Programación Dinámica

Bach. Rodolfo Mercado Gonzales
Universidad Nacional de Ingeniería

Programación Dinámica

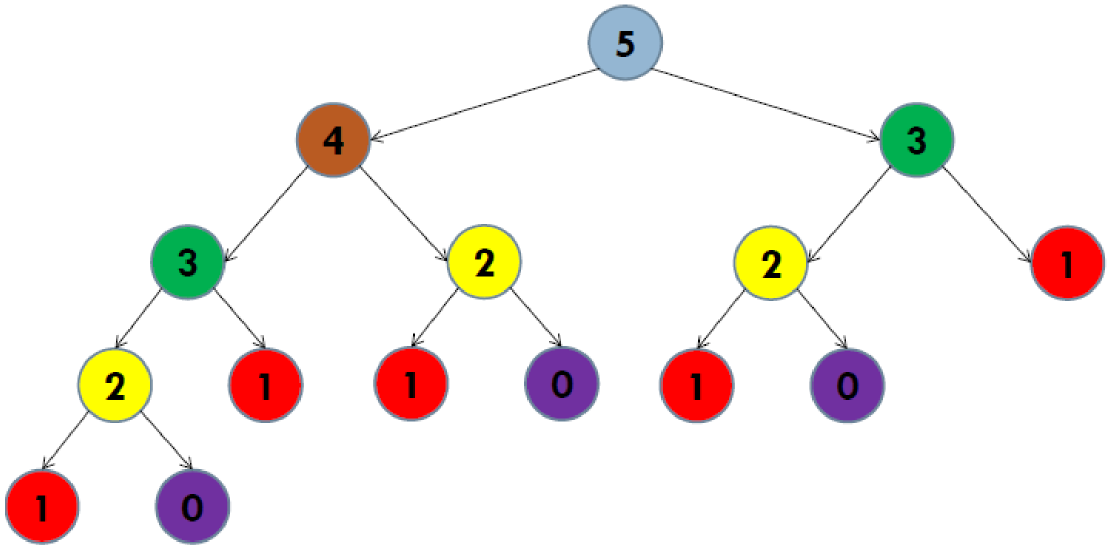
Ya sabemos que la serie de Fibonacci está formada por los siguiente números:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

Además lo hemos implementado recursivamente, entonces tratemos de **hallar el Fibonacci de la posición 50.**

¿ Qué pasa ?

Programación Dinámica



Programación Dinámica

- Llamamos muchas veces a la función con los mismos parámetros.
- El algoritmo repite acciones realizadas en el pasado (no tiene memoria).

¿Cómo lo mejoramos?

Programación Dinámica

La programación dinámica consiste en agregarle memorización a nuestra definición recursiva.

Programación Dinámica

Estrategia a seguir:

- ❑ Dar una definición recursiva a un problema (resolver el problema en base a subproblemas del mismo tipo).
- ❑ Resolver cada subproblema de la misma manera hasta llegar a un caso base.
- ❑ Guardar el resultado de cada estado del problema la primera vez que se calcula.

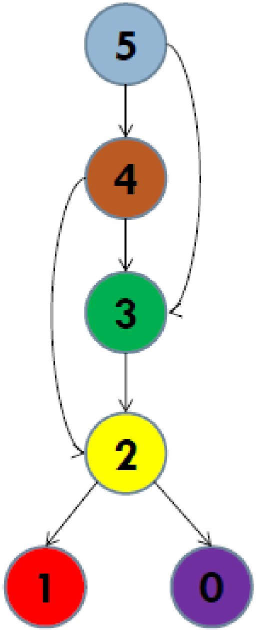
Estructura básica de una programación dinámica

```
int memo[100][100][100];
int dp(int parameter1, int parameter2, int parameter3) {
  if (case_base) {
    return resultado_base;
  }
  if (memo[parameter1][parameter2][parameter3] != -1) return memo[parameter1][parameter2][parameter3];
  int &ans = memo[parameter1][parameter2][parameter3] = 0;  //inicializando la respuesta
  /*
    Logica dentro del DP
  */
  return ans;
}
```

$$O(parameter1 * parameter2 * parameter3 * LogicaDP)$$

Programación Dinámica

Logramos reducir el número de llamadas a la función.



Números Combinatorios

Los números combinatorios se representan de la forma:

$$\binom{n}{k}$$

Cuentan el número de formas que se pueden escoger k elementos de un conjunto de n elementos.

Números Combinatorios

Por ejemplo si queremos escoger dos números del siguiente conjunto de tamaño 4.

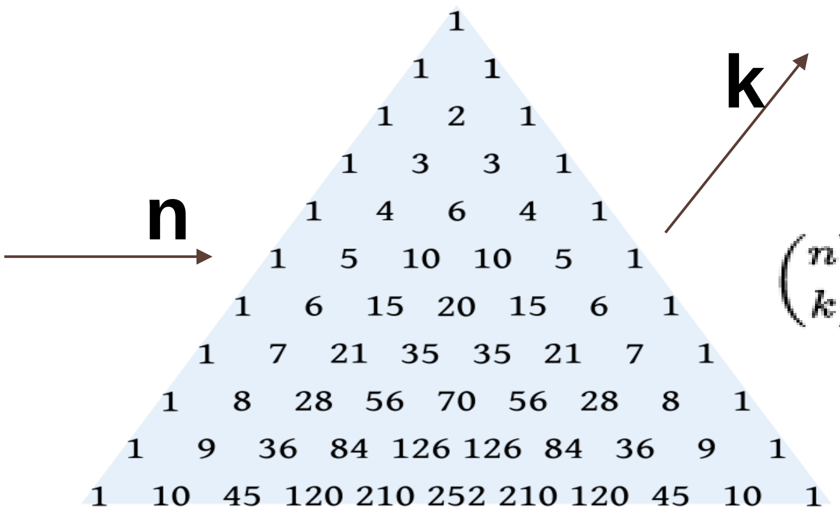
{ 4, 10, 8 , 9 }

Existen 6 posibles grupos como resultado:

{4, 10} {4, 8 } {4, 9}
{10, 8} {10, 9 }   { 8, 9}

Números Combinatorios

Para calcular un número combinatorio también lo podemos hacer recursivamente (Triángulo de Pascal):



$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Subset Sum

Dado un conjunto A de enteros positivos y un entero positivo S, se desea saber si existe algún subconjunto de A cuya suma sea S. Por ejemplo:

A = { 1 , 4 , 2 }

Subconjuntos de A: { 1 } { 4 } {2 } {1, 4} { 1, 2 } { 4, 2 } {1, 4, 2}

S = 7 ? Sí con el subconjunto {1, 4, 2}.

S = 0 ? Sí con el subconjunto { }

Subset Sum

f(n, s) nos dirá si es posible encontrar un subconjunto de los **n** primeros elementos del arreglo, tal que sume **s**. Devolverá true o false.

$$f(n, s) = f(n-1, s) \; or \; f(n-1, s - A[n-1])$$

KnapSack Problem

Dado un conjunto A de ítems y una mochila (knapsack) que permite cargar un máximo W peso, cada ítem tiene un peso ‘p’ y un beneficio ‘b’. Se debe seleccionar un subconjunto de ítems tal que la suma de sus pesos no supere el máximo peso posible a cargar en la mochila y maximizar el beneficio. Por ejemplo:

W = 15
A1 = 10 peso – 15 beneficio
A2 = 2 peso – 1 beneficio
A3 = 5 peso – 12 beneficio
A4 = 13 peso – 20 beneficio
A5 = 1 peso – 1 beneficio

KnapSack Problem

f(n, w) nos dirá el máximo beneficio de obtener un subconjunto de los **n** primeros elementos del arreglo, con una capacidad máximo de **W**. La recursión seria.

$$f(n,w) = \max(f(n-1,w)\,, \mathrm{B}[\mathrm{n}-1] + \ f(n-1, \mathrm{W} - P[n-1]))$$

Problemas

SPOJ MAIN72 – Subset sum

SPOJ KNAPSACK – The knapsack Problem

Longest Common Subsequence (LCS)

Dado dos secuencias (cadena o arreglo) A y B, se desea hallar la longitud que tiene la subsecuencia más larga, común a ambas. Por ejemplo:

A = "abc”
B = “bac”

subsecuencias de A = { “a”, “b”, “c”, “ab”, “ac”, “bc”,“abc”}
subsecuencias de B = { “a”, “b”, “c”, “ba”, “bc”, “ac”,“bac”}

subsecuencias comunes de mayor tamaño: { “bc”, “ac”}

Entonces el LCS = 2

Longest Common Subsequence (LCS)

f(n, m) : nos devolverá la longitud del LCS para la cadena formada por los n primeros caracteres de A y la cadena con los m primeros caracteres de B.

$$f(n,m)\begin{cases} 1 + f(n-1, m-1)\,, & A_{n-1} = B_{n-1} \\ \max\ (f(n, m-1), f(n-1, m)) & \end{cases}$$

Problemas

UVA 10192 - Vacation

UVA 10066 – The Twin Towers

DP Iterativo

- ❑ Para transformar un DP recursivo a iterativo necesitamos ver el sentido en que se está llenando nuestra tabla de memorización, para ello tomamos como referencia uno de los parámetros de la recursión.

- ❑ Tomemos como ejemplo el LCS :

- En el LCS si tomamos como referencia el tamaño de la primera cadena ($\boldsymbol{n}$), vemos que depende de $\boldsymbol{n-1}$ y $\boldsymbol{n}$ mismo, entonces este parámetro se recorre menor a mayor.

- Asimismo en el LCS vemos que $\boldsymbol{n}$ también depende $\boldsymbol{n}$ mismo, para este caso vemos que $\boldsymbol{m}$ depende de $\boldsymbol{m-1}$, entonces este parámetro también debemos recorrerlo de menor a mayor.

DP Iterativo

- Así quedaría el LCS de forma iterativa:

```
for( int j = 0; j <= m; ++j ) dp[ 0 ][ j ] = 0;
for( int i = 0; i <= n; ++i ) dp[ i ][ 0 ] = 0;

for( int i = 1; i <= n; ++i ){
    for( int j = 1; j <= m; ++j ){
        if( a[ i - 1 ] == b[ j - 1 ] ) dp[ i ][ j ] = 1 + dp[ i - 1 ][ j - 1 ];
        else dp[ i ][ j ] = max( dp[ i - 1 ][ j ], dp[ i ][ j -1 ] );
    }
}
```

Reconstrucción de un DP

Podemos usar la misma recursión para reconstruir una solución, solo que en cada estado debemos escoger el camino óptimo.

Para el LCS quedaría así :

```
void rec( int n, int m ){
    if( n == 0 || m == 0 ) return;
    if ( a[ n - 1 ] == b[ m - 1 ] ){
        rec( n - 1, m - 1 );
        cout << a[ n - 1 ];  //letra que hizo match
    }
    else{
        if ( f( n - 1, m ) > f( n, m - 1 ) ) rec( n - 1, m ); // fue el óptimo ?
        else rec( n, m - 1 );
    }
}
```

¡ Good luck and have fun !