Programación Dinámica

MSc. (c) Jhosimar George Arias Figueroa

jariasf03@gmail.com

State University of Campinas
Institute of Computing

Contenido

Introducción

- Es un paradigma para resolver problemas.
- Similar en algunos aspectos a las técnicas de divide y conquista y backtracking.
- En divide y conquista tenemos:
  - Dividir el problema en **subproblemas independientes**.
  - Resolver cada problema recursivamente.
  - Combinar las soluciones de los subproblemas.
- En programación dinámica tenemos:
  - Dividir el problema en **subproblemas solapados o sobrepuestos**.
  - Resolver cada problema recursivamente.
  - Combinar las soluciones de los subproblemas.
  - No computar la respuesta para el mismo problema más de una vez.

Introducción

Usar DP donde tenemos problemas con las siguientes propiedades:

- **Subestructuras óptimas**
  - Solución óptima del problema original contiene soluciones óptimas de subproblemas.
  - *Esto es similar al requerimiento de un algoritmo voraz.*
- **Sobreposición de subproblemas**
  - Número de distintos subproblemas son actualmente pequeños.
  - Subproblemas son calculados repetidamente.
  - *Esto es diferente de Divide y Conquista.*

Introducción

- Las habilidades clave que se tienen que desarrollar para ser bueno en DP son las habilidades de determinar los estados del problema y las relacioens o transiciones entre problemas y subproblemas.
- DP es principalmente usado para resolver problemas de optimización y conteo.
- Si encontramos un problema que indica **"minimizar esto"** o **"maximizar esto"** o **"contar el número de formas de hacer esto"**, entonces tendremos una alta probabilidad que sea un problema de DP.

Contenido

Top-Down

- Recursion + tabla de memorización.
- Es un simple cambio en la solución por búsqueda completa, backtracking.
- Podemos asumir que DP **top-down** es un tipo de backtracking recursivo **'inteligente'** o **'veloz'**.
- Tambien llamado **Memoization**.

Bottom-Up

- Preparar una tabla que tenga un tamaño igual al número de diferentes estados del problema.
- Comenzar a llenar la tabla con casos bases.
- Obtener el **orden topológico** en la cual la tabla es llenada.
- Diferentes formas de pensar en comparación con Top-Down.
- Tomar en consideración que ambas variaciones de DP usan una **"tabla"**.

Top-Down o Bottom-Up?

- Top-Down
  - Pros:
    - Transformación natural de una recursión normal.
    - Solamente se calculan subproblemas cuando sea necesario.
  - Cons:
    - Más lento si existen muchos subproblemas, debido a las llamadas recursivas.
    - El tamaño de la tabla puede ser muy grande (MLE).
- Bottom-Up
  - Pros:
    - Más rápido si muchos subproblemas son visitados.
    - Se puede ahorrar espacio en memoria.
  - Cons:
    - Tal vez no muy intuitivo para aquellos acostumbrados a recursión.
    - Si existen M estados, se llena/visitan todos los estados.

Contenido

Fibonacci

Los primeros dos números en la secuencia Fibonacci son 1 y 1. Todos los demas números en la secuencia son definidos como la suma de los dos números anteriores en la secuencia.

- Problema: Encontrar el *n*th número en la secuencia Fibonacci.
- Vamos a resolverlo con programación dinámica.

1. Formulemos el problema en términos de pequeñas versiones del problema (recursivamente).

$$\text{fibonacci}(1) = 1$$
$$\text{fibonacci}(2) = 1$$
$$\text{fibonacci}(n) = \text{fibonacci}(n-2) + \text{fibonacci}(n-1)$$

Secuencia Fibonacci

2. Convirtiendo la fórmula en recursión.

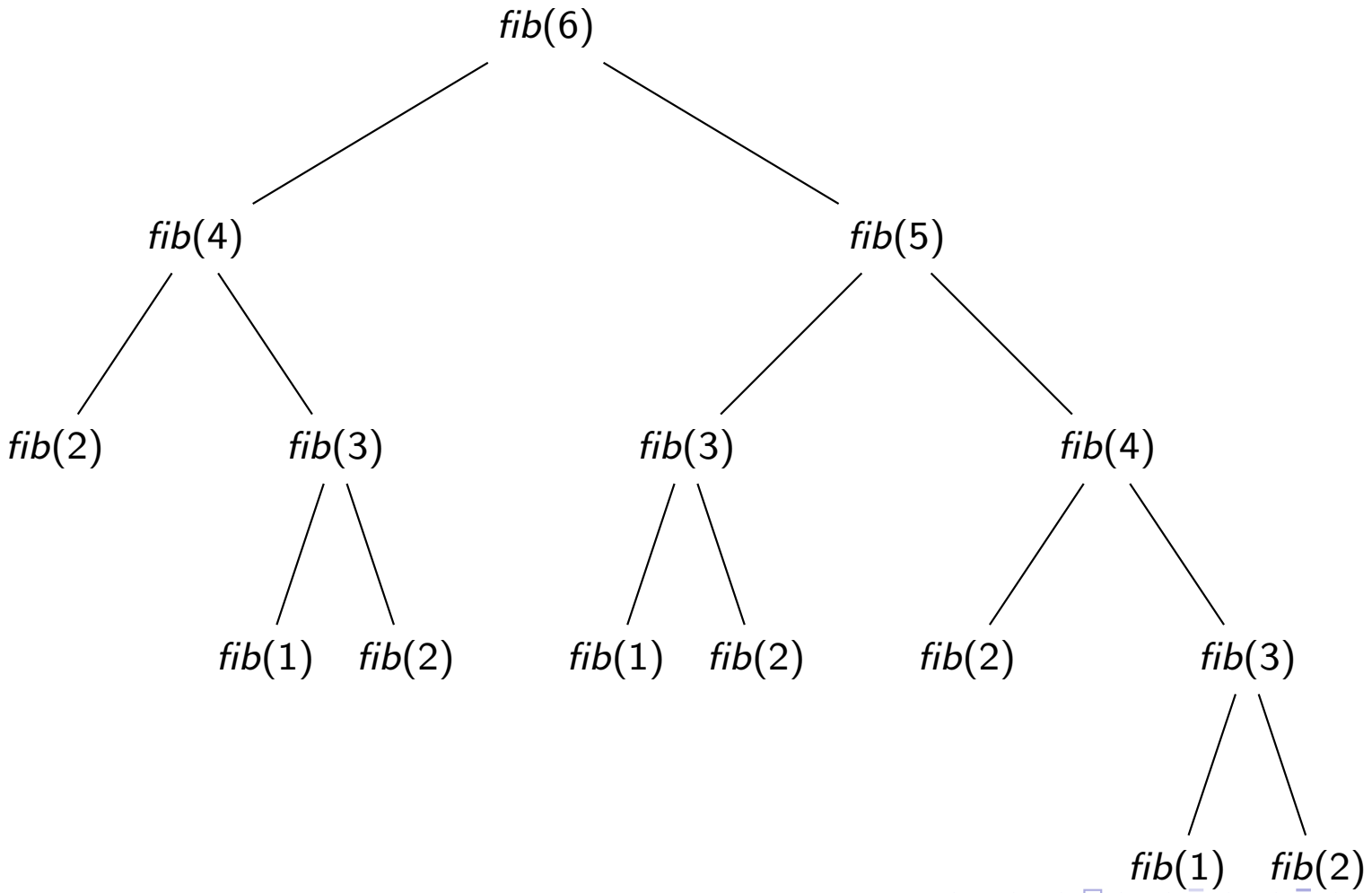
```
int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    return res;
}
```

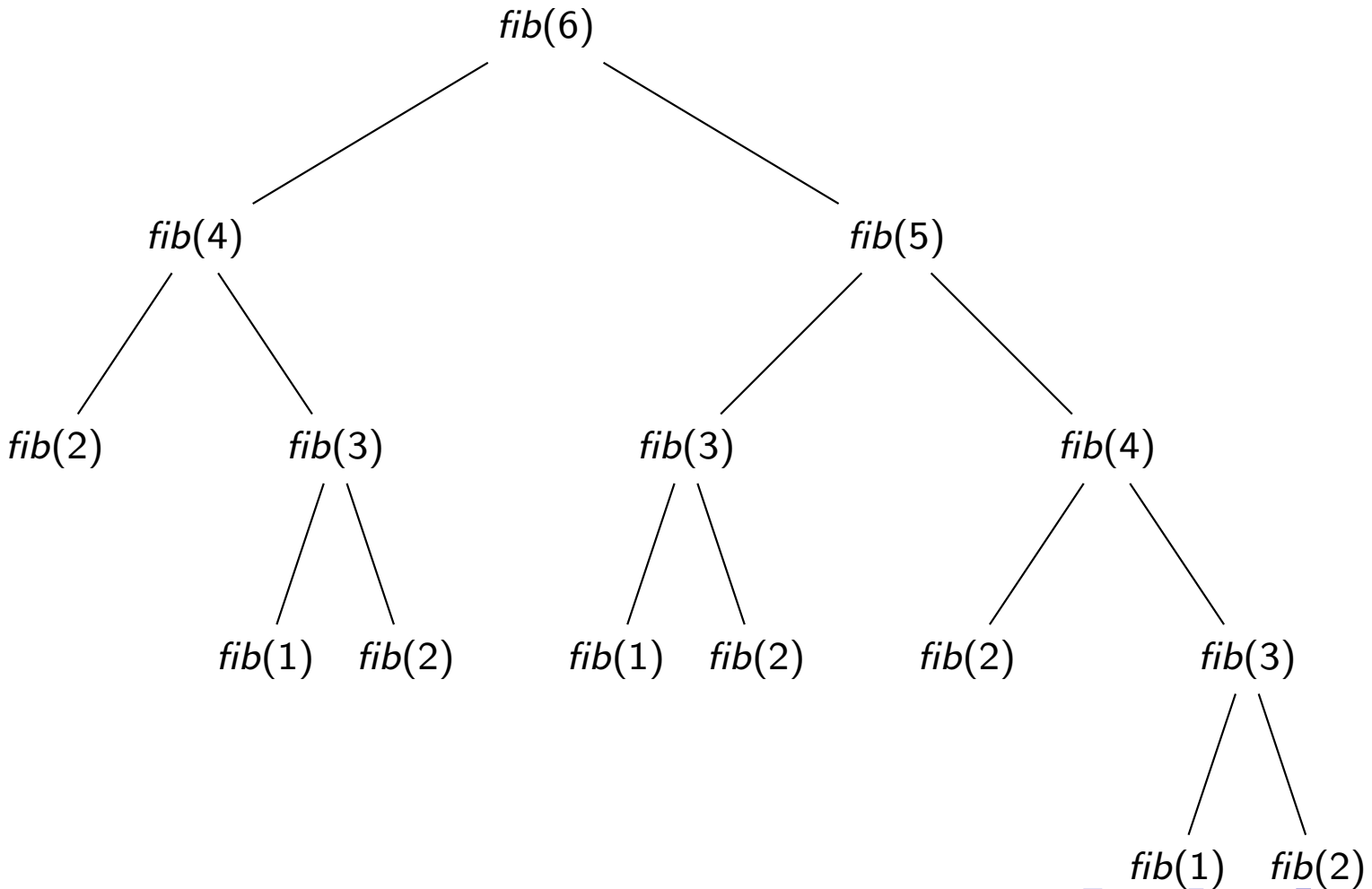
Secuencia Fibonacci

- Cual es su complejidad?



Secuencia Fibonacci

- Cual es su complejidad? Exponencial, aproximadamente $O(2^n)$



Top-Down Approach

```
#define MAX 1005
int memo[MAX];
for (int i = 0; i < MAX; ++i)
    memo[i] = -1;

int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    if (memo[n] != -1) {
        return memo[n];
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    return memo[n] = res;
}
```

Bottom-Up Approach

```
int dp[1000];

int fibonacci(int n) {
    dp[0] = dp[1] = 1;
    for( int i = 2 ; i <= n ; ++i ){
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

Secuencia Fibonacci

- Cual es el tiempo de complejidad ahora?
- Tenemos n posibles entradas en la función: $1, 2, \ldots, n$.
- Cada entrada:
  - será calculada, y el resultado guardado.
  - será retornado desde la memoria.
- Cada entrada será calculada a lo mucho una vez.
- Tiempo de complejidad es $O(n)$.

Contenido

Introducción

Una **subsecuencia** de un string S, es un conjunto de caracteres que aparecen en el orden de izquierda a derecha, pero no necesariamente consecutivamente.

Ejemplo:

ACTTGCG

ACT, ATTC, T, ACTTGC son todos subsecuencias. TTA no es una subsecuencia.

Una subsecuencia común de 2 strings es una subsecuencia que aparece en ambos strings. Una subsecuencia común más larga es una subsecuencia común de máximo tamaño.

Subsecuencia común más larga

- Dados dos strings (o arreglo de enteros) $a[0], \ldots, a[n-1]$ y $b[0], \ldots, b[m-1]$, encontrar la longitud de la subsequencia más larga que tienen en común.

String A	a	c	b	a	e	d
String B	a	b	c	a	d	f

- La longitud de la subsecuencia común más larga de a y b es "acad" de longitud 4.

Subsecuencia común más larga

- Lo que se desea es alinear ambos strings adicionando gaps entre ellos. De tal forma que el número de caracteres iguales sea maximizado. Por ejemplo: Si S = "ACGTCGTGT" y T = "CTAGTGGAG", el alineamiento óptimo seria:

$$\text{S} = \text{A}\mathbf{C}\text{--}\mathbf{GT}\text{CT}\mathbf{G}\text{T}\mathbf{G}\text{T}$$
$$\text{T} = \text{-}\mathbf{C}\text{TA}\mathbf{GT}\text{G-}\mathbf{G}\text{A}\mathbf{G}\text{-}$$

- La longitud de la subsecuencia común más larga de S y T es "CGTGG" de longitud 5.

Contenido

Solución usando recursión

Observando a los primeros caracteres de S y T. Hay 3 cosas que podemos hacer:

- Emparejarlos (escribir uno debajo del otro).
- Insertar un espacio(gap) en S = Remover carácter en T.
- Insertar un espacio(gap) en T = Remover carácter en S.

Podemos observar que una vez hayamos decidido que hacer con los primeros caracteres de los strings, el subproblema restante es nuevamente hallar la subsecuencia común más larga, sobre dos string mas pequeños. Por lo tanto podemos resolverlo recursivamente.

Solución usando recursión

Comencemos comparando los strings de atrás hacia adelante, considerando un carácter a la vez. Tendremos 2 casos:

- Ambos caracteres son iguales:
  - Adicionamos 1 al resultado y removemos el último carácter de ambos strings y realizamos una llamada recursiva a los strings modificados.
- Ambos caracteres son diferentes:
  - Removemos el último carácter del string 1 y realizamos una llamada recursiva.
  - Removemos el último carácter del string 2 y realizamos una llamada recursiva.
  - Retornamos el máximo de ambas llamadas recursivas.

Solución usando recursión

Ejemplo:

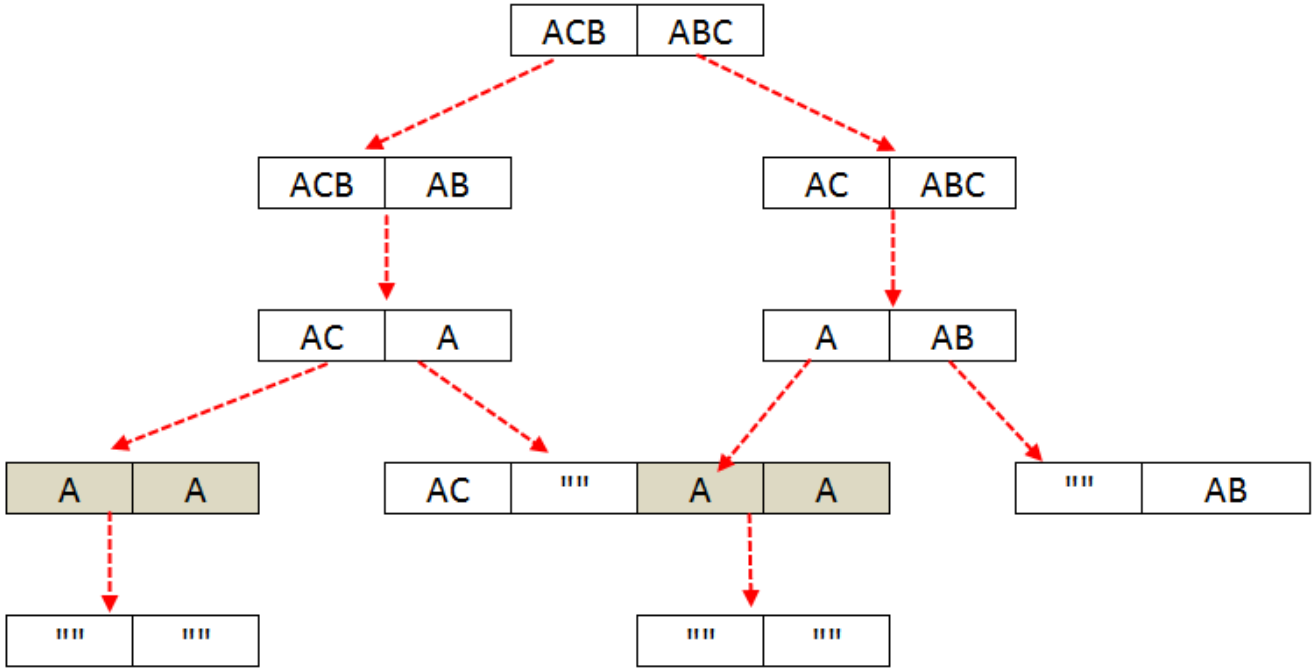
Caso 1

String A: "ABCD", String B: "AEBD"
LCS("ABCD", "AEBD") = 1 + LCS("ABC", "AEB")

Caso 2

String A: "ABCDE", String B: "AEBDF"
LCS("ABCDE", "AEBDF") = Max(LCS("ABCDE", "AEBD"), LCS("ABCD", "AEBDF"))

Solución usando recursión



Para un string de tamaño n, pueden haber 2^n subsecuencias. Si hacemos esto por recursión la complejidad será $O(2^n)$ debido a que resolveremos subproblemas repetidamente.

Recurrencia

- Considerando $\mathrm{lcs}(i, j)$ como el tamaño de la subsecuencia común más larga de los strings $a[0], \ldots, a[i]$ y $b[0], \ldots, b[j]$
- $$\mathrm{lcs}(i,j) = \begin{cases} 0 & \text{if } i = 0 \mid j = 0 \\ 1 + \mathrm{lcs}(i-1, j-1) & \text{if } a[i] = b[j] \\ \max(\mathrm{lcs}(i, j-1), \mathrm{lcs}(i-1, j)) & \text{if } a[i] \; ! = b[j] \end{cases}$$

Solución usando recursión

```
int recursive_lcs(string S, string T, int n, int m){
    if( n == 0 || m == 0 )
        return 0;

    int lcs = 0;
    if( S[n-1] == T[m-1]){ //Caso 1
        lcs = 1 + recursive_lcs( S, T, n-1, m-1 );
    }else{ //Caso 2
        lcs = max( recursive_lcs( S, T, n-1, m ),
                   recursive_lcs( S, T, n, m-1 ) );
    }
    return lcs;
}
```

Contenido

Solución Top-Down

El problema con la solución recursiva es que subproblemas similares son llamados muchas veces. Un subproblema consiste de una llamada a *recursive_lcs*, siendo los argumentos dos prefijos de S y T. Si tenemos 2^n llamadas recursivas, algunos de estos subproblemas deben ser llamados más de una vez.

La solución usando programación dinámica es revisar si ya hemos resuelto un subproblema con anterioridad. Para ello solo basta en adicionar una tabla de memorización, de forma similar como se hizo con el problema de fibonacci.

Solución Top-Down

```
#define MAX 1005
int memo[MAX][MAX];
//O(n*m)
int topdown_lcs(string S, string T, int n, int m){
    if( n == 0 || m == 0 )
        return 0;

    //Subproblema resuelto con anterioridad
    if( memo[n][m] != -1 )
        return memo[n][m];

    int lcs = 0;
    if( S[n-1] == T[m-1]){ //Caso 1
        lcs = 1 + recursive_lcs( S, T, n-1, m-1 );
    }else{ //Caso 2
        lcs = max( topdown_lcs( S, T, n-1, m ),
                   topdown_lcs( S, T, n, m-1 ) );
    }
    return memo[n][m] = lcs; //Memorizar subproblema
}
```

Solución Bottom-Up

- La solución Top-Down es una forma más inteligente de la solución recursiva, ya que al guardar la solución de un subproblema, no necesitamos volver a realizar los calculos de ese subproblema.
- El algoritmo recursivo controla el orden de llenado en la tabla, sin embargo; podriamos obtener los mismos resultados si llenamos la tabla en otro orden.
- Podemos usar una solución iterativa.
- La única cosa de la cual tenemos que preocuparnos es que cuando llenemos un valor en la tabla lcs[i][j], necesitamos tenemos conocimiento de valores de los cuales depende, para nuestro problema son los dos casos: Caso 1 - lcs[i-1][j-1] y Caso 2 - lcs[i-1][j], lcs[i][j-1].

Solución Bottom-Up

```
#define MAX 1005
int lcs[MAX][MAX];
//O(n*m)
int bottom_up_lcs(string S, string T, int n, int m){
    //Inicializacion
    for( int i = 0 ; i <= n ; ++i ) lcs[0][i] = 0;
    for( int i = 0 ; i <= m ; ++i ) lcs[i][0] = 0;

    for( int i = 1; i <= n ; ++i ){
        for( int j = 1 ; j <= m ; ++j ){
            if( S[i - 1] == T[j - 1] ){ //Caso 1
                lcs[i][j] = 1 + lcs[i - 1][j - 1];
            }else{ //Caso 2
                lcs[i][j] = max( lcs[i - 1][j], lcs[i][j - 1]);
            }
        }
    }
    return lcs[n][m];
}
```

Solución Bottom-Up

		A	B	C	D	A
	0	0	0	0	0	0
A	0	1	1	1	1	1
C	0	1	1	2	2	2
B	0	1	2	2	2	2
D	0	1	2	2	3	3
E	0	1	2	2	3	3
A	0	1	2	2	3	4

LCS - "ACDA"

Solución Bottom-Up

- Las ventajas de este método incluye el hecho de que cada iteración es usualmente mas rápido que la recursión.
- No necesitamos inicializar todos los valores de la tabla a -1.
- Una desventaja con respecto a **Memoization** es que este método llena la tabla entera, incluso cuando tal vez sea posible resolver el problema solo considerando algunas celdas de la tabla.

Contenido

Introducción

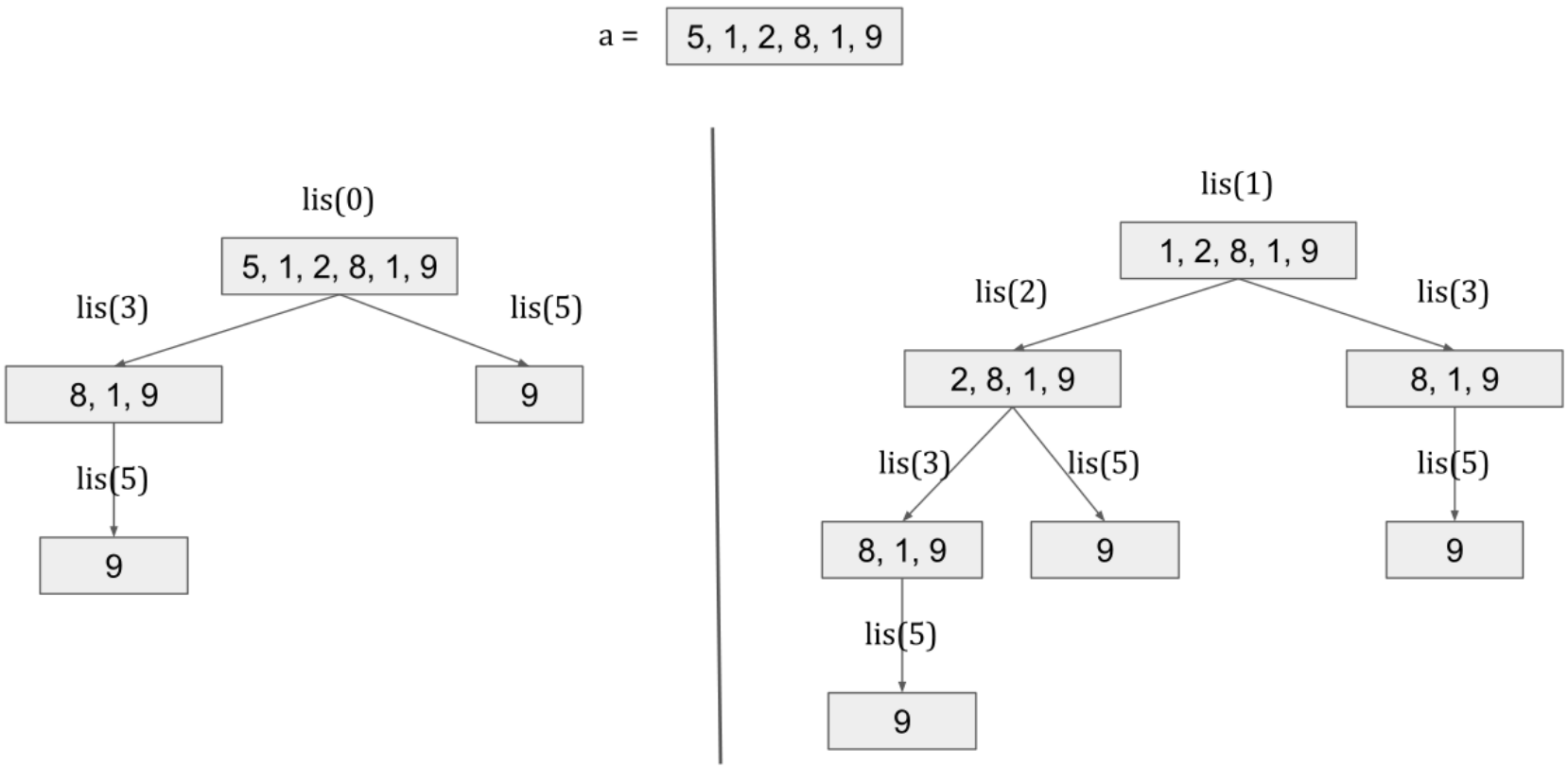
- Dado un arreglo $a[0]$, $a[1]$, ..., $a[n-1]$ de enteros, cual es el tamaño de la subsecuencia creciente más larga?
- Una subsecuencia creciente de a es una subsecuencia de a, tal que sus elementos están en (estrictamente) orden creciente.
- Formalmente, calcular una subsecuencia $b[0]$, $b[1]$, ..., $b[m-1]$ con $b[i] < b[i+1]$ donde i = [0,1,...,m-1] tal que m es máximo.
- $[5, 8, 9]$ y $[1, 8, 9]$ son las subsecuencias crecientes más largas de $a = [5, 1, 8, 1, 9, 2]$
- Existen 2^n subsecuencias, podemos probar con todas ellas.
- Esto resultaria en una complejidad de $O(n2^n)$, lo cual solo puede ser útil si $n \leq 20$.

Recurrencia

- Sea $\mathrm{lis}(i)$ el tamaño de la subsecuencia creciente más larga del arreglo $a[0], \ldots, a[i]$, *que termina en* i
- $\mathrm{lis}(i) = 1 + \max(\mathrm{lis}(j))$ donde $j < i$ *y* $a[j] < a[i]$, si no hay tal j entonces $\mathrm{lis}(i) = 1$

Contenido

Solución usando Recursión



Solución usando Recursión

```
//O(2^n)
int recursive_lis( vector<int> a, int n, int pos){
    int lis = 1;
    for( int i = pos; i < n ; ++i){
        if(a[i] > a[pos]){
            lis = max( 1 + recursive_lis(a, n, i ), lis );
        }
    }
    return lis;
}

//O(n*2^n)
int solve_recursive_lis(vector<int> a, int n){
    int lis = 0;
    for( int i = 0 ; i < n ; ++i ){
        lis = max(lis, recursive_lis(a, n, i));
    }
    return lis;
}
```

Contenido

Solución Top-Down

```
#define MAX 1005
int memo[ MAX ];
//O(n)
int topdown_lis( vector<int> a, int n, int pos){
    if( memo[pos] != -1 ) //Subproblema resuelto con anterioridad
        return memo[pos];

    int lis = 1;
    for( int i = pos; i < n ; ++i){
        if(a[i] > a[pos]){
            lis = max( 1 + topdown_lis(a, n, i ), lis );
        }
    }
    return memo[pos] = lis;
}
//O(n^2)
int solve_topdown_lis(vector<int> a, int n){
    memset( memo, -1, sizeof(memo) );
    int lis = 0;
    for( int i = 0 ; i < n ; ++i ){
        lis = max(lis, topdown_lis(a, n, i));
    }
    return lis;
}
```

Solución Bottom-Up

```
#define MAX 1005
int dp[ MAX ];

int bottom_up_lis(vector<int> a, int n){
    for( int i = 0 ; i < n ; ++i ) dp[ i ] = 1;

    for( int i = 0 ; i < n ; ++i ){
        for( int j = i + 1  ; j < n ; ++j ){
            if( a[ i ] < a[ j ] && dp[ j ] < dp[ i ] + 1 ){
                dp[ j ] = dp[ i ] + 1;
            }
        }
    }

    int ans = 1;
    for( int i = 0 ; i < n ; ++i )
        ans = max( ans, dp[i] );
    return ans;
}
```

Solución Bottom-Up

Considerando el siguiente arreglo:

$$A = 1, 12, 7, 0, 23, 11, 52, 31, 61, 69, 70, 2$$

Dos posibles resultados serian:

Array	1	12	7	0	23	11	52	31	61	69	70	2
LIS	1	2	2	1	3	3	4	4	5	6	7	2

Array	1	12	7	0	23	11	52	31	61	69	70	2
LIS	1	2	2	1	3	3	4	4	5	6	7	2

Complejidad

- Se tienen n posibles números en el arreglo.
- Cada entrada es calculada en $O(n)$, asumiendo que las llamadas recursivas son $O(1)$.
- La complejidad total es $O(n^2)$
- Esta solución será lo suficientemente rápida para $n \leq 10\ 000$, mucho mejor que fuerza bruta.
- Sin embargo, para tamaños mayores la solución aún es ineficiente.
- Existe un algoritmo voraz que resuelve el problema en $O(nlogn)$.