# Programación Dinámica

Wilmer Emiro Castrillón Calderón

31 de enero de 2020

## 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-esima del vector corresponda al número i-esimo de la sucesión. Para solucionar el problema con dp primero debemos identificar los sub-problemas, 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 $v[x] = fibo(x)$. A continuación se muestra el ejemplo en C++ calculando los primero 45 números de la sucesión:

```
int v[45];

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

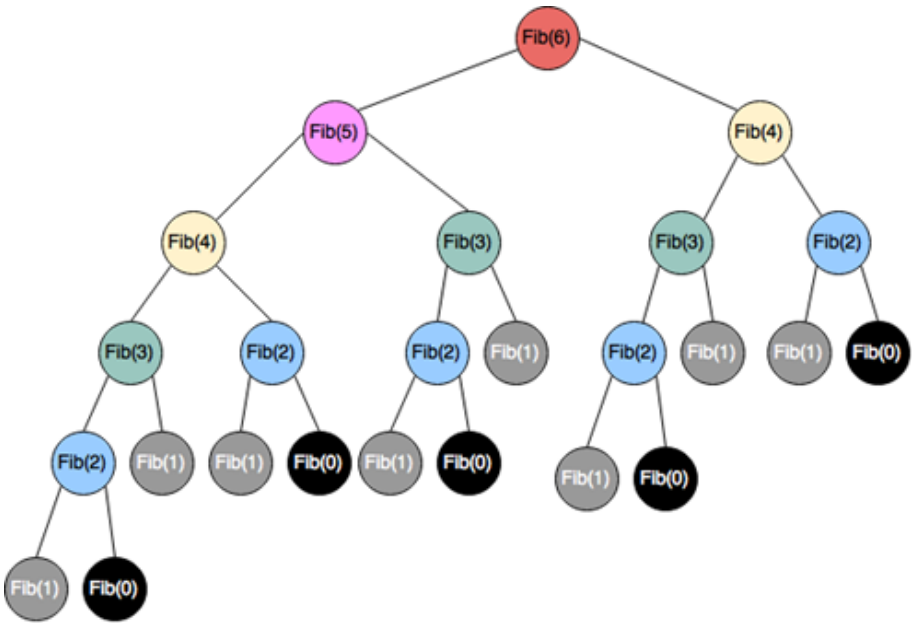
Aunque en este enfoque no utilizamos llamados recursivos, puede resultar mas difícil de implementar, pues por lo general es menos intuitivo.

**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:

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

Si ejecutamos este código para $fibo(6)$ 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 tiene como valor -1 entonces se debe calcular.

```
1  int v[45];
2  int fibo(int x){
3      if(v[x] != -1) return v[x];
4
5      if(x <= 1){
6          return v[x] = x;
7      }else{
8          return v[x] = fibo(x-1) + fibo(x-2);
9      }
10 }
```

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 $n$ x $m$ 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.

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 $n$ x $m$ 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] + máximo entre(camino óptimo hasta T[n-1][m], camino óptimo hasta T[n][m-1])*, y a su vez el óptimo para ellos se calcula de manera similar, el optimo para llegar hasta T[n-1][m] es *T[n-1][m] + máximo entre(camino óptimo hasta T[n-2][m], 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 optimo 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:

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

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 palabra 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(memo[i][j] != -1) return memo[i][j];
```

```
3
4       if(i==0 && j==0)
5           return memo[0][0] = T[0][0];
6       if(i == 0)
7           return memo[0][j] = T[0][j] + f(0, j-1);
8       if(j == 0)
9           return memo[i][0] = T[i][0] + f(i-1, 0);
10
11      return memo[i][j]= T[i][j] + max(f(i-1, j), f(i, j-1));
12  }
```

para conocer el resultado solo se debe imprimir $f(n-1, m-1)$. para la solución con Buttom-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];
3       for(int i = 1; i < n; i++)//llenar filas
4           memo[i][0] += memo[i-1][0] + T[i][0];
5
6       for(int i = 1; i < m; i++)//llenar columnas
7           memo[0][i] += memo[0][i-1] + T[0][i];
8
9       for(int i = 1; i < n; i++){
10          for(int j = 1; j < m; j++) {
11              memo[i][j] += max(memo[i - 1][j], memo[i][j - 1]) + T[i][j];
12      }
13    }
14  }
```

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 exponencial de fuerza bruta.

### Suma de subconjuntos:

También conocido como *Sub-SetSum* 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 1, 2, 5 es posible formar los números 1, 2, 3, 5, 6, 7, 8 sumando todos los elementos de cada subconjunto.

Utilizando un algoritmo de BackTraking para probar todos los posibles subconjuntos se obtendría una complejidad O($2^n$), siendo bastante alta. 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: $n = v[k_1] + v[k_2] + ... + v[k_n]$.

En este problema el resultado deberá ser 'verdadero' o 'falso', mostrando si es posible sumar $n$ con algún subconjunto de $V$, con los subproblemas establecidos se puede ahora definir los casos base, si en algún momento se llega a $x_n = 0$ entonces la solución global sera 'verdadero', pero si en algún momento usamos todos los elementos en $V$ o encontramos $x_n < 0$ entonces la solución en ese subproblema sera 'falso',

Con un enfoque Buttom-Up se debe buscar todas las posibles sumas que se pueden formar con los números de un vector $v$ de tamaño $n$, por facilidad para este ejemplo vamos a indexar desde 1, para dar solución primero vamos a identificar los sub-problemas, la solución general incluye sumar el elemento $v[n]$ a todos los números encontrados hasta $v[n-1]$, y este a su vez depende de encontrar $v[n-2]$ 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 cojunto de soluciones hasta ahora es $\{v[1], v[2], v[1]+v[2]\}$, ahora la solución hasta $v[3]$ sera $\{v[1], v[2], v[1]+v[2], v[1]+v[3], v[2]+v[3], v[1]+v[2]+v[3]\}$, y así sucesivamente hasta $v[n]$, en este conjunto los numeros repetidos seran descartados.

Por ejemplo supongamos que tenemos {3, 5 y 8}, la solución hasta 3 es él mismo, nuestro conjunto de soluciones hasta ahora es {3}, las soluciones hasta 5 es él mismo y su suma a las soluciones anteriores, nuestro conjunto de soluciones ahora es {3, 5, 8} y por ultimo incluyendo el 8, nuestro conjunto de soluciones queda de la siguiente forma: {3, 5, 8, 11, 13, 16}.

Esto se puede resolver recursivamente con Top-Down pero quedara como tarea para el lector, áca se usara una solución iterativa usando Buttom-Up, para tabla de memorización usaremos una matriz con una cantidad de filas igual o mayor a la longitud del vector y de columnas igual al resultado más grande que se puede obtener, sera de tipo *bool* pues el problema solo requiere decir si un numero se puede formar o no, cada fila i representara las soluciones encontradas hasta el $v[i]$.

```
1  bool memo[5][50];
2  void dp(vector<int> &num){
3      memset(memo, false, sizeof(memo));
4
5      for(int i = 0; i < num.size(); i++){
6          if(i){
7              for(int j = 1; j < 50; j++){
8                  if(memo[i - 1][j]){
9                      memo[i][j] = true;
10                     memo[i][j + num[i]] = true;
11                 }
12             }
13         }
14         memo[i][num[i]] = true;
15     }
16 }
17 */
```

En el anterior codigo tomamos el vector *num* como el conjunto de números $v$, tenemos dos ciclos el primero recorrera el vector *num* y el segundo buscara las soluciones anteriores y le sumara a esa solución $num[i]$, como el caso base es cuando $i = 0$ entonces debemos saltarnos el segundo ciclo, pues resulta innecesario porque no hay soluciones anteriores, para eso ponemos un condicional que ejecute el segundo ciclo si $i$ es mayor a 0, y por último si queremos saber si un número $m$ se puede formar entonces cosultamos en la última fila de la tabla de memorización, si $memo[num.size() - 1][m]$ es verdadero entonce si existe un sub-conjunto que sume $m$, en caso contrario no es posible formar esa suma.

### Otros problemas clásicos:

**Problema de la mochila:** teniendo una mochila de capacidad $n$, un conjunto de objetos con peso $m$ y una ganancia $c$, se debe encontrar la máxima ganancia que se puede guardar en

la mochila sin exceder su capacidad.

**Max Range Sum:** Dado un arreglo X de números enteros positivos y negativos, se debe encontrar la máxima suma que se puede lograr sumando posiciones consecutivas del arreglo.

**Dominos en un tablero de 2*N:** consiste en encontrar la cantidad total de maneras en las que se puede llenar un tablero de tamaño 2*N con fichas de domino de tamaño 2X1, el tablero se debe llenar completamente sin tener espacios vacíos.

# 2. Problema de la mochila(Knapsack problem)

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