x_1 como $x_1 = v[k_2] + x_2$ y así sucesivamente hasta encontrar $x_n = v[k_n] + 0$, de esta manera si se toma cada elemento de V que fue utilizado, se podría encontrar el subconjunto: $v[k_1], v[k_2], \dots, v[k_n]$ el cual sumando todos sus elementos obtenemos n , en general se deberá probar para cada elemento en V si puede pertenecer a algún subconjunto que sume n .

En este problema el resultado deberá ser 'verdadero' o 'falso', mostrando si es posible sumar n con algún subconjunto, con los sub-problemas establecidos se puede ahora definir los casos base, si en algún momento se llega a $x_n = 0$ entonces la solución global sera 'verdadero' (se encontró una solución, entonces es posible sumar n), pero si en algún momento usamos todos los elementos en V y aun $x_n > 0$, o en algún momento se llega a $x_n < 0$ entonces la solución en ese sub-problema sera 'falso', entonces se tiene la siguiente función recursiva (siendo 1 verdadero, y 0 falso):

$$f(n, k) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n < 0, k > \text{longitud}(V) \\ 1 & \text{if } f(n - V[k], k + 1) = 1 \\ 0 & \text{if } \text{otro} \end{cases}$$

Con esta formula la solución sera igual a $f(n, 0)$, para implementarla se debe hacer una tabla de memorización, la cantidad de columnas de la tabla debe ser por lo menos igual a la longitud del vector, y la cantidad de filas por lo menos igual al resultado más grande que se puede obtener. Igualmente se llenara inicialmente con -1.

```
1  int memo[10][5];
2  int dp(int n, int k, vector<int> &V){
3      if(memo[n][k] != -1) return memo[n][k];
4      if(n == 0) return 1;
5      if(n < 0 || k >= V.size()) return 0;
6
7      if(dp(n - V[k], k + 1, V) == 1 || dp(n, k + 1, V) == 1)
8          return memo[n][k] = 1;
9      else return memo[n][k] = 0;
10 }
```

Este problema también puede ser resuelto usando el enfoque Bottom-Up, donde se buscaría todas las posibles sumas que se pueden formar con los números en V , se tomara m como el tamaño de V y por facilidad se usara 1 como índice inicial. Los subproblemas se podrían ver de la siguiente manera: la solución general incluye sumar el elemento $V[m]$ a todas las soluciones encontradas hasta $V[m - 1]$, y este a su vez depende de encontrar $V[m - 1]$ y así sucesivamente hasta llegar a $V[1]$ que sera nuestro caso base, mas detalladamente la solución hasta $V[1]$ es

solamente él mismo, las soluciones hasta $V[2]$ es él mismo, $V[1]$ y la suma $V[1] + V[2]$, el conjunto de soluciones hasta ahora es $\{V[1], V[2], V[1] + V[2]\}$, ahora la solución hasta $V[3]$ sera $\{V[1], V[2], V[3], V[1] + V[2], V[1] + V[3], V[2] + V[3], V[1] + V[2] + V[3]\}$, y así sucesivamente hasta $V[m]$, descartando números repetidos.

Por ejemplo sea $V = \{3, 5 \text{ y } 8\}$, la solución hasta 3 es él mismo, el conjunto de soluciones hasta ahora es $\{3\}$, la solución hasta 5 es él mismo y su suma con las soluciones anteriores, el conjunto de soluciones cambia a $\{3, 5, 8\}$, y por ultimo incluyendo el 8 el conjunto resultante es: $\{3, 5, 8, 11, 13, 16\}$. Una posible implementación seria:

```

1 | bool memo[10][5];
2 | void dp(vector<int> &V){
3 |     memset(memo, false, sizeof(memo));
4 |     for(int i = 0; i < V.size(); i++){
5 |         if(i){
6 |             for(int j = 1; j < 10; j++){
7 |                 if(memo[j][i - 1]){
8 |                     memo[j][i] = true;
9 |                     memo[j + V[i]][i] = true;
10 |                 }
11 |             }
12 |             memo[V[i]][i] = true;
13 |         }
14 |     }

```

En el anterior código se tienen dos ciclos, el primero recorriendo el vector V y el segundo buscando las soluciones anteriores para sumarles $V[i]$, como el caso base es cuando $i = 0$ entonces se debe omitir el segundo ciclo cuando i tiene este valor, ejecutando este código se obtendría todas las soluciones en la última columna, entonces si $memo[n][V.size() - 1]$ es verdadero entonces si existe un sub-conjunto que sume n , en caso contrario no es posible.

3. Problema de la mochila(Knapsack problem)

Este es un problema clásico de programación dinámica, consiste en lo siguiente, hay una mochila que tiene una capacidad limitada de peso que puede contener, y hay un grupo de objetos, cada uno tiene un valor y peso, el objetivo consiste en llenar la mochila con los objetos de tal forma que no se exceda su capacidad de peso y que el valor de los objetos que hay dentro de la mochila sea lo más alto posible. Por ejemplo:

| | | | | | | |
|-------|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 |
| W_i | 1 | 2 | 4 | 5 | 4 | 2 |
| V_i | 2 | 7 | 5 | 9 | 5 | 3 |

Sea $N = 6$ la cantidad de objetos, $C = 10$ la capacidad de la mochila, W_i el peso del i -ésimo objeto y V_i el valor del i -ésimo objeto, entonces las respuesta es 21, usando los objetos 0, 1, 3, 5 obtenemos un peso 10 con valor 21.

Para cada objeto existen dos opciones, tomarlo o dejarlo, entonces para n objetos existen 2^n posibles opciones, de todas ellas la solución será una configuración que no exceda el peso de la mochila y la suma de los objetos seleccionados sea el más alto, una solución probando todos los posibles casos tendrá complejidad $O(2^n)$, la cual es bastante alta, mientras que una solución

greddy no siempre va a funcionar.

Sin embargo es posible reducir bastante la complejidad trabajando con programación dinámica, sea x_i la capacidad usada al estar en el i -ésimo objeto, iniciando con $i = 0$ y $x_i = 0$, se repite el siguiente proceso: para cada objeto existe la decisión de no tomarlo y pasar al objeto $i + 1$ con $x_{i+1} = x_i$ o tomarlo y pasar al objeto $i + 1$ con $x_{i+1} = x_i + W_i$ siempre y cuando $x_i + W_i \leq C$, es decir que no se exceda la capacidad, entonces se puede escribir la siguiente función recursiva:

$$f(i, x) = \begin{cases} 0 & \text{if } i = N \\ \max(f(i + 1, x), f(i + 1, x + W_i) + V_i) & \text{if } x + W_i \leq C \\ f(i + 1, x) & \text{if } \textit{otro} \end{cases}$$

Al programar esta función recursiva se obtendrá una complejidad de $O(2^n)$, pero al agregar la tabla de memorización la complejidad se reduce al tamaño de la tabla, quedando en $O(N * C)$, y finalmente se puede obtener la siguiente implementación.

```
1 | int dp(int i, int x){
2 |     if(i == N) return 0;
3 |     if(memo[i][x] != -1) return memo[i][x];
4 |
5 |     if(x+W[i] <= C)
6 |         return memo[i][x] = max(dp(i+1,x), dp(i+1,x+W[i])+V[i]);
7 |     else
8 |         return memo[i][x] = dp(i+1,x);
9 | }
```

4. Bibliografía

<http://trainingcamp.org.ar/antiores/2017/clases.shtml>.
<http://programacioncompetitivaufps.github.io/>
<https://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/>
libro: competitive programming 3.