

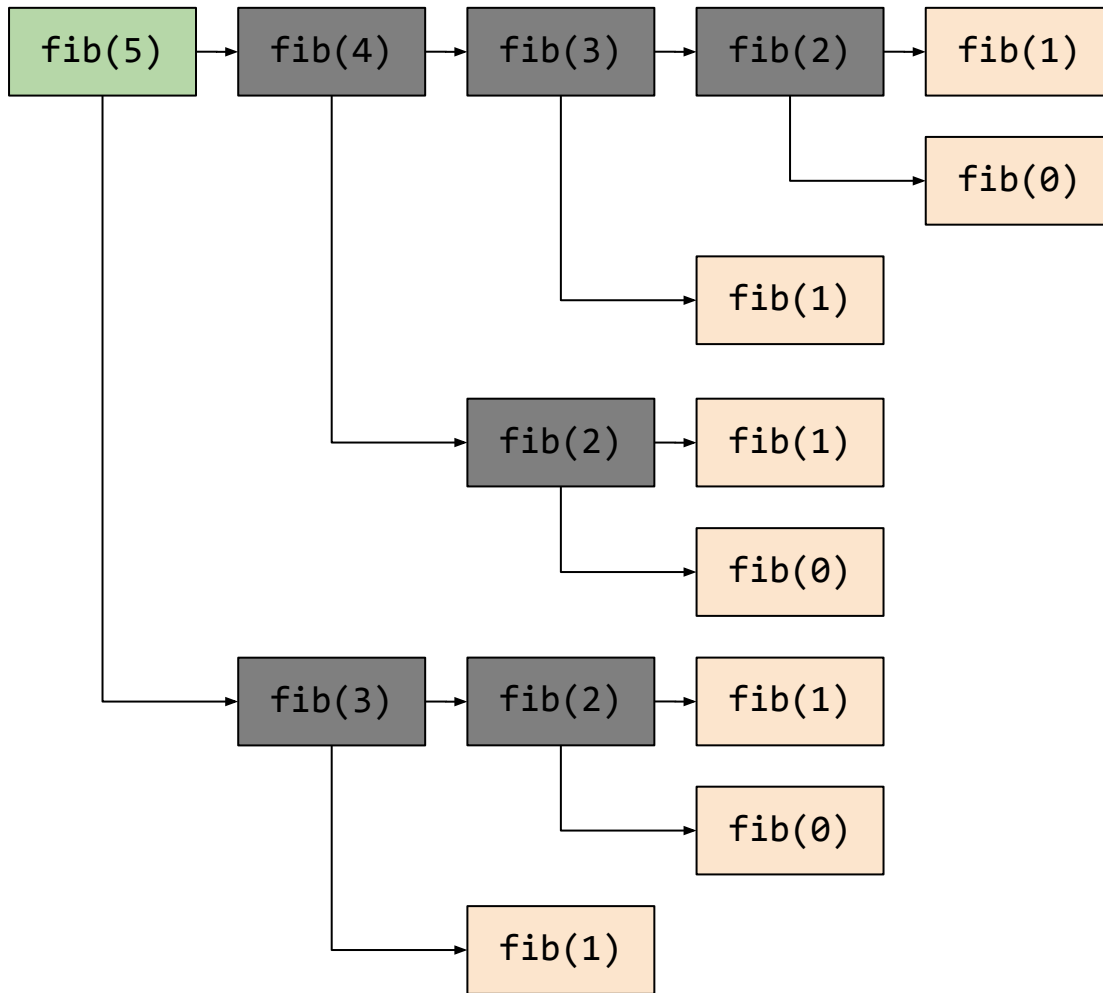
Ejemplo Sencillo: Fibonacci

El cálculo de la **Serie de Fibonacci**, donde cada número es la suma de los dos números anteriores (1, 1, 2, 3, 5, 8, 13, ...).

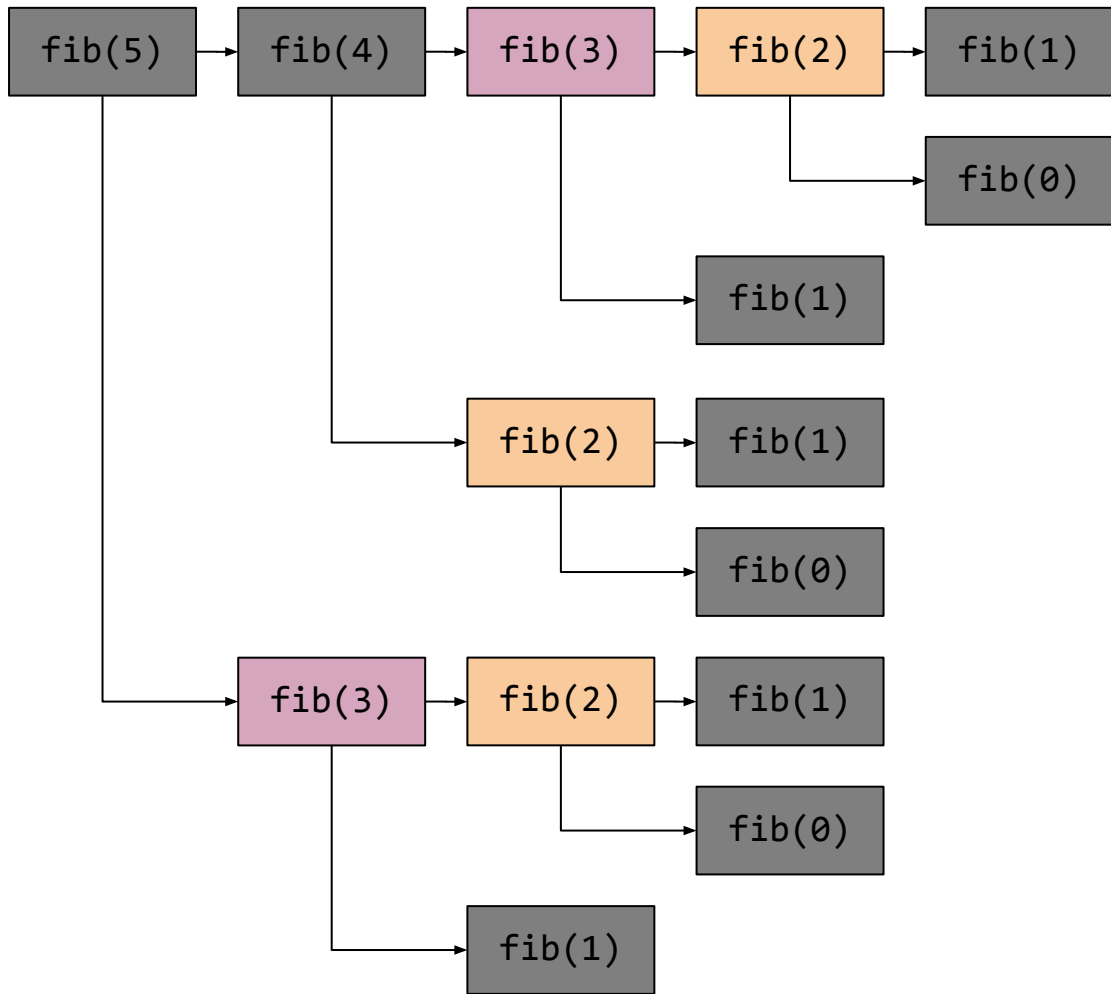
La solución se puede obtener mediante la definición recursiva, pero esto lleva a un alto costo computacional, $O(2^n)$

$$fib(n) = \begin{cases} n & \text{si } n < 2 \\ fib(n-1) + fib(n-2) & \text{en otro caso} \end{cases}$$

Ejemplo Sencillo: Fibonacci



Ejemplo Sencillo: Fibonacci (oportunidad)



Ejemplo Sencillo: Fibonacci (oportunidad)



Podemos aprovechar resultados previos para acelerar el cálculo final.

¿Cómo lo haríamos?

La programación dinámica utiliza una solución incremental, almacenando los resultados anteriores en una tabla, o variables individuales, y utilizando esta información para calcular los siguientes valores de forma eficiente

Técnicas

Existen principalmente dos aproximaciones para resolver problemas con Programación Dinámica:

1. Top-Down/Memorización
2. Bottom-Up/Tabulación

Top-Down

Se conserva una **resolución recursiva**, pero se agrega la **memorización** de resultados previamente calculados para evitar recalcularlos.

Esta aproximación es similar al concepto de *caching*, utilizando habitualmente un **Diccionario** para ello.

¿Qué es *caching*?

Top-Down: Pasos

1. Agregar un parámetro, variable o atributo con la **memoria**, que se utilizará a lo largo de las invocaciones
2. Determinar la **key** para cada subproblema
3. Agregar un caso base donde **chequeamos si** la solución actual no fue **anteriormente calculada**
4. Para cada caso recursivo, **antes de devolverlo, debemos guardarlo** en la memoria

Top-Down: Ejemplo

```
public int fibonacci(int n) {  
    if (n < 2) {  
        return n;  
    }  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```


Top-Down: Ejemplo

1. Agregar la memoria

```
public int fibonacci(int n) {  
    return fibonacci(n, new HashMap<Integer, Integer>());  
}  
  
private int fibonacci(int n, Map<Integer, Integer> mem) {  
    if (n < 2) {  
        return n;  
    }  
    return fibonacci(n-1, mem) + fibonacci(n-2, mem);  
}
```

Top-Down: Ejemplo

2. Definir la key

```
public int fibonacci(int n) {  
    return fibonacci(n, new HashMap<Integer, Integer>());  
}  
  
private int fibonacci(int n, Map<Integer, Integer> mem) {  
    int key = n;  
    if (n < 2) {  
        return n;  
    }  
    return fibonacci(n-1, mem) + fibonacci(n-2, mem);  
}
```

Top-Down: Ejemplo

3. Verificar si
este
subproblema
ha sido
resuelto
previamente

```
public int fibonacci(int n) {  
    return fibonacci(n, new HashMap<Integer, Integer>());  
}  
  
private int fibonacci(int n, Map<Integer, Integer> mem) {  
    int key = n;  
    if (mem.containsKey(key)) {  
        return mem.get(key);  
    }  
    if (n < 2) {  
        return n;  
    }  
    return fibonacci(n-1, mem) + fibonacci(n-2, mem);  
}
```

Top-Down: Ejemplo

4. Si debimos
calcular,
guardar en la
memoria antes
de retornar

```
public int fibonacci(int n) {  
    return fibonacci(n, new HashMap<Integer, Integer>());  
}  
  
private int fibonacci(int n, Map<Integer, Integer> mem) {  
    int key = n;  
    if (mem.containsKey(key)) {  
        return mem.get(key);  
    }  
    if (n < 2) {  
        mem.put(key, n);  
        return mem.get(key);  
    }  
    mem.put(key, fibonacci(n-1, mem) + fibonacci(n-2, mem));  
    return mem.get(key);  
}
```

Top-Down: Ejemplo

Versión final

```
public int fibonacci(int n) {  
    return fibonacci(n, new HashMap<Integer, Integer>());  
}  
  
private int fibonacci(int n, Map<Integer, Integer> mem) {  
    int key = n;  
    if (mem.containsKey(key)) {  
        return mem.get(key);  
    }  
    if (n < 2) {  
        mem.put(key, n);  
    } else {  
        mem.put(key, fibonacci(n-1, mem) + fibonacci(n-2, mem));  
    }  
    return mem.get(key);  
}
```

Bottom-Up

Se resuelven los problemas más pequeños (ya conocidos), y se **componen soluciones más grandes** basándose en las anteriormente calculadas.

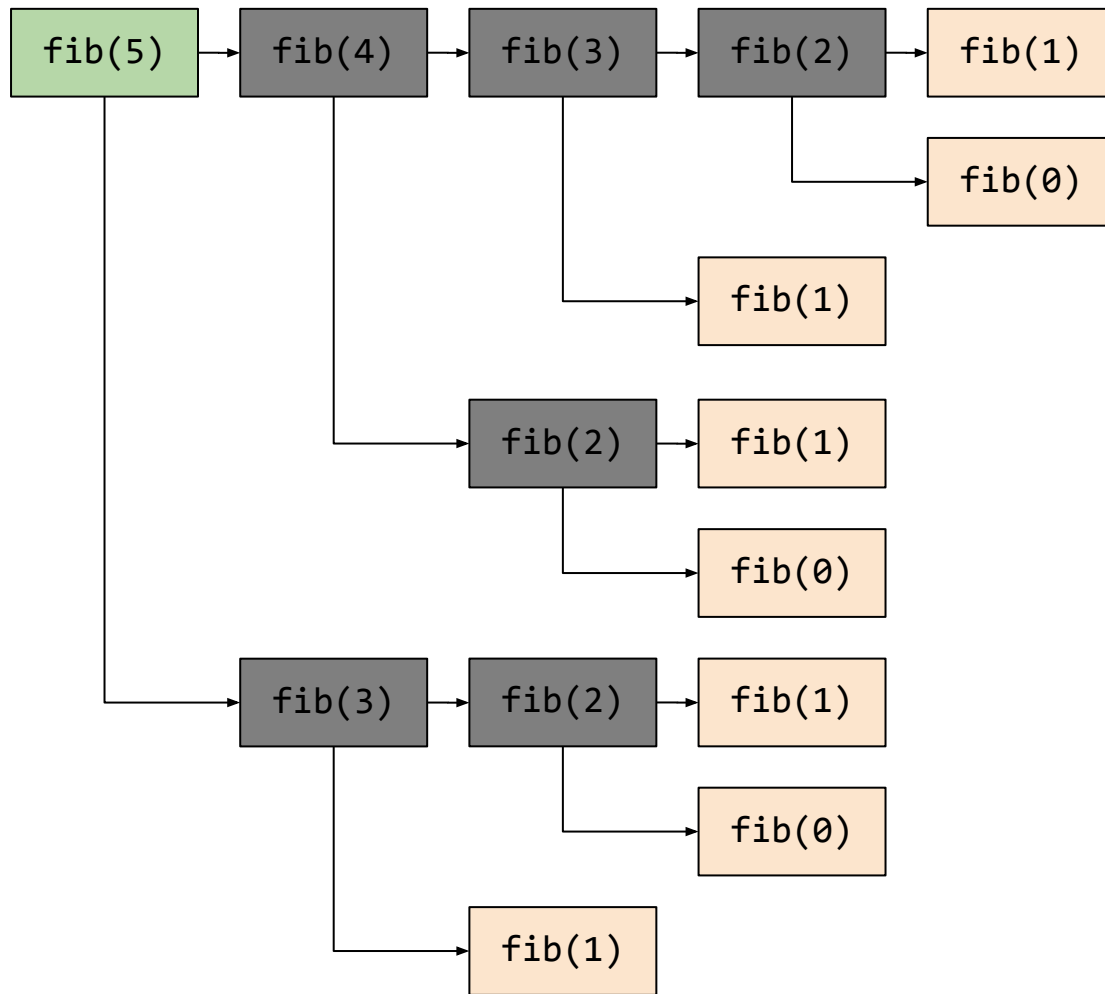
Se requiere una tabla con **tantas dimensiones como parámetros** tenga la función.

Bottom-Up: Pasos

1. Utilizar el grafo para armar la tabla
2. Entender qué significa cada elemento de la misma
3. Comenzar a llenar la tabla con la estrategia de resolución, en forma incremental
4. Ver cuál de los valores de la tabla debe devolverse como solución del problema general

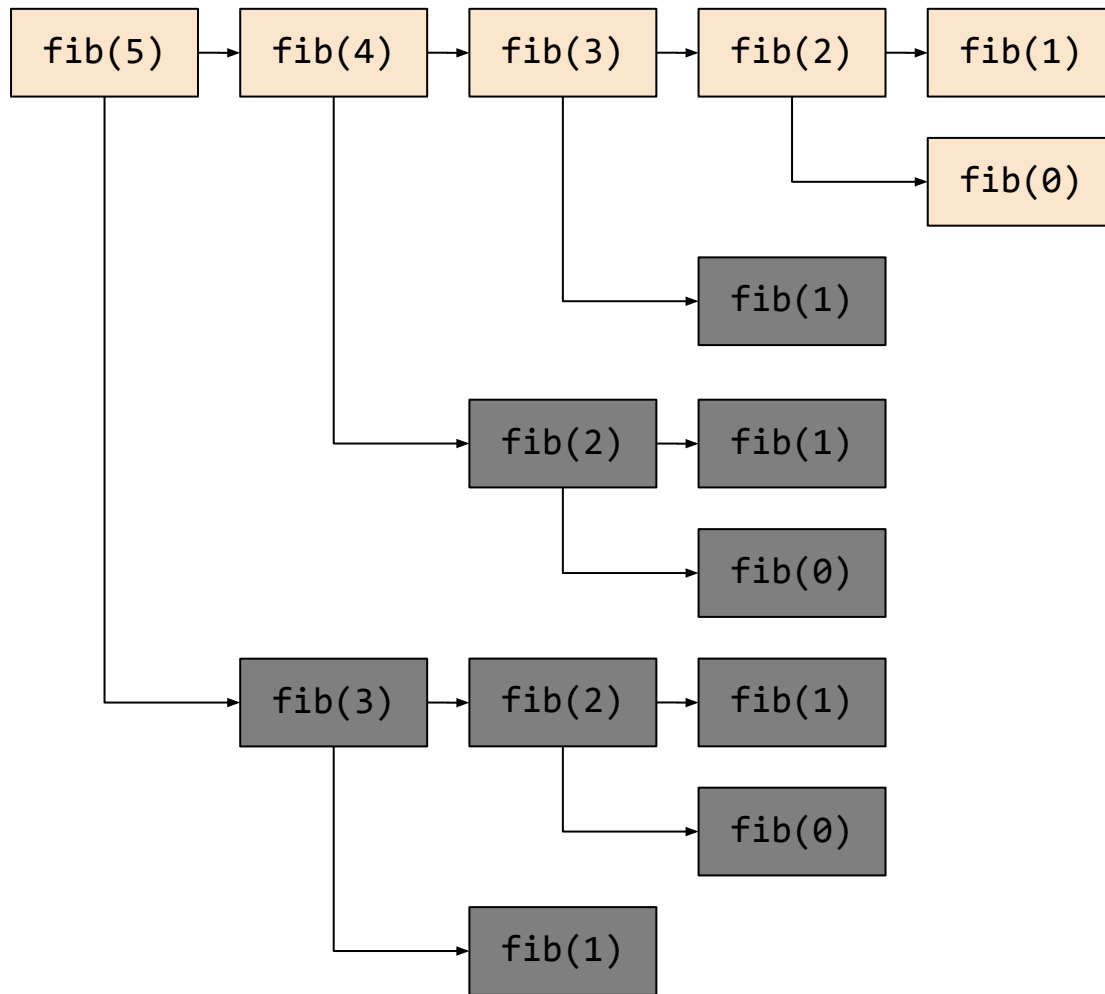
Bottom-Up: Ejemplo

1. Armar el
grafo dirigido
acíclico y la
tabla



Bottom-Up: Ejemplo

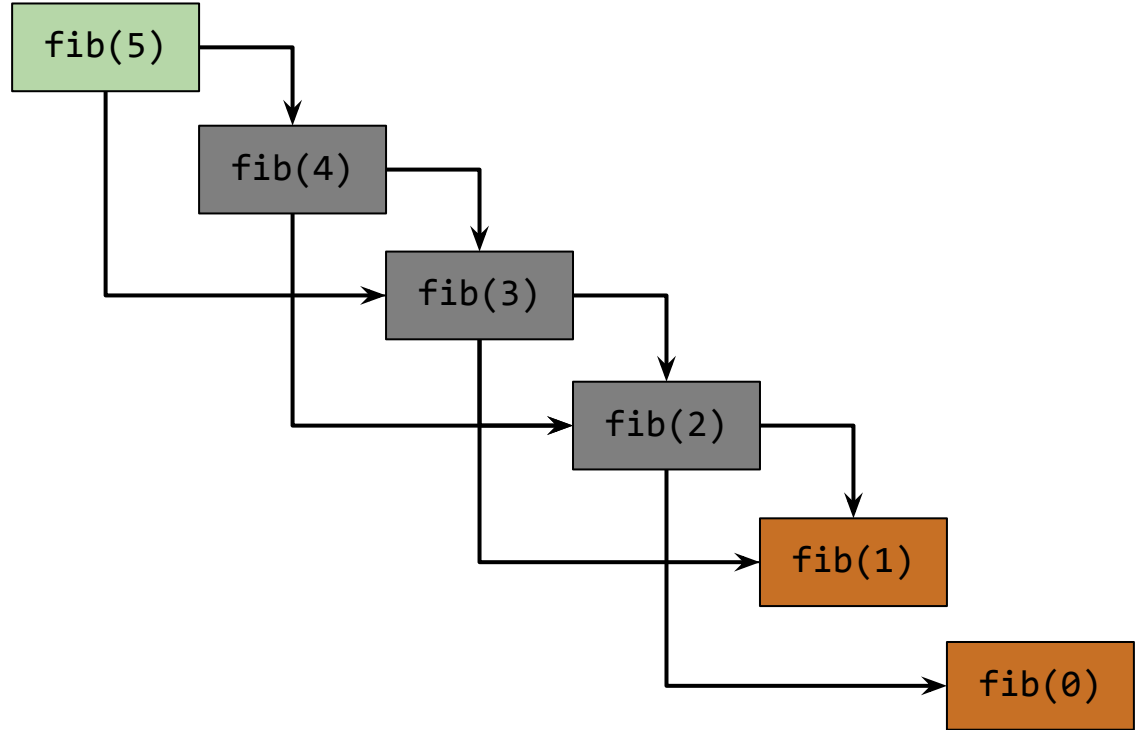
1. Armar el
grafo dirigido
acíclico y la
tabla



Bottom-Up: Ejemplo

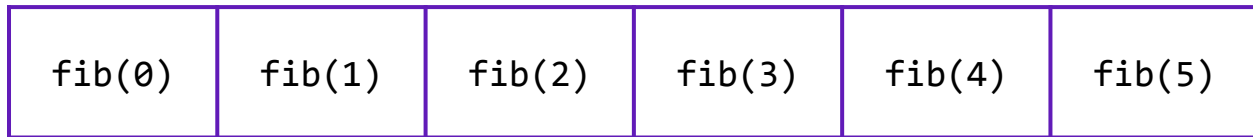
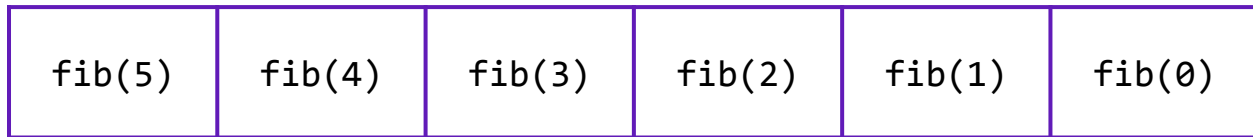
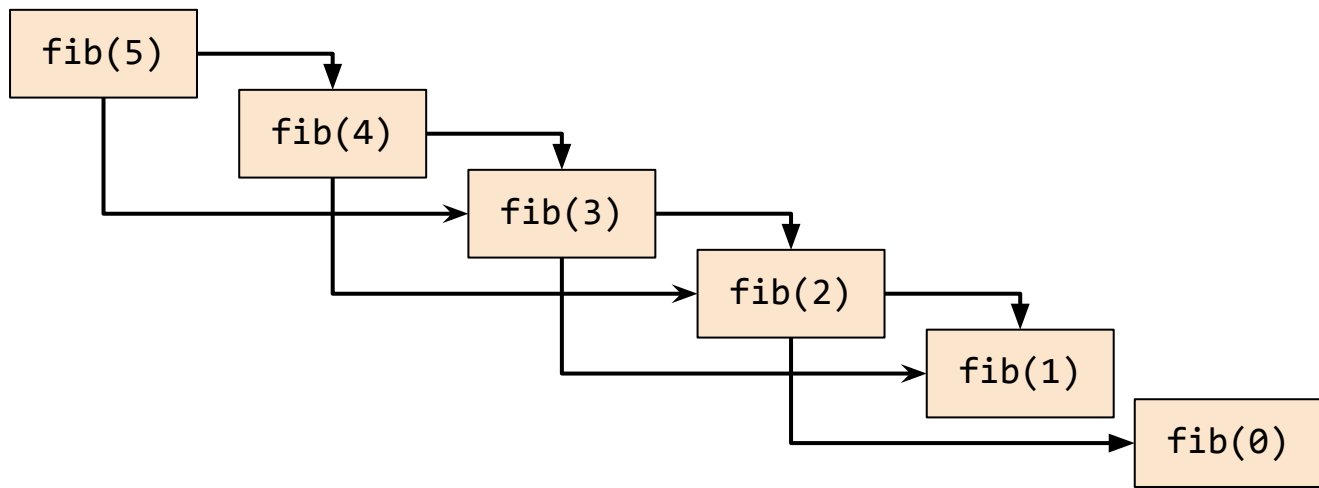
1. Armar el
grafo dirigido
acíclico y la
tabla

Otra vista del
GDA



Bottom-Up: Ejemplo

1. Armar el
grafo dirigido
acíclico
y la tabla



Bottom-Up: Ejemplo

2. Entender los
elementos de
la tabla

<code>t[i]</code>	<code>fib(0)</code>	<code>fib(1)</code>	<code>fib(2)</code>	<code>fib(3)</code>	<code>fib(4)</code>	<code>fib(5)</code>
<code>i</code>	0	1	2	3	4	5


Para cada elemento i , $t[i] = \text{fib}(i)$. Entonces...

<code>fib[n]</code>	<code>fib(0)</code>	<code>fib(1)</code>	<code>fib(2)</code>	<code>fib(3)</code>	<code>fib(4)</code>	<code>fib(5)</code>
<code>n</code>	0	1	2	3	4	5

Bottom-Up: Ejemplo

3. Comenzar a
llenar la tabla
en forma
incremental

n	0	1	2	3	4	5
fib(n)	0 (*)	1 (*)				




(*) Dado por definición

Bottom-Up: Ejemplo

3. Comenzar a
llenar la tabla
en forma
incremental

n	0	1	2	3	4	5
fib(n)	0 (*)	1 (*)	1			




(*) Dado por definición

Bottom-Up: Ejemplo

3. Comenzar a
llenar la tabla
en forma
incremental

n	0	1	2	3	4	5
fib(n)	0 (*)	1 (*)	1	2		




(*) Dado por definición

Bottom-Up: Ejemplo

3. Comenzar a
llenar la tabla
en forma
incremental

n	0	1	2	3	4	5
fib(n)	0 (*)	1 (*)	1	2	3	



(*) Dado por definición

Bottom-Up: Ejemplo

3. Comenzar a
llenar la tabla
en forma
incremental

n	0	1	2	3	4	5
fib(n)	0 (*)	1 (*)	1	2	3	5

(*) Dado por definición

Bottom-Up: Ejemplo

3. Comenzar a
llenar la tabla
en forma
incremental

```
public int fibonacci(int n) {  
    int[] tabla = new int[n + 1];  
    // tabla[0] = 0;  
    tabla[1] = 1;  
    for (int i = 2; i < tabla.length; i++) {  
        tabla[i] = tabla[i-1] + tabla[i-2];  
    }  
    return -1; // ¿qué debería devolver?  
}
```

Bottom-Up: Ejemplo

4. Ver cuál
elemento
retornar

n	0	1	2	3	4	5
fib(n)	0	1	1	2	3	5

El elemento a retornar será el último calculado. ¿Por qué?

1. Es el objetivo (n)
2. No tiene sentido continuar calculando más allá de n
3. Por la forma de construcción

Bottom-Up: Ejemplo

4. Ver cuál
elemento
retornar

```
public int fibonacci(int n) {  
    int[] tabla = new int[n + 1];  
    // tabla[0] = 0;  
    tabla[1] = 1;  
    for (int i = 2; i < tabla.length; i++) {  
        tabla[i] = tabla[i-1] + tabla[i-2];  
    }  
    return tabla[n];  
}
```

Bottom-Up: Ejemplo

Una mejora
posible

```
public int fibonacci(int n) {  
    int[] tabla = new int[n + 1];  
    // tabla[0] = 0;  
    tabla[1] = 1;  
    for (int i = 2; i < tabla.length; i++) {  
        tabla[i] = tabla[i-1] + tabla[i-2];  
    }  
    return tabla[n];  
}
```

Cuando para calcular el *iésimo* valor en forma tabulada sólo se requieran los *k elementos inmediatos anteriores*, podemos **reemplazar la tabla por variables individuales**

Bottom-Up: Ejemplo

Una mejora
posible

```
public int fibonacci(int n) {  
    if (n < 2) {  
        return n;  
    }  
    int anteriorDelAnterior = 0;  
    int anterior = 1;  
    int actual = 1;  
  
    for (int i = 2; i <= n; i++) {  
        actual = anteriorDelAnterior + anterior;  
        anteriorDelAnterior = anterior;  
        anterior = actual;  
    }  
    return actual;  
}
```