

# PROGRAMACIÓN DINÁMICA

Problema motivación, definición formal, LIS, y RSQ.

Facundo Galán

# PROBLEMA MOTIVACIÓN

Calcular el N-ésimo número de la sucesión de Fibonacci.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Ejemplo:

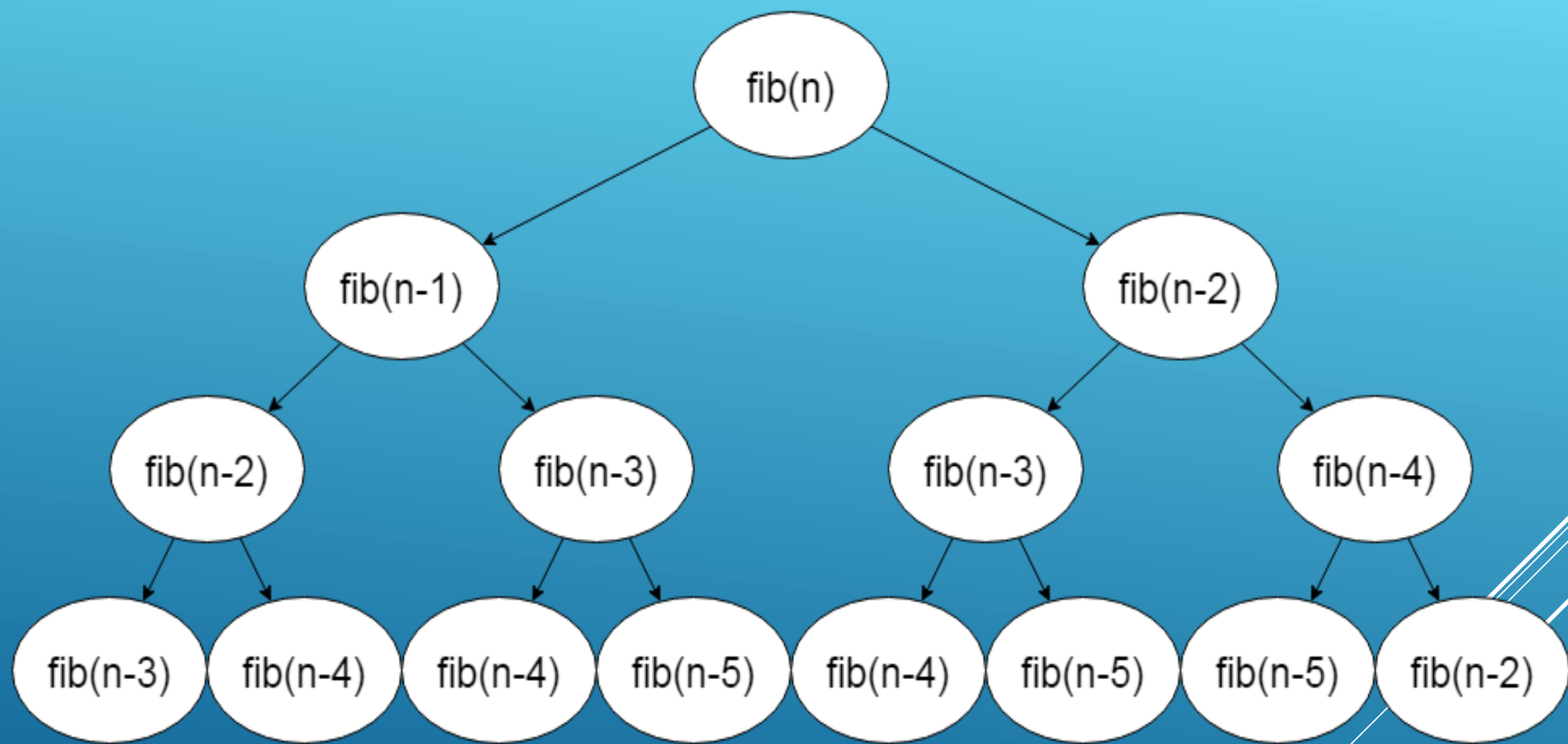
- $F(0) = 1;$
- $F(1) = 1;$
- $F(2) = 2;$
- $F(10) = 8;$

# PROBLEMA MOTIVACIÓN (CONT.)

Solución posible:

```
int fib (int idx) {  
    if (idx == 0 || idx == 1) return 1;  
    else return fib(idx-1) + fib(idx-2);  
}
```

¿Qué complejidad de tiempo tiene este algoritmo?



# PROBLEMA MOTIVACIÓN (CONT.)

Posible mejora:

```
int FIB[MAXN]; // Inicializada en -1;
int fib (int n) {
    if (n== 0 || n == 1) return 1;
    else if (FIB[n] != -1) return FIB[n];
    else return FIB[n] = fib(n-1) + fib(n-2);
}
```

¿Y ahora qué complejidad tiene?

# PROGRAMACIÓN DINÁMICA: DEFINICIÓN

Es una estrategia de resolución de problemas mediante la descomposición en subproblemas más chicos y más simples.

Se puede aplicar a problemas que muestren tener dos propiedades fundamentales:

- Overlapping subproblems.
- Optimal substructure.

El objetivo es optimizar el cómputo y reducir el tiempo de ejecución de los algoritmos. Normalmente se aplica a algoritmos de optimización.

# OVERLAPPING SUBPROBLEMS

Se dice que un problema tiene solapamiento de subproblemas, si el problema puede ser dividido en subproblemas que son usados muchas veces, o si un algoritmo recursivo para el problema resuelve el mismo subproblema una y otra vez en lugar de generar nuevos subproblemas.

En el ejemplo de Fibonacci, usamos sólo unas pocas funciones, pero repetidas veces.

# OPTIMAL SUBSTRUCTURE

Un problema tiene subestructura óptima, si una solución óptima puede ser construida eficientemente a partir de las soluciones óptimas de sus subproblemas.

En el ejemplo de Fibonacci la definición misma da como resultado la subestructura óptima, pero en otros problemas puede no ser tan obvia.

Several white lines of varying lengths and slopes are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.



# IMPLEMENTACIONES

Existen dos formas de resolver un problema mediante programación dinámica:

- Top-Down approach: surge directamente de la solución recursiva. Utilizando memoization no se recalculan los llamados recursivos a los mismos subproblemas.
- Bottom-Up approach: una vez que se planteó la solución recursiva en términos de los subproblemas, se intenta reformular el problema de forma tal de empezar por subproblemas chicos para ir construyendo soluciones a subproblemas mayores hasta llegar a la solución del problema. Esto da soluciones del tipo iterativas que usan tablas de valores intermedios.

# TOP DOWN VS. BOTTOM UP

Ventajas de Top-Down sobre Bottom-Up:

- Es más simple de plantear la solución.
- Es más fácil de implementar (map en C++, HashMap en Java)

Ventajas de Bottom-Up sobre Top-Down:

- No es recursiva -> más eficiente.
- Puede ocupar menos memoria. Descartar los resultados de subproblemas más chicos una vez utilizados.
- No requiere estructuras de datos complejas.

# LONGEST INCREASING SUBSEQUENCE (LIS)

The background is a blue gradient, darker at the bottom. Several thin, white, parallel diagonal lines run from the bottom right towards the top right, creating a sense of movement or a stylized 'L' shape.

# LIS

Definición del problema: dada una secuencia de números, se desea encontrar una subsecuencia de la misma, tal que sus elementos mantengan su orden original, y sea estrictamente creciente.

Ejemplo:

Secuencia inicial: {1, 6, 3, 4, 4, 7, 5, 6}

LIS => {1, 3, 4, 5, 6}




# OPTIMAL SUBSTRUCTURE

Primero, debemos hallar una solución recursiva.

Podemos pensar a  $lis(i)$ , como la longitud de la subsecuencia creciente mas larga, que termina e incluye a la posición  $i$ .

Luego, podemos definir que  $lis(i)$ , es el mayor de los  $lis$  de los elementos anteriores a  $i$ , que su valor en la secuencia, es menor al de  $i$ .

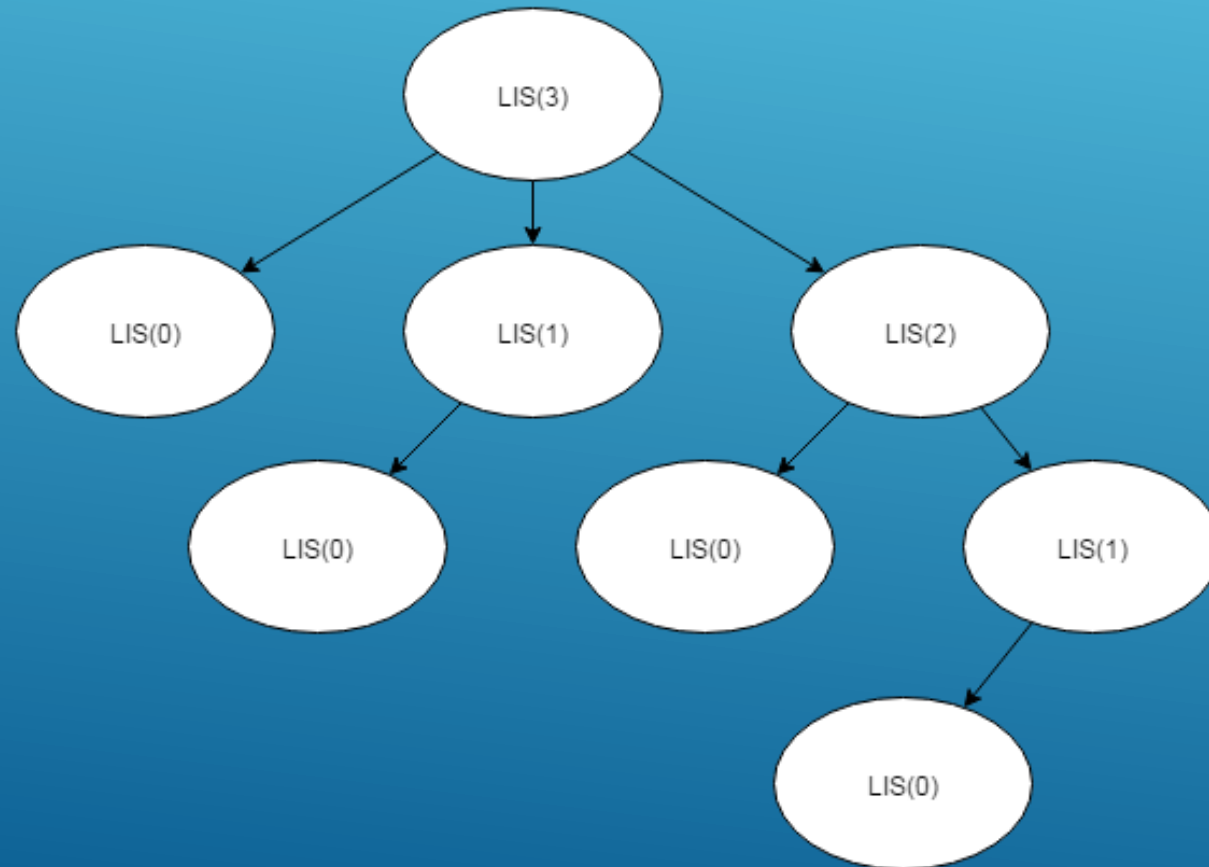
Several white lines of varying lengths and slopes are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.

# OPTIMAL SUBSTRUCTURE (CONT.)

```
vector<int> seq;  
int lis (int idx) {  
    int ans = 1; //Mínimo LIS = 1  
    for (int i=0; i<idx; i++) {  
        if ((seq[i] < seq[idx]) && ans < lis(i) + 1) {  
            ans = lis(i) + 1;  
        }  
    }  
    return ans;  
}
```

# OVERLAPPING SUBPROBLEMS

Luego, tenemos que ver si tiene subproblemas que se repiten.





# PLANTEO DE LA SOLUCIÓN

¿Cómo se puede plantear una solución con DP?

Notemos que para calcular  $LIS(0)$  no necesitamos nada, y que si tenemos  $LIS(0)$  podemos calcular  $LIS(1)$ . Al tener  $LIS(0)$  y  $LIS(1)$ , se puede calcular  $LIS(2)$ ... etc!

Podemos calcular progresivamente los valores de los  $LIS(x)$ , comenzando por  $LIS(0)$ .

# PLANTEO DE LA SOLUCIÓN (CONT.)

¿Qué estructuras necesitaremos?

- $DP[0 \dots N-1]$ : se almacenarán los LIS(x) resueltos. A través de este arreglo implementaremos la memoization.
- $prev[0 \dots N-1]$ : se almacenará el anterior elemento de la subsecuencia, para cada elemento, de forma tal de poder reconstruirla.

¿Cómo plantearían el código para resolver el problema?

# SOLUCIÓN TOP-DOWN

```
int DP[MAXN], prev[MAXN];  
vector<int> seq;  
int lis (int idx) {  
    if (DP[idx] != -1) return DP[idx];  
    DP[idx] = 1; prev[i] = -1; //Mínimo LIS = 1  
    for (int i=0; i<idx; i++) {  
        if ((seq[i] < seq[idx]) && DP[idx] < lis(i) + 1) {  
            DP[idx] = lis(i) + 1; prev[i] = j;  
        }  
    }  
    return DP[idx];  
}
```

// Importante: esta función tiene que ser invocada desde todas las posiciones,  
// y mientras se hace eso, ir guardando el LIS de la secuencia completa.

# SOLUCIÓN BOTTOM-UP

```
int DP[MAXN], prev[MAXN];
vector<int> seq;
int lis () {
    int ans = 0;
    for (int i=0; i<seq.size(); i++) {
        DP[i] = 1; prev[i] = -1; // Mínimo LIS = 1
        for (int j=0; j<i; j++) {
            if ((seq[j] < seq[i]) && DP[i] < DP[j] + 1) {
                DP[i] = DP[j] + 1; prev[i] = j;
            }
        }
        if (DP[ans] < DP[i]) ans = i;
    }
    return ans; // OJO: lo que se retorna es el índice.
}
```

# RANGE SUM QUERY (RSQ)



# RSQ

Dado un arreglo  $A[]$  con  $N$  números, y una cantidad de consultas  $M$ , imprimir la respuesta a cada una de ellas.

Las consultas están compuestas por un par de números  $L$  y  $R$ , donde  $0 \leq L \leq R < N$ , y lo que se pide es la suma de los elementos de  $A$  en el rango  $[L, R]$ .

Ejemplo:

$$A[7] = \{1, 4, 2, 7, 5, 3, 6\}$$

$$M = 1$$

$$L = 2, R = 6$$

$$RSQ(L, R) = 23$$

## ¿DE QUÉ FORMA APLICA DP ACÁ?

Notemos que la solución para la suma de elementos desde la posición  $L$  hasta la posición  $R$ , es igual a la suma de los elementos desde la posición  $0$  hasta la posición  $R$ , restándole los elementos de la posición  $0$  hasta la posición  $L-1$ .

Ejemplo:

$$A[7] = \{1, 4, 2, 7, 5, 3, 6\}$$

$$M = 1$$

$$L = 2, R = 6$$

$$RSQ(L, R) = RSQ(0, R) - RSQ(0, L-1) = 28 - 5 = 23$$

# PLANTEO DE LA SOLUCIÓN

Si pre-calculamos un arreglo con la suma de todos los elementos desde 0 hasta sí mismos, podemos calcular luego en orden constante la respuesta a cualquier consulta de rangos.

Ejemplo:

$$A[7] = \{1, 4, 2, 7, 5, 3, 6\}$$

$$DP[7] = \{1, 5, 7, 14, 19, 22, 28\}$$

$$M = 1$$

$$L = 2, R = 6$$

$$RSQ(L, R) = DP[R] - DP[L-1] = 28 - 5 = 23;$$

Tener cuidado con el caso  $L = 0$ , en cuyo caso la respuesta es  $DP[R]$ .



## PLANTEO DE LA SOLUCIÓN (CONT.)

Otra alternativa es indexar los elementos del arreglo de 1 a N, y dejar la posición 0 con valor 0.

Ejemplo:

$$A[7] = \{0, 1, 4, 2, 7, 5, 3, 6\}$$

$$DP[7] = \{0, 1, 5, 7, 14, 19, 22, 28\}$$

$$M = 1$$

$$L = 2, R = 6$$

Incrementamos los índices de la consulta en 1:

$$L = 3, R = 7$$

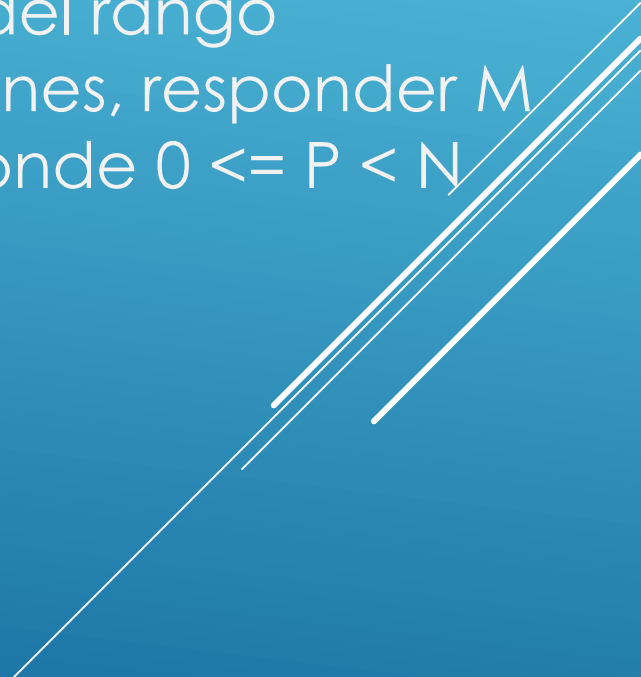
$$RSQ(L, R) = DP[R] - DP[L-1] = 28 - 5 = 23;$$

# SOLUCIÓN CON MEMOIZATION

```
int A[MAXN], DP[MAXN];  
int build_rsq (int n) { // Una vez cargado el arreglo A[]  
    DP[0] = 0;  
    for (int i=1; i<n; i++) DP[i] = DP[i-1] + A[i-1];  
} // A[i-1] para dejar la posición 0 vacía  
int rsq (int l, int r) {  
    return DP[r] - DP[l-1];  
}
```

# PROBLEMA ALTERNATIVO

Se tiene un arreglo  $A[]$  de tamaño  $N$ , inicialmente con todos sus datos en 0. Se deben realizar  $K$  actualizaciones de rangos entre  $U$  y  $V$ , siendo  $0 \leq U \leq V < N$ , sumándole un valor  $W$  a cada posición del rango especificado. Luego de realizadas todas las actualizaciones, responder  $M$  consultas sobre el valor del elemento en la posición  $P$ , donde  $0 \leq P < N$ .


The bottom right corner of the slide features several thin, white, parallel diagonal lines that extend from the bottom edge towards the right edge, creating a modern, abstract graphic element.



# PLANTEO DE LA SOLUCIÓN

Podemos actualizar de forma “lazy” los rangos de las consultas, sumando sólo en el primer elemento y restando en el siguiente al segundo, y al final de las actualizaciones, sumando a cada posición el anterior.

Por favor, no salir corriendo, esto es simple y se explica mejor en la siguiente diapositiva.

Several white lines of varying lengths and slopes are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.



