Estructuras de Datos

Programación Dinámica

Motivación - Superposición de Subproblemas

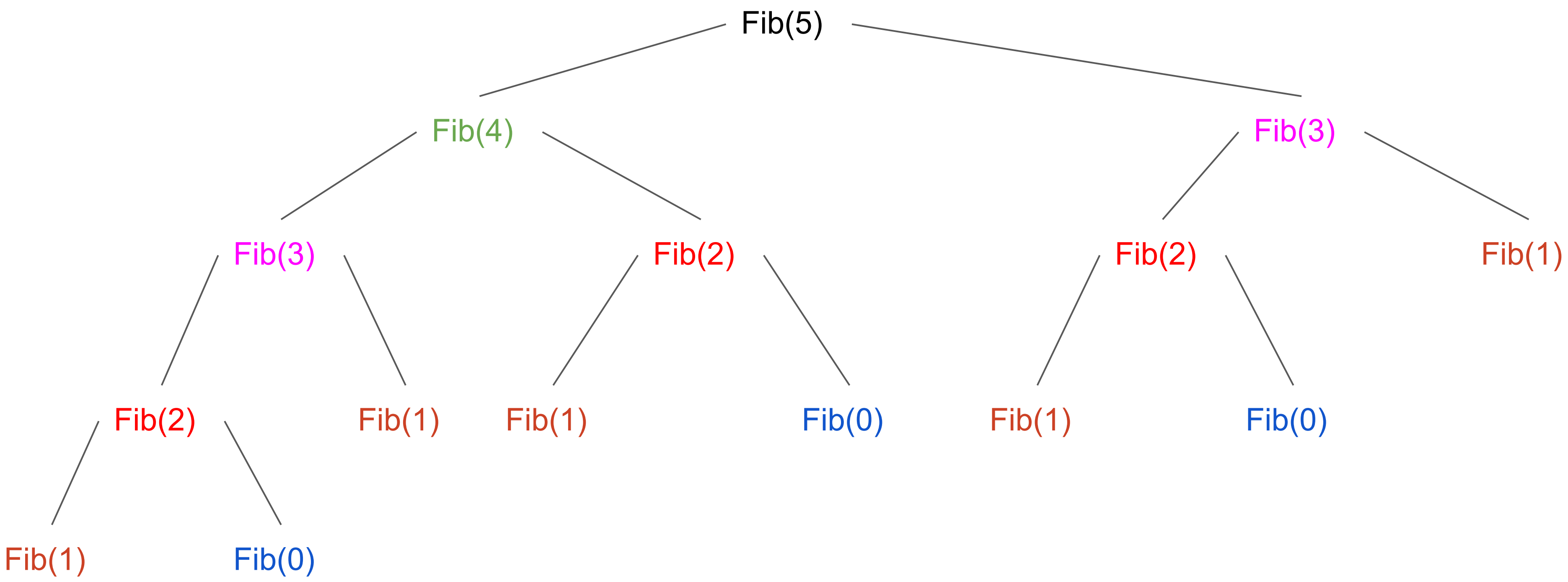
Siguiendo la idea de división de un problema en sub-problemas presentada con los algoritmos de Divide and Conquer, y resolver el problema en base a la solución de problemas más simples. Es posible toparse con sub-problemas que se **superpongan**, es decir, que a su vez compartan sub-instancias o soluciones parciales. Si continuamos con la idea de resolverlas independientemente nos topamos con que estaríamos **resolviendo varias veces las mismas instancias**, una y otra vez, gastando mucho poder de cómputo.

Para observar este hecho veamos una implementación de la función de Fibonacci:

- fib(0) = 1
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)

Motivación - Superposición de Subproblemas

Cómputo de Fib(5):



Motivación - Superposición de Subproblemas

Si bien la definición recursiva se puede implementar fácilmente

```
int rec_fib(int n){
  int res = 1;
  if (n > 1)
    res = rec_fib(n-1) + rec_fib(n-2);
  return res;
}
```

Nos lleva a una solución al problema muy costosa innecesariamente: Veamos que para el caso de `rec_fib`(5) necesitamos calcular `rec_fib`(4) y `rec_fib`(3). Pero para calcular `rec_fib`(4) necesitaremos calcular nuevamente `rec_fib`(3) (y además `rec_fib`(2)), etc.

Memorización

Una posible solución a este problema es identificar que ciertos sub-problemas volverán a aparecer, y así guardar sus soluciones para no tener que volver a computarlas. Es decir, podemos invertir en el uso de una *tabla* o espacio en memoria, para **memorizar** resultados que serán utilizados nuevamente.

Esta técnica es conocida como **Memorización**

Aplicaremos entonces esta técnica al cómputo de la función de Fibonacci.

Memorización - Función de Fibonacci

Para evitar tener que recomputar resultados lo que haremos es **memorizar** el resultado de la llamada a Fibonacci en un arreglo, y utilizar dicho valor para computar los valores siguientes. Siguiendo con una implementación recursiva obtenemos:

```
// Asumimos que |n| se encuentra dentro de la memoria.
int mem_Fib(int mem[], int n){
  // Si n es 0 o 1, Fib(0) = Fib(1) = 1
  if (n < 2) mem[n] = 1;
  // En el caso que mem[n] no se haya computado.
  else if (mem[n] == 0)
    mem[n] = mem_fib(mem, n-1) + mem_fib(mem, n-2);
  return mem[n];
}
```

Motivación - Bottom-Up

Los algoritmos de programación Dinámica utilizan la técnica denominada *bottom-up*. Es decir, comienzan de las subinstancias más chicas (y sencillas), combinando sus soluciones y así construyendo una solución al problema general.

Por ejemplo, en vez de comenzar a computar Fib(n), y por consecuencia Fib(n-1) y Fib(n-2), y por consecuencia … Se comienza desde Fib(0), y Fib(1), a partir de ellos Fib(2), a partir de ellos Fib(3) y así, hasta llegar a Fib(n).

Por su contraparte, los algoritmos de Divide-and-Conquer siguen una técnica denominada *top-down*, donde intentan resolver el problema general directamente, dividiéndolo y resolviendo las subinstancias necesarias que se requieran, y uniendo su resultado.

El ejemplo top-down es la definición recursiva usual de Fib(n).

Bottom-Up - Función de Fibonacci

Revisemos la implementación recursiva de Fibonacci.
Podemos observar que una llamada a `rec_fib`(n) con n > 1, genera dos llamadas recursivas, y estas otras dos llamadas recursivas y así sucesivamente. Es decir, 1 llamada, genera 2 llamadas, que generan 4 llamadas, que generan 8 llamadas, y así hasta llegar a los casos base (0 o 1). Esto genera un orden de $O(2^n)$ llamadas recursivas.
Esto se produce por la forma recursiva en que fue implementada la función, y que comenzamos directamente por el caso que buscamos. Es decir, primero queremos tener el resultado de `rec_fib`(n) y eso genera que calculemos `rec_fib`(n-1) y `rec_fib`(n-2), y estos a su vez los valores para (n-3) y (n-4), y así.
Lo que haremos entonces es calcular los valores al revés, comenzando desde `rec_fib`(0), `rec_fib`(1), y subiendo hasta alcanzar el valor de `rec_fib`(n).

Bottom-Up - Función de Fibonacci

Cambiamos el flujo del programa, y reescribir la función Fib utilizando la técnica de Bottom-Up:

```
int bup_Fib(int n){
  // Declaramos la memoria donde guardaremos los resultados parciales
  int mem[n+1];
  // Comenzando desde Fib(0), Fib(1),
  mem[0] = mem[1] = 1;
  // E iremos incrementando el valor computando Fib(2), Fib(3)
  // hasta alcanzar Fib(n)
  for (int i = 2; i <= n;i ++){
    mem[i] = mem[i-1] + mem[i-2];
  }
  return mem[n];
}
```

Bottom-Up - Función de Fibonacci

Analizando el cuerpo de `bup_Fib`(n) podemos ver que **reemplazamos** las llamadas recursivas por un bucle for. El bucle itera desde el valor 2 hasta n, y dentro de él se realizan accesos al arreglo `mem`. Dado que el acceso es constante, O(1), y se realiza unas n-2 veces, la complejidad computacional de `bup_Fib`(n) es O(n), y dado que usamos un arreglo de memoria de longitud n+1, su complejidad espacial es O(n).

Finalmente si observamos con atención **no** es necesario recordar **todos** los valores ya computados. Para computar el n-ésimo número de Fibonacci sólo se requiere el (n-1)-ésimo y el (n-2)-ésimo, es decir, **los dos anteriores**.

Optimización - Función de Fibonacci

Con lo que llegamos a una versión optimizada de Fibonacci:

```
int opt_Fib(int n){
  // Declaramos sólo dos variables que usaremos
  // para almacenar el último y penúltimo valor computado.
  int pFib, ppFib;
  int res = 1;
  if (n > 1){
    pFib = ppFib = 1;
    // Realizamos el computo Bottom-Up comenzando desde 2 hasta n
    for(int i = 2; i <= n; i ++){
      // Defino como el penúltimo valor el último de la iteración anterior
      ppFib = pFib;
      // Defino como último al resultado de la iteración anterior
      pFib = res;
      // Computo el resultado para la iteración actual
      res = pFib + ppFib;
    }
  }
  return res;
}
```

Programación Dinámica - Función de Fibonacci

La versión optimizada de Fibonacci `opt_Fib` tiene un costo computacional de O(n) pero un costo constante O(1) de memoria (debido a las variables utilizadas en la implementación).

Resumiendo el recorrido desde la definición inicial recursiva `rec_fib` hasta la versión optimizada `opt_Fib` mediante la aplicación de dos técnicas memorización y bottom-up, logramos reducir el costo computacional de $O(2^n)$ a O(n).

En conjunto estas técnicas dan lugar a lo que se conoce como la estrategia de Programación Dinámica. Un último requerimiento es la noción de subestructuras óptimas visto en la presentación de Greedy. Nuevamente debido a la construcción de la solución a partir de soluciones parciales.