Erick Vargas

DP

Contents

1 Programación dinámica **2**
1.1 Fibonacci 2
1.2 Coeficiente binomial 4
1.3 Problema 4
1.3.1 Solución 5

1 Programación dinámica

Partimos de una solución recursiva bruta y podemos hacer uso de una función de memorización. El ejemplo más clásico es el de Fibonacci

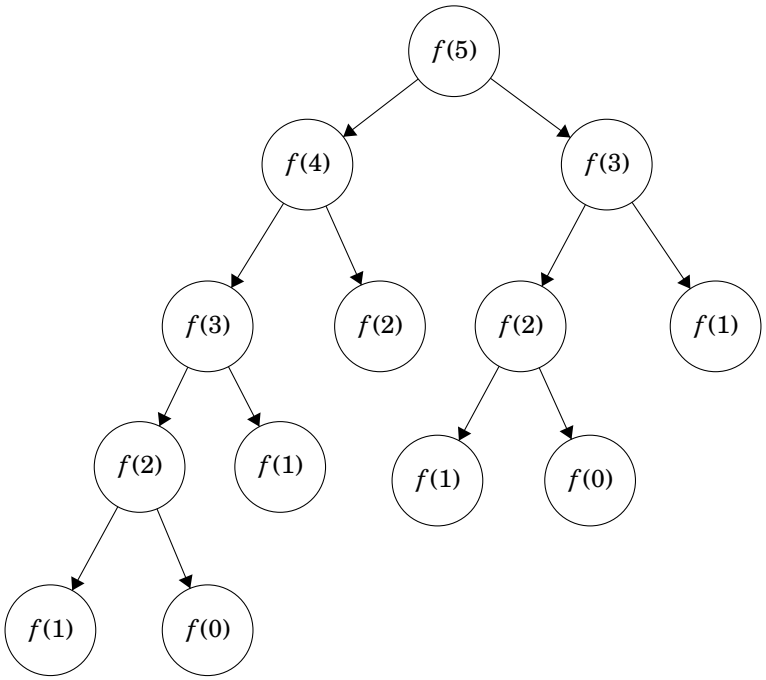
1.1 Fibonacci

$$fibonacci(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & , n \geq 2 \end{cases}$$

En código

```
int fibonacci( int n ){
    if( n == 0 )
        return 0;
    if( n == 1 )
        return 1;
    return fibonacci( n - 1 ) * fibonacci( n - 2 );
}
```

Si dibujamos las llamadas recursivas como un árbol tenemos lo siguiente



Como podemos observar hay llamadas recursivas que se repiten varias veces como ejemplo tenemos la llamada recursiva con un valor de 3, o 2. ¿Podemos evitar esto?

```
int memoria[ 100000 ];
.
.
.
int fibonacci( int n ){
    if( n == 0 )
        return 0;
    if( n == 1 )
        return 1;
    if( memoria[ n ] != -1 )
        return memoria[ n ];
    return memoria[ n ] = ( fibonacci( n - 1 ) * fibonacci(
        n - 2 ) );
}

int main(){
    memset( memoria, -1, sizeof(memoria) );
    .
    .
    .
}
```

La complejidad la podemos calcular viendo los estados, en nuestro caso es el tamaño de la memoria, por ejemplo si llamamos a fibonacci de 10 siempre ten-

emos el mismo resultado, por tanto multiplicamos los estados por la complejidad de la función en nuestro caso es $10^5 * constante$

1.2 Coeficiente binomial

$$Coef\_Bin(n,k) = \begin{cases} 1 & , n = k \\ 1 & , k = 0 \\ Coef\_Bin(n-1,k-1) + Coef\_Bin(n-1,k) & , k \leq n \end{cases}$$

Si programamos esta función tenemos lo siguiente:

```
int coef_bin( int n, int k ){
    if( n == k || k == 0 )
        return 1;
    return ( coef_bin( n - 1, k - 1 ) + coef_bin( n - 1, k )
         );
}
```

Si aplicamos DP

```
memoria[ 1000 ][ 1000 ];
.
.
.
int coef_bin( int n, int k ){
    if( n == k || k == 0 )
        return 1;
    if( mem[ n ][ k ] != -1 )
        return memoria[ n ][ k ];
    return memoria[ n ][ k ] = ( coef_bin( n - 1, k - 1 ) +
        coef_bin( n - 1, k ) );
}
```

En complejidad tenemos $O(n*k)$ y sin la función de memorización tenemos $O(2^n)$

1.3 Problema

Dado un grid de nxm, cada casilla tiene un número. Obtener un camino de la fila 1 a la fila n con suma máxima, ejemplo:

3	5	10
6	4	3
2	1	0

Como restricciones tenemos que $n, m \leq 10^3$ y además $A_{i,j} \leq 10^6$ Además solo nos podemos mover de la siguiente manera

1.3.1 Solución

```
int max_sum( int f, int c ){
    //Primero comprobamos si es un caso válido
    if( c < 0 || c >= m )
        return MIN_INT //MIN_INT es como un menos infinito
    //Caso base
    if( f == n )
        return grid[ f ][ c ];
    int A = max_sum( f + 1, c - 1);
    int B = max_sum( f + 1, c );
    int C = max_sum( f + 1, c + 1 );
    //No exite una función max para 3 elementos, en dado
        caso podemos usar vector es decir ponemos max({A, B,
         C})
    return max( A, max(B, C) ) + grid[ f ][ c ];
}
```

Ahora bien ¿Cómo reducimos la complejidad de el código anterior? Debemos aplicar programación dinámica, primeramente ¿Cuál es el tamaño máximo de nuestra función de memoria? en nuestro caso es 1005 * 1005

```
int memoria[ 1005 ][ 1005 ];
int max_sum( int f, int c ){
    //Primero comprobamos si es un caso válido (que no esté
        fuera de rango)
    if( c < 0 || c >= m )
        return MIN_INT //MIN_INT es como un menos infinito
    //Caso base
    if( f == n )
        return grid[ f ][ c ];

    if( memoria[ f ][ c ] != MIN_INT )
        return memoria[ f ][ c ]

    int A = max_sum( f + 1, c - 1);
    int B = max_sum( f + 1, c );
    int C = max_sum( f + 1, c + 1 );
    //No exite una función max para 3 elementos, en dado
        caso podemos usar vector es decir ponemos max({A, B,
         C})
    return memoria[ f ][ c ] = max( A, max(B, C) ) + grid[ f
         ][ c ];
}
```