Erick Vargas

Club algoritmia (ESCOM)

Contents

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

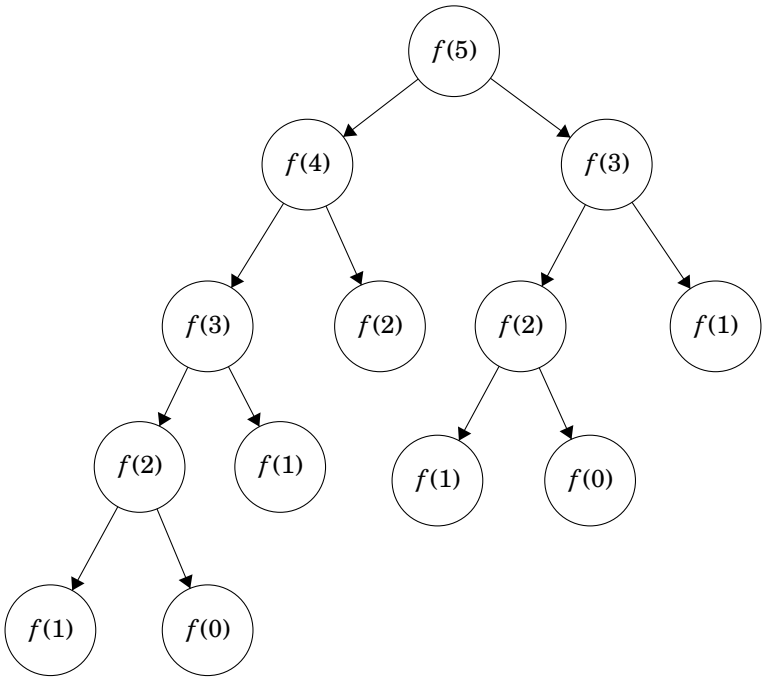
1.1 Fibonacci

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

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. Supongamos que estamos en el recuadro con el color gris más oscuro, estando en esa posición podemos movernos a cualquiera de los tres rectangulos con color gris más claro, es decir, podemos movernos hacía $(f+1,c),(f+1,c-1),(f+1,c+1)$

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 -1 //-1 es un valor bandera que identifica si
             ya visitamos o no la casilla
    //Caso base
    if( f == n )
        return grid[ f ][ c ];

    if( memoria[ f ][ c ] != -1 )
        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 ];
}
```

1.4 Problema

Del ejercicio anterior que sucede si podemos añadir ¿números negativos? Podemos hacer uso de otro arreglo auxiliar de booleanos, en el que marcamos si ya visitamos o no una casilla

2 Problemas clásicos de DP

2.1 Mochilas

Vas a una tienda y quieres comprar algunos productos, digamos que hay n productos y cada producto tiene un precio y tenemos M pesos para gastar. Queremos comprar la mayor cantidad de productos gastando lo más posible.
Restricciones:

- N productos $N \leq 1000$
- M Pesos $M \leq 1000$
- $A_i \leq 1000$

Ejemplo:
N = 3, M = 8, y tenemos los productos A = 3, 4, 6 nuestro solución es comprar 3, 4 el precio es 7 y nos llevamos 4 productos.

Para poder solucionar este problema debemos de calcular el caso de probar y no probar el i-ésimo elemento. Es decir generamos todos los posibles subconjuntos del arreglo:

- ()
- (3)
- (3, 4)
- (3, 6)
- (4)
- (4, 6)

- (6)
- (3, 4, 6)

```
int max_sum( int indice, int pesos_restantes ){
    //Validación, n es el número de elementos del arreglo A
    if( indice >= n )
        return 0;
    //Como no es una respuesta válida para descartarla
        regresamos un menos infinito
    if( pesos_restantes < 0)
        return MIN_INT;
    //Recursivamente puedo tomar o no tomar un elemento
    int tomar = max_sum( indice + 1, pesos - A[ indice ]) +
        A[ indice ];
    int no_tomar = max_sum( indice + 1, pesos);
    return max( tomar, no_tomar );
}
```

Esta solución es la más bruta si queremos aplicar programación dinámica ¿Cuáles son los estados que tenemos? En este caso tendremos los índices y la cantidad de pesos restantes. Es decir:

```
.
.
.
int memoria[ 1005 ][ 1005 ];
bool visitado[ 1005 ][ 1005 ];
.
.
.
int max_sum( int indice, int pesos_restantes ){
    //Validación, n es el número de elementos del arreglo A
    if( indice >= n )
        return 0;
    //Como no es una respuesta válida para descartarla
        regresamos un menos infinito
    if( pesos_restantes < 0)
        return MIN_INT;

    if( visitado[ indice ][ pesos_restantes ] )
        return memoria[ indice ][ pesos_restantes ];

    //Recursivamente puedo tomar o no tomar un elemento
    int tomar = max_sum( indice + 1, pesos - A[ indice ]) +
        A[ indice ];
    int no_tomar = max_sum( indice + 1, pesos);
    //Como no se ha visitado el cálculo actual lo marcamos
        como visitado
    visitado[ indice ][ pesos_restantes ] = true;
    //Almacenamos el valor calculado en nuestra memoria
    return memoria[ indice ][ pesos_restantes ] = max( tomar
        , no_tomar );
}
```

Con esta solución tenemos una complejidad de $O(n^2)$ mientras que la solución bruta tiene una complejidad de $O(2^n)$

2.2 Mochila clásica

Queremos meter N piedras a una mochila cuya capacidad es de M y cada piedra tiene un valor V, queremos meter la mayor cantidad de piedras respetando que el peso límite soportado por la mochila no se exceda y se maximice el valor de las piedras dentro de la mochila
Restricciones:

- N piedras $N \leq 1000$
- M capacidad de la mochila $M \leq 1000$
- V_i valor de la piedra $V_i \leq 10^9$
- W_i peso de la piedra $W_i \leq 1000$

Ejemplo:
N = 3, M = 50;
V = [60, 100, 120]
W = [10, 20, 30]
Para este caso tenemos como solución tomar la segunda y tercer piedra teniendo un valor de 220 y un peso de 50, el cual puede soportar perfectamente nuestra mochila.

Podemos encontrar una solución siguiendo más o menos lo mismo que el ejercico anterior, vamos a tratar de maximizar el valor tomando o no tomando el i-ésimo peso

```
int sum_max( int indice, int capacidad ){
    if( indice >= n )
        return 0;
    if( capacidad < 0 )
        return MIN_INT;
    //Si lo tomo cuál es el valor máximo que podemos tener
        con la respuesta actual?
    int tomar = sum_max( indice + 1, capacidad - W[ indice ]
         ) + V[ indice ];
    int no_tomar = sum_max( indice + 1, capacidad );
    return max(tomar, no_tomar);
}
```

Si aplicamos programación dinámica

```
.
.
.
int memoria[ 1005 ][ 1005 ];
.
.
.
int sum_max( int indice, int capacidad ){
    if( indice >= n )
        return 0;
    if( capacidad < 0 )
```

```
            return MIN_INT;
        if( memoria[ indice ][ capacidad ] != MIN_INT )
            return memoria[ indice ][ capacidad ];
        //Si lo tomo cuál es el valor máximo que podemos tener
            con la respuesta actual?
        int tomar = sum_max( indice + 1, capacidad - W[ indice ]
             ) + V[ indice ];
        int no_tomar = sum_max( indice + 1, capacidad );
        return memoria[ indice ][ capacidad ] = max(tomar,
            no_tomar);
    }
```

Ahora bien debemos responder ¿dónde está la respuesta a nuestro problema?

```
    .
    .
    .
    int main(){
        .
        .
        .
        //Si indexamos desde cero aquí estará la solución
        cout << sum_max( 0, M );
        .
        .
        .
        return 0;
    }
```

2.3 Problema

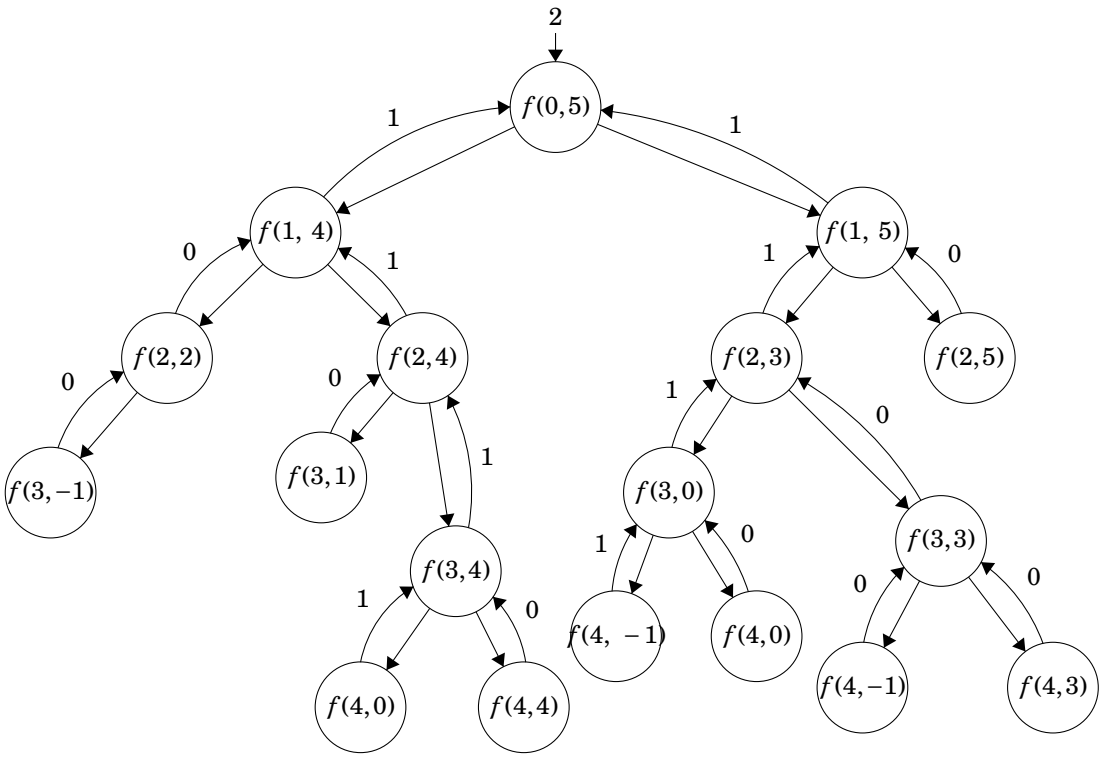
Imagina que compraste algo en algúna tienda, supermercado, etc, tienes que pagar con N monedas, pero quieres saber ¿De cuantas formas utilizando cualquiera de esas N monedas puedo dar el cambio (M)?
Restricciones:

- N monedas $N \leq 1000$
- M cambio $N \leq 1000$
- C_i Monedas $1leqC_i \leq 1000$

Ejemplo: N = 4, M = 5
C = [1, 2, 3, 4] //Indexado desde cero

- (0, 3) = 1 + 4 = 5
- (1, 2) = 2 + 3 = 5

Es decir nuevamente debemos de considerar el problema como tomar o no tomar



Como solución bruta ¿Qué podemos programar?

```
.
.
.
int cuenta( int indice, int cambio ){
    //Verificamos que no nos salgamos del rango
    if( indice == n ){
        //Si llegamos al límite verificamos si lo que
            tenemos es una solución válida
        if( cambio == 0 )
            //Como encontramos una solución la contamos
            return 1;
        //Como no hay solución retornamos cero
        return 0;
    }
    //Si el cambio es negativo no es una combinación válida
    if( cambio < 0 )
        return 0;
    //Intentamos tomando el elemento actual
    int tomar = cuenta( indice + 1, cambio - C[ indice ] );
    int no_tomar = cuenta( indice + 1, cambio );
    return tomar + no_tomar;
}
```

Si añadimos programación dinámica tenemos lo siguiente

```
.
.
.
int memoria[ 1005 ][ 1005 ];
bool visitado[ 1005 ][ 1005 ];
.
.
.
int cuenta( int indice, int cambio ){
    //Verificamos que no nos salgamos del rango
    if( indice == n ){
        //Si llegamos al límite verificamos si lo que
            tenemos es una solución válida
        if( cambio == 0 )
            //Como encontramos una solución la contamos
            return 1;
        //Como no hay solución retornamos cero
        return 0;
    }
    //Si el cambio es negativo no es una combinación válida
    if( cambio < 0 )
        return 0;

    if( visitado[ indice ][ cambio ] )
        return memoria[ indice ][ cambio ];
    //Intentamos tomando el elemento actual
    int tomar = cuenta( indice + 1, cambio - C[ indice ] );
    int no_tomar = cuenta( indice + 1, cambio );
    visitado[ indice ][ cambio ] = true;
    return memoria[ indice ][ cambio ] = (tomar + no_tomar);
}
```