



Programación Dinámica



Bach. Rodolfo Mercado Gonzales
Universidad Nacional de Ingeniería

Programación Dinámica

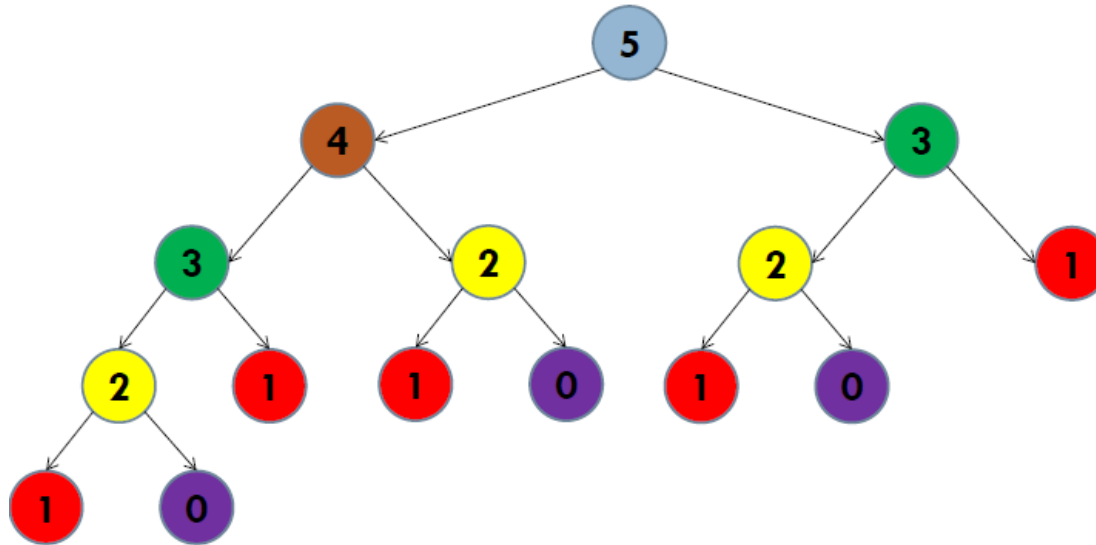
Ya sabemos que la serie de Fibonacci está formada por los siguiente números:

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:

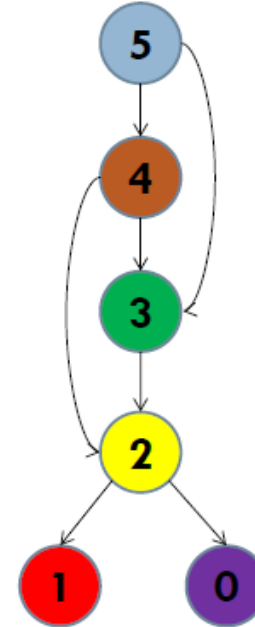
- ❑ Dar una definición recursiva a un problema (resolver el problema en base a subproblemas del mismo tipo).
- ❑ Resolver cada subproblema de la misma manera hasta llegar a un caso base.
- ❑ Guardar el resultado de cada estado del problema la primera vez que se calcula.

Programación Dinámica

En programación dinámica los subproblemas se repiten (sobrelapan), algo que no se da en el enfoque “divide y vencerás”

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.

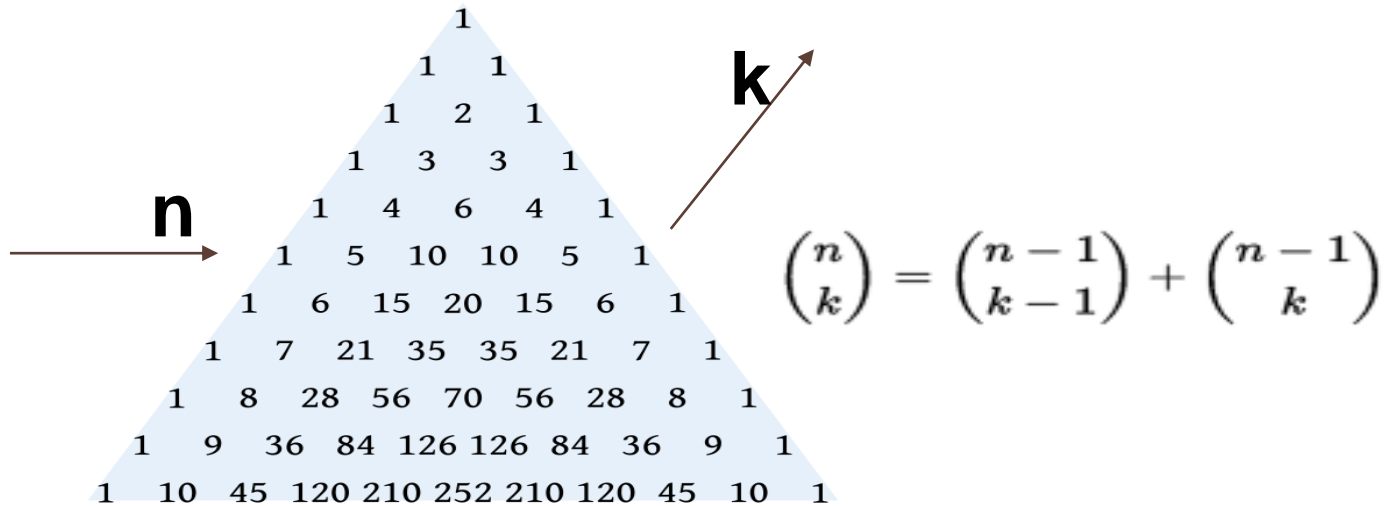
$\{ 4, 10, 8, 9 \}$

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):



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.

$$f(n, s) = f(n - 1, s) \text{ or } f(n - 1, s - A[n - 1])$$

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:

$A = \text{"abc"}$

$B = \text{"bac"}$

subsecuencias de $A = \{ \text{"a"}, \text{"b"}, \text{"c"}, \text{"ab"}, \text{"ac"}, \text{"bc"}, \text{"abc"} \}$

subsecuencias de $B = \{ \text{"a"}, \text{"b"}, \text{"c"}, \text{"ba"}, \text{"bc"}, \text{"ac"}, \text{"bac"} \}$

subsecuencias comunes de mayor tamaño: $\{ \text{"bc"}, \text{"ac"} \}$

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.

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

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 (n), vemos que depende de $n - 1$ y n mismo, entonces este parámetro se recorre menor a mayor.
 - Asimismo en el LCS vemos que n también depende n mismo, para este caso vemos que m depende de $m - 1$, 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 ] );
    }
}
```

DP Iterativo con ahorro de memoria

Algunas veces podemos reducir la memoria que usa nuestro DP, analizando nuestro parámetro de referencia y notando si es posible que solo tome dos valores (tener cuidado cuando el valor de los casos base depende del parámetro de referencia)

Para el LCS quedaría así :

```
int dp[ 2 ][ M ];

for( int j = 0; j <= m; ++j ) dp[ 0 & 1 ][ j ] = 0;
for( int i = 0; i <= n; ++i ) dp[ i & 1 ][ 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 & 1 ][ j ] = 1 + dp[ (i - 1) & 1 ][ j - 1 ];
        else dp[ i & 1 ][ j ] = max( dp[ (i - 1) & 1 ][ j ], dp[ i & 1 ][ 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 );
    }
}
```

Problemas

UVA 10192 - Vacation

UVA 10066 – The Twin Towers

Edit Distance

El objetivo es hallar el mínimo número de operaciones requeridos para transformar una cadena A en otra B. Las operaciones permitidas son insertar, eliminar y sustituir un carácter de la primera cadena por cualquier otro.

Ejm:

A = "bac"

B = "abdc"

Aquí se puede hacer como mínimo 2 operaciones : insertar una 'a' al inicio y luego reemplazar la última 'a' por una 'd'.

Edit Distance (Levenshtein distance)

f(n, m) : nos devolverá el edit distance para la cadena formada por los n primeros caracteres de A y la cadena con los m primeros caracteres de B.

$$f(n, m) \begin{cases} f(n-1, m-1), & A_{n-1} = B_{m-1} \\ \min \begin{cases} 1 + f(n, m-1), & \text{insertar} \\ 1 + f(n-1, m), & \text{eliminar} \\ 1 + f(n-1, m-1), & \text{reemplazar} \end{cases} \end{cases}$$

Problemas

SPOJ – Edit distance

UVA 526 – String Distance and Transform Process

DP Sobre Dígitos

Se denomina así a los problemas que tienen el siguiente tipo de pregunta:

Hallar la cantidad de números entre A y B que satisfacen cierta propiedad

DP Sobre Dígitos

Podemos definir una función f , tal que:

$f(n)$: cantidad de números entre 0 y n que cumplen cierta propiedad

Por lo tanto la solución al problema sería : $f(B) - f(A - 1)$

DP Sobre Dígitos / Comparando Números

Sea la representación decimal de dos números A y B :

$$A = a_1 a_2 a_3 \dots a_n \quad B = b_1 b_2 b_3 \dots b_m$$

Podemos decir que $A < B$ si se cumple alguno de los casos :

- $n < m$
- $n = m$ y \exists una posición i / $a_i < b_i$ y $a_j = b_j \quad \forall j < i$

DP Sobre Dígitos / Comparando Números

Si hacemos que los números tengan la misma longitud, completando con **0's** a la izquierda cuando hace falta, entonces ahora sólo nos importa el segundo caso de comparación.

0000000

0000005

0500000

3000000

DP Sobre Dígitos

Ahora podemos hallar la función $f(n)$, que encuentra todos los números menores o iguales a n que cumplen cierta propiedad, utilizando la programación dinámica.

Formamos cada número iterando por los dígitos de n y reemplazándolos por un dígito entre 0 y 9 según sea el caso y si al final este nuevo número cumple la propiedad, incrementamos nuestra respuesta en 1 .

1 9 **8** 7 6

1 9 **6** $d_4 d_5$ $0 \leq d_4 \leq 9$

1 9 **8** 7 6

1 9 **8** $d_4 d_5$ $0 \leq d_4 \leq 7$

DP Sobre Dígitos

Sea $s = d_1 d_2 d_3 \dots d_L$ (conviene verlo como una cadena)

Forma general:

$$f(\mathbf{s}, \mathbf{pos}, \mathbf{lower}, \dots)$$

s : cadena que representa al número del cual no nos debemos exceder (puede ser una variable global)

pos : entero que representa la posición de cada uno de los dígitos de s (de izq a der).

$lower$: booleano que indica que el número que estamos formando ya es menor que s .

DP Sobre Dígitos

$$f(s, pos, lower, \dots) = \begin{cases} \sum_{i=0}^9 f(s, pos + 1, lower, \dots), & lower = 1 \\ \sum_{i=0}^{d_{pos}} f(s, pos + 1, i < d_{pos} ? 1 : 0, \dots), & lower = 0 \end{cases}$$

DP Sobre Dígitos

Calcular la cantidad de números entre $[A, B]$ usando programación dinámica ($1 \leq A, B \leq 10^9$)

Por ejemplo: $A = 5$, $B = 103$

Rpta: 99

DP Sobre Dígitos

En este caso no nos piden que cumpla ninguna propiedad, así nuestra función quedaría:

$f(pos, lower)$

$f(0,0)$ nos daría la cantidad números entre 0 y n.

```
ll dp[ 10 ][ 2 ];
string s; // número leído como string

ll f( int pos, int lower ){
    if( pos == s.size() ) return 1;
    ll &ret = dp[ pos ][ lower ];
    if( ret != -1 ) return ret;
    ret = 0LL;
    if( !lower ){ // el número formado todavía no es menor que s
        for( int i = 0; i <= ( s[pos] - '0' ); ++i ){
            if( i == ( s[pos] - '0' ) ) ret += f( pos + 1, lower );
            else ret += f( pos + 1, 1 );
        }
    }
    else{ // el número formado de hecho será menor que s
        for( int i = 0; i <= 9; ++i ){
            ret += f( pos + 1, lower );
        }
    }
    return ret;
}

int main(){
    s = "103";
    memset( dp, -1, sizeof( dp ) );
    cout << f( 0, 0 ) << endl;
}
```


DP Sobre Dígitos

Calcular la suma de todos los dígitos que se usan al escribir los números del 1 a n ($1 \leq n \leq 10^9$)

Por ejemplo: $n = 10$

Entonces: $1 + 2 + 3 + 4 + \dots + 9 + 1 + 0 = 46$

DP Sobre Dígitos

$f(s, pos, lower, sum)$

```
string s; // número leído como string
ll memo[ 10 ][ 2 ][ 90 ];

ll f( int pos, int lower, int sum ){
    if( pos == s.size() ) return sum;
    ll &ret = memo[ pos ][ lower ][ sum ];
    if( ret != -1 ) return ret;
    ret = 0;
    int d = s[ pos ] - '0';
    if( lower ){ // número formado ya es menor que s
        for( int i = 0; i <= 9; ++i ){
            ret += f( pos + 1, lower, sum + i );
        }
    }
    else{ // número formado todavía no es menor que s
        for( int i = 0; i <= d; ++i ){
            ret += f( pos + 1, (i < d) ? 1 : 0, sum + i );
        }
    }
    return ret;
}
```

Problemas

SPOJ – Sum of digits

DP con Máscara de Bits

- ❑ Se denomina así a los problemas en los que nos piden calcular una función sobre un conjunto de elementos y para ello previamente debemos calcular la función sobre sus subconjuntos.
- ❑ El número de elementos del conjunto generalmente es 20.

DP con Máscara de Bits

Calcular el número de diferentes asignaciones de n ($1 \leq n \leq 20$) cursos a n alumnos de tal forma que cada uno lleve exactamente un curso que le guste.

Entrada

2
0 1
1 1

Salida

1

Explicación : alumno 1 escoge curso 2 (el único que le gustaba) y el alumno 2 escoge curso 1 (a él le gustaban los dos cursos)

DP con Máscara de Bits

Muy parecido a la idea del DP sobre dígitos, ahora las casillas a llenar serán los **n** alumnos y en cada casilla tenemos **n** posibles cursos a asignar (ahora cada curso se usa una vez).

Sea $S = \{cur_1, cur_2, \dots, cur_n\}$

$$f(pos, S) = \sum_{i=1}^n f(pos + 1, S - \{cur_i\})$$

*$O(n * 2^n * n)$... se puede reducir?*

DP con Máscara de Bits

```
11 f( int pos, int mask ){
    if( pos == n ){
        if( mask == ( ( 1 << n ) - 1 ) ) return 1;
        return 0;
    }
    11 &ret = dp[ pos ][ mask ];
    if( ret != -1) return ret;
    ret = 0LL;
    for( int i = 0; i < n; ++i){
        if( ! ( ( mask >> i ) & 1 ) && A[ pos ][ i ] == 1 ) {
            ret += f( pos + 1, mask | ( 1 << i ) );
        }
    }
    return ret;
}
```

Observar la relación entre pos y mask!!

Problemas

SPOJ – Assignments

¡ Good luck and have fun !