Programación Dinámica

¿Qué es la programación Dinámica?

La programación Dinámica es una técnica para diseñar algoritmos la cual podemos pensar como buscar en todo el espacio de forma eficiente. De manera informal la podemos ver como una fuerza bruta "inteligente".

¿Para qué sirve normalmente la programación dinámica?

Usualemte existe dos tipos de problemas que suele requerir programación dinámica.

- Problemas de optimización, es decir, maximizar o minimizar.
- Contar la cantidad de maneras de hacer tal cosa.

Características de La Programación dinámica.

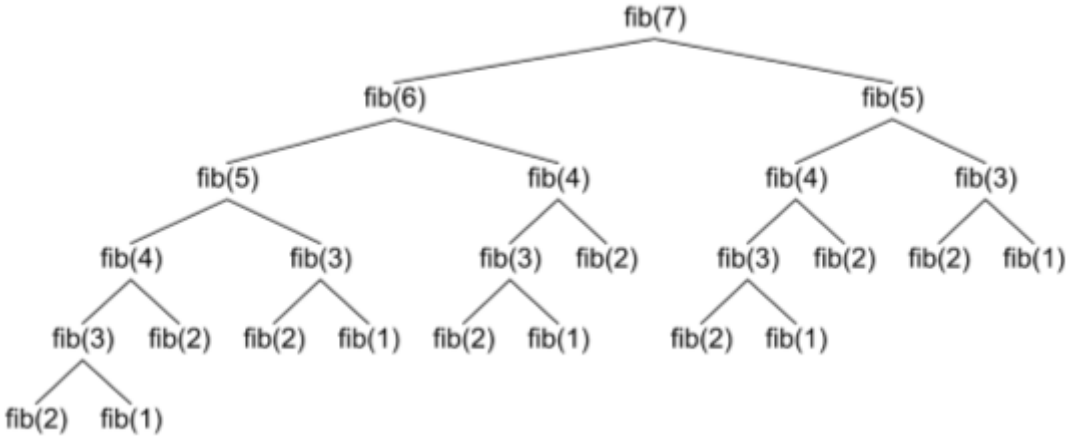
La programación dinámica puede ser aplicada si el problema puede ser dividido en subproblemas con traslapes los cuales se pueden resolver independientemente.

EJEMPLO SENCILLO

```
int fibonacci(int n){
   if( n== 0 || n == 1){
       return n;
   }
   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Sabemos que esta función obtenemos la el n-ésimo término de fibonacci. pero, ¿es esto eficiente? Sabemos que la recurrencia de la sucesión de fibonacci se da por: $fib_n = fib_{n-1} + fib_{n-2}$ con los casos $fib_0 = 0, fib_1 = 1$

Notemos algo en el siguiente arbol:



Notemos que por ejemplo:

fib_4 se repite numerosas veces y lo tenemos que calcular cada vez que se requiera, algo tonto pues ya lo hemos calculado antes. ¿Cómo lo solucionamos? pues simplemente podemos guardar cuanto vale fib_4 y cuando se necesite solo lo retornamos.

Con lo que con este ejemplo lo podemos optimizar usando programación dinámica.

```
vector<int> memorizacion(200, -1);//200 espacios con -1 iniciados

int fibonacci(int n){
   if( n== 0 || n == 1) return n;
   if(memorizacion[n] != -1 ) return memorizacion[n];
   return memorizacion[n] = fibonacci(n - 1) + fibonacci(n - 2);//si no
esta calculado lo guardamos para l proxima
}
```

Analicemos lo que acabamos de hacer

1. Identificamos que distingue a estado (en este caso únicamente n)
2. Dividimos el problema de en subproblemas ($fib_n = fib_{n-1} + fib_{n-2}$)
3. Identificamos casos base ($fib_0 = 0, fib_1 = 1$)
4. Notamos que habia traslapes de subproblemas.(fib_4 repetido)
5. Resolvimos separadamente cada sub-problema.(resolver fib_{n-1} y fib_{n-2})
6. Memorizamos subproblemas.

Aproximación por Bottom-Up

Sin mencionarlo al resolver el sencillo problema de la sucesión de fibonacci usamos una técnica que se llama Top-Down, es decir construimos la solución "desde arriba", de la raiz del arbol a las hojas, ahora

vamos a ver como construir la solución desde las hojas a la raíz que se le conoce como Bottom-Up suele ser preferida pues no ocupa recurrencia y puede ser más factible ocupar diferentes técnicas en ella.

Ocupamos los mismo pasos anteriores.

Ahora en vez de una recursión ocupamos un arreglo uni-dimensional (En este caso es unidimensional pues la función de recursión solo necesita una entrada(datos que distinguen los estados.)

Podemos usar el siguiente código para la versión Bottom-Up.

```
vector<int> Fibo[200];
Fibo[0] = 0;
Fibo[1] = 1; //casos base;
for(int i=2 ; i<10; i++){//decimo termino de la sucesión
   Fibo[i] = Fibo[i -1] + Fibo[i - 2]
}
```