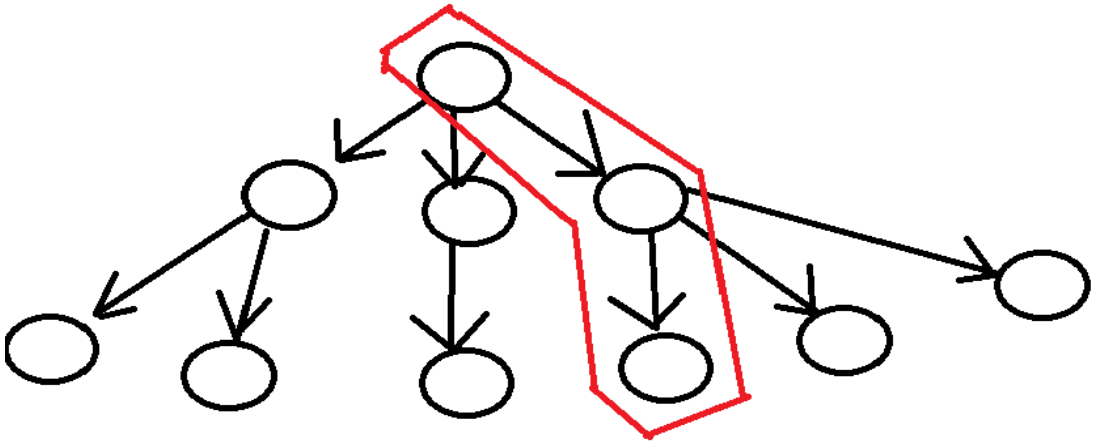
Programación Dinámica

Bach. Rodolfo Mercado Gonzales
Universidad Nacional de Ingeniería

Introducción

En el tema pasado vimos algoritmos greedy. ¿Pero qué sucede cuándo el problema en cuestión no presenta **greedy choice property?**

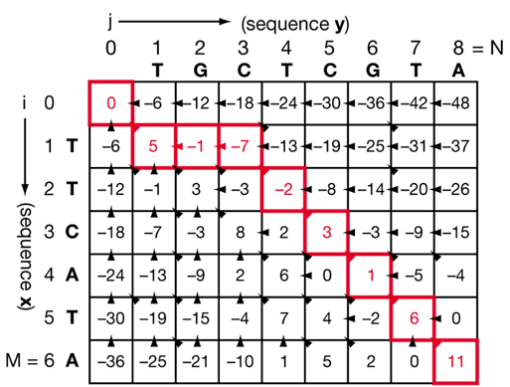


Programación Dinámica

El término fue creado por Richard Bellman. Aquí, **programación** no hace referencia a un crear un código en la computadora, sino a un **método tabular de optimización** , o también a un proceso de **planificación** para hallar la solución óptima. Es decir, de la misma forma que se usa el término en “programación lineal”.

El término **dinámico** hace referencia a que es un proceso multi-etapa.

Dynamic programming matrix:

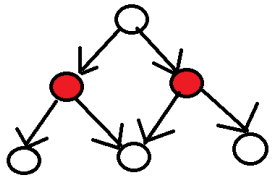


Programación Dinámica

En el sentido clásico, programación dinámica se aplicaba generalmente a problemas de optimización (al igual que greedy). Para que se pudiera aplicar programación dinámica, se necesitaban las siguientes 2 propiedades :

- **Optimal substructure:** Un problema tiene subestructura óptima cuando su solución óptima puede construirse a partir de la **solución óptima** de sus **subproblemas**.

- **Overlapping subproblems:** Decimos que hay problemas que se sobrelapan cuando, al descomponer un problema en subproblemas y así sucesivamente, llegamos a necesitar **resolver un mismo subproblema varias veces.**



Programación Dinámica

Otro punto importante a tomar en cuenta es que los subproblemas en los que se divide un problema deben ser **independientes**.

Esto quiere decir que **no interfieran** entre ellos.

Memorización

Para poder resolver eficientemente estos subproblemas, la idea de programación dinámica es resolverlos solo **una vez** y luego guardar el resultado y volverlo a utilizar cuando se necesite. Esta técnica se llama **memorización.**

Ventajas

- En problemas que estamos haciendo cierto número de consultas, puede ayudarnos a disminuir el número de consultas. Esto es particularmente útil en problemas interactivos cuando tenemos un número limitado de estas.

- En ciertos problemas puede disminuir tanto el número de operaciones que puede ayudarnos a disminuir la complejidad. Este es el caso en programación dinámica.

Memorización

Veamos un ejemplo de **memorización en funciones recursivas**:

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

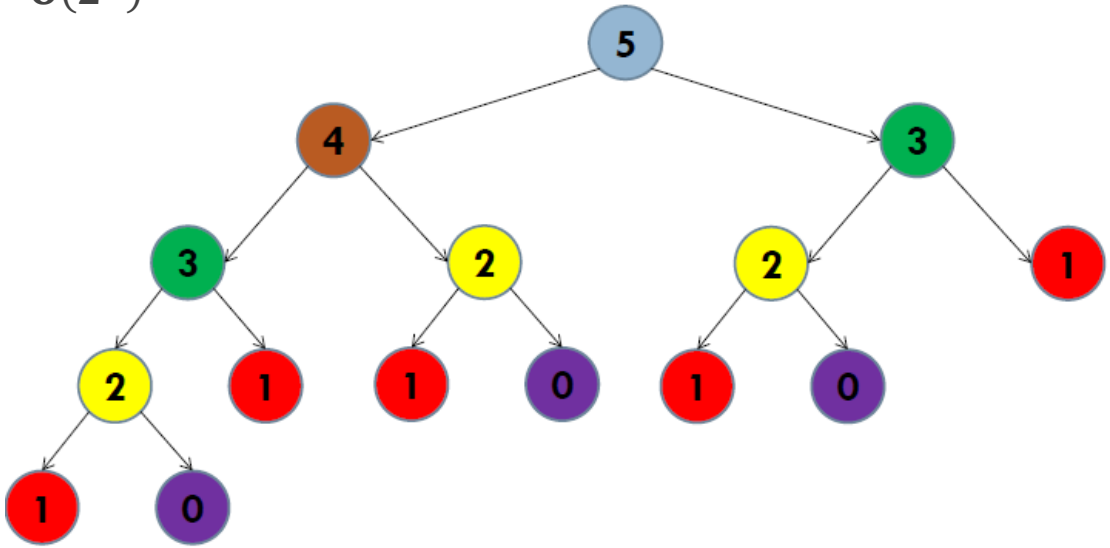
0, 1, 1, 2, 3, 5, 8, 13, 21, …

Además lo hemos implementado recursivamente, entonces tratemos de **hallar el Fibonacci de la posición 5.**

¿ Qué pasa ?

Memorización

Muy lento : $O(2^N)$



Memorización

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

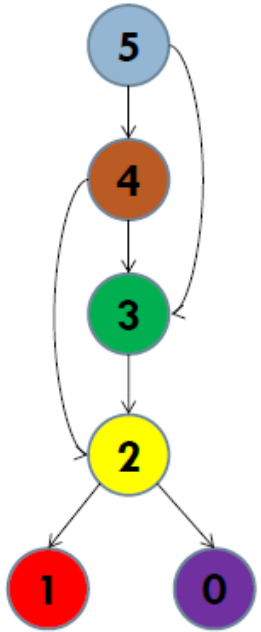
¿Cómo lo mejoramos?

Spoiler : Memorización

Memorización

Logramos reducir el número de llamadas a la función.

$$\boldsymbol{O(N)}$$



Programación Dinámica

Volvamos a analizar los **problemas de optimización** de los que hablábamos inicialmente (fibonacci no es un problema de optimización).

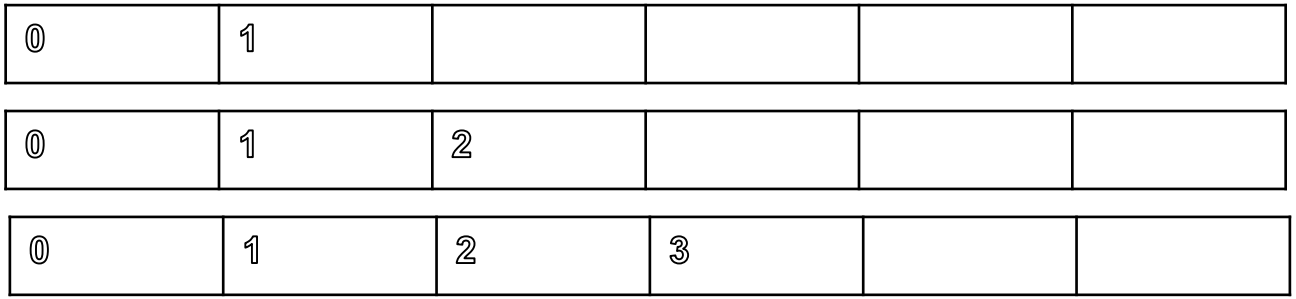
La idea es similar a como hacíamos en la estrategia **greedy**. Solo que en lugar de irnos por el camino que parezca el mejor, trataremos de "probar" todos de forma eficiente. La idea es **dividir** un problema en subproblemas de forma recursiva. **Resolver** recursivamente los subproblemas. **Combinar** todas las soluciones de los subproblemas para hallar la solución óptima (esto es posible gracias a la **subestructura óptima**).

¿Hasta ahí te suena a algún conocido?

Spoiler : Divide and Conquer

Programación Dinámica

Sin embargo, el gran problema de aplicar **Divide and Conquer** aquí, sería que la solución sería exponencial (debido a la repetición de subproblemas). Para evitar esto , lo que haremos es que cada vez que resolvamos un problema, **memorizamos** su solución. Si volvemos a entrar a ese mismo problema posteriormente, **devolvemos lo que hemos memorizado.** Esta memorización se suele hacer en arreglos o en estructuras de STL. Esto es lo que da la idea de **solución tabular.**



Programación Dinámica

Hay que tener en cuenta que existen dos enfoques de implementación de programación dinámica.

- **Top down :** De forma recursiva
- **Bottom up :** De forma iterativa

En esta primera parte veremos la implementación **top down** ya que es la forma más natural de verlo en un primer momento.

Programación Dinámica

Como estaremos utilizando funciones recursivas, hay que tener en cuenta ciertos elementos en nuestras soluciones :

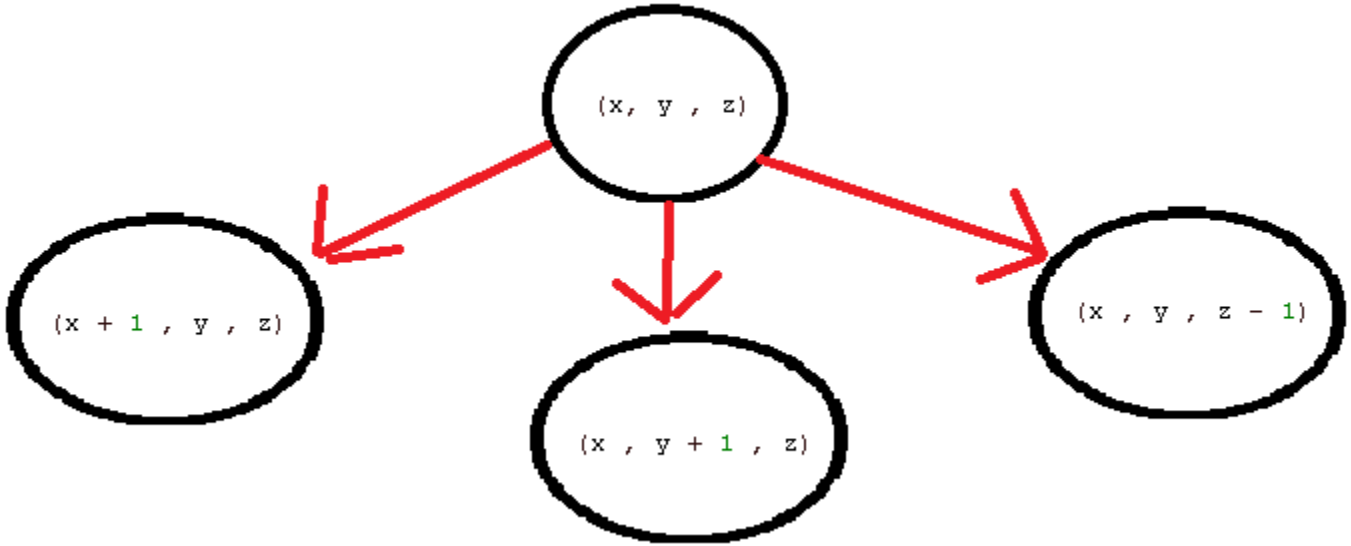
- **Estados** : Son los parámetros o datos que nuestra función tendrá. Hay que tener en cuenta de tener los datos que sean **necesarios y suficientes**.

```
int f(int x , int y, int z){
    ...
}
```

- **Transiciones**: Es la forma en la que pasarás a otros estados

```
int f(int x , int y, int z){
    ...
    return max( f(x + 1 , y , z) , f(x , y + 1 , z) + f(x , y , z - 1) );
}
```

Programación Dinámica



Ejemplo : Knapsack problem

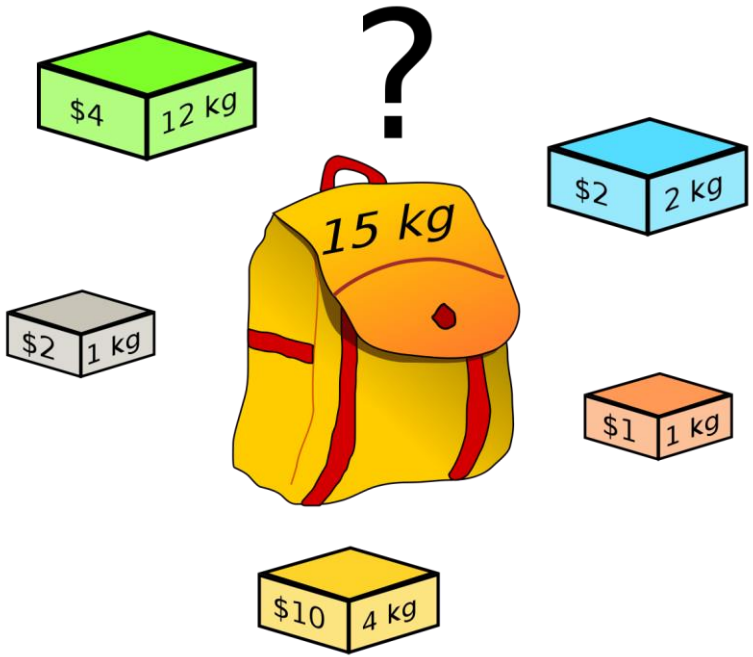
Este es uno de los problemas clásicos de optimización.

Problema : Tenemos **n** objetos. Cada uno de estos tiene un valor monetario asociado $\mathbf{V_i}$ y un peso $\mathbf{W_i}$. Se quiere meter estos objetos en una mochila, pero lamentablemente no todos entran. Mi mochila solo puede soportar un peso total de **W**. Se pide maximizar el valor monetario total de los objetos de tal forma que no excedan el peso de la mochila.

En otras palabras , se pide :

$$\begin{aligned} \text{maximizar} \quad & \sum_{i=1}^{n} \text{Vi}\, x_i \\ \text{tal que} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ \text{y} \quad & x_i \in \{0, 1\}. \end{aligned}$$

Ejemplo : Knapsack problem



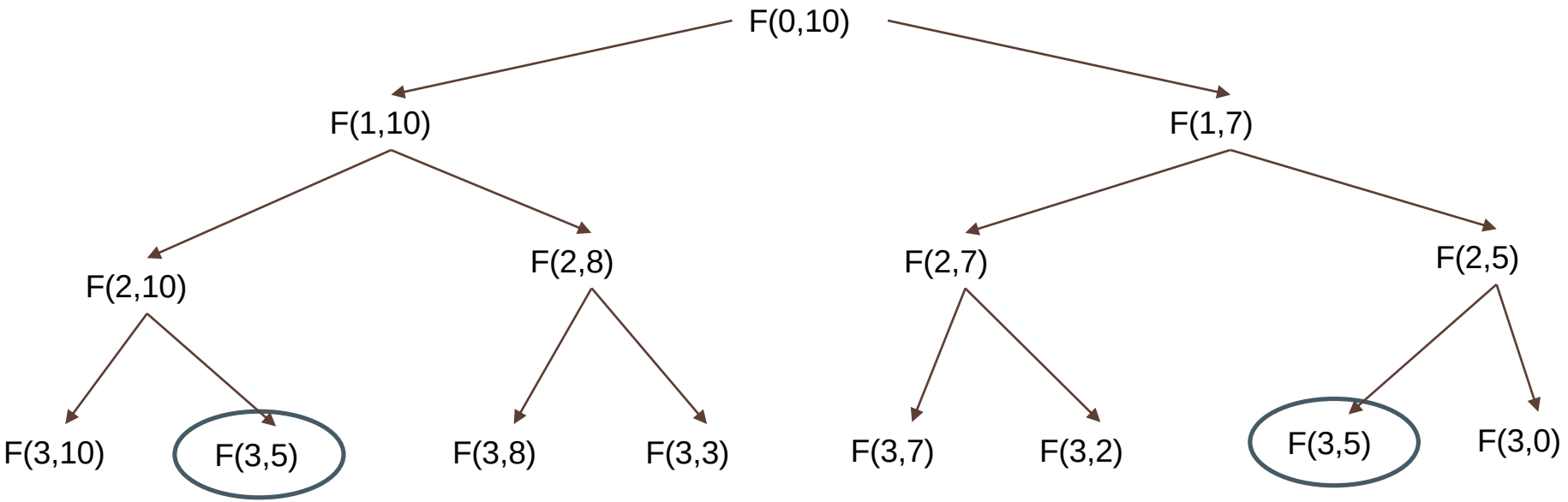
Ejemplo : Knapsack problem

Solución con dp :
Definimos la función f(pos , W) como el máximo valor monetario que puedo conseguir a partir de la posición **pos**, si mi tamaño máximo es **W**.

- Nuestro **estado** está compuesto por un iterado **pos** , y el peso permitido **w**.
- Podemos notar que las **transiciones** dependen de si en una determinada posición pongo el objeto o no lo pongo.

f(pos , W) = max(f(pos + 1 , W) , v[pos] + f(pos + 1 , W – w[pos]))

¿Tiene subestructura óptima? Spoiler : Sí
Sabemos que si no memorizamos , la complejidad será $\mathbf{O}(\mathbf{2^N})$. ¿A cuánto mejora si memorizamos?


F(0,10)
F(1,10)
F(1,7)
F(2,10)
F(2,8)
F(2,7)
F(2,5)
F(3,10)
F(3,5)
F(3,8)
F(3,3)
F(3,7)
F(3,2)
F(3,5)
F(3,0)

N / W	0	1	2	3
0				X
1		Z		Y
2				
3				
4				

Análisis de complejidad

En general, cuando se analice la complejidad en los problemas de dp se puede seguir el siguiente procedimiento :

- **Contar cuántos estados** posibles hay : Generalmente esto sale de las dimensiones de tu arreglo de memorización.

- **Sumarizar** los aportes de cada estado: Para cada estado, analizar cuántas operaciones realiza para hacer las transiciones. Usualmente el número de operaciones es constante para cada estado, pero **no siempre**.

Teniendo eso en cuenta, la complejidad del dp del knapsack 0-1 es O(N x W)

Programación Dinámica

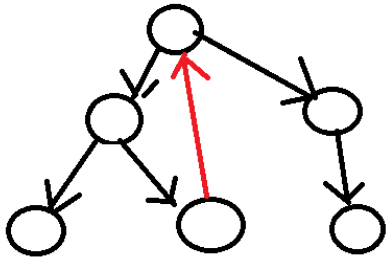
Hemos utilizado esta técnica en problemas de optimización. Sin embargo , en programación competitiva, se suele generalizar dp a problemas que van más allá de los problemas de optimización. Por ejemplo, se utiliza mucho en problemas de **conteo**.

Podemos “generalizar” la propiedad de “subestructura óptima”, a que un problema puede ser resuelto en función de las soluciones de sus subproblemas.

De esta forma , muchas veces se ve a dp como una **recursividad con memorización**. Así, algunos podrían considerar el problema de fibonacci como dp.

Observaciones

- Si el problema tiene subestructura óptima, pero no tiene subproblemas que se sobrelapan, convendría utilizar un divide and conquer.
- Muchas veces un problema que sale con greedy también sale con dp. El greedy suele ser más rápido, pero el dp es más seguro.
- Tener cuidado que **no** haya **bucles** al pasar por los estados. Si eso sucede **no** se podrá aplicar dp al problema (hay otras técnicas que se verán en grafos)



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.

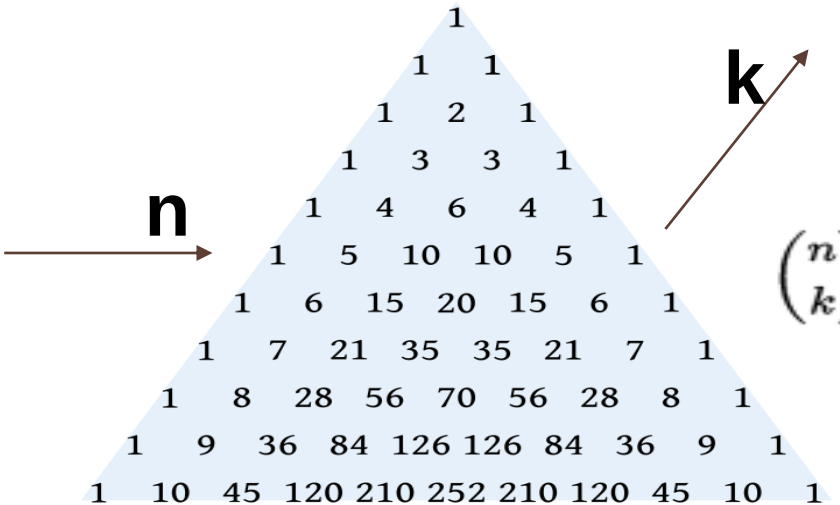
{ 4, 10, 8 , 9 }

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.

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

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 $\boldsymbol{n} - \mathbf{1}$ 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 $\boldsymbol{m} - \mathbf{1}$, 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 );
    }
}
```

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.

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

Problemas

SPOJ – Edit distance

UVA 526 – String Distance and Transform Process

DP Sobre Dígitos

Es un dp en el que se recorren los dígitos. Es decir, uno de los parámetros del estado del dp será la posición del dígito en el que nos encontramos. Generalmente nos sirve para resolver problemas de este tipo:

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

DP Sobre Dígitos

Podemos definir una función f , tal que:

$$\boldsymbol{f(\mathbf{n})} : \text{cantidad de números entre } \mathbf{0} \text{ y } \mathbf{n} \text{ que cumplen cierta propiedad}$$

Por la tanto la solución al problema sería **:** $\boldsymbol{f(B) - f(A-1)}$

DP Sobre Dígitos / Comparando Números

Sea la representación decimal de dos números A y B :

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

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 $\boldsymbol{f(n)}$, que encuentra todos los números menores o iguales a $\boldsymbol{n}$ que cumplen cierta propiedad, utilizando la programación dinámica.

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

1 9 **8** 7 6

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

1 9 **8** 7 6

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

DP Sobre Dígitos

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

Forma general:

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

$\boldsymbol{s}$ **:** cadena que representa al número del cual no nos debemos exceder (puede ser una variable global)

$\boldsymbol{pos}$ **:** entero que representa la posición de cada uno de los dígitos de s

$\boldsymbol{lower}$**:** booleano que indica que el número que estamos formando ya es menor que s .

DP Sobre Dígitos

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

DP Sobre Dígitos

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

Por ejemplo: $\mathrm{A} = 5\,,\ \mathrm{B} = 103$

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 + … + 9 + 1 +0 = 46

DP Sobre Dígitos

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

```
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	Salida
2 0 1 1 1	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 $S = \{cur_1, cur_2, \dots, cur_n\}$

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

$\boldsymbol{O(n * 2^n * n)}$ *... se puede reducir?*

DP con Máscara de Bits

```
ll f( int pos, int mask ){
    if( pos == n ){
        if( mask == ( ( 1 << n ) - 1 ) ) return 1;
        return 0;
    }
    ll &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 !