Programación Dinámica en Grafos

Quién? Carlos Miguel Soto

Cuándo? 2019-10-24

Introducción

¿Qué es la Programación Dinámica (dp)?

Ejemplo clásico – Sucesión de Fibonacci.

$$F_n = \begin{cases} 0 & \text{Si } n = 0 \\ 1 & \text{Si } n = 1 \\ F_{n-1} + F_{n-2} & \text{Si } n \geq 2 \end{cases}$$

Función recursiva

Podemos implementar la definición directamente...

```
1  int fib(int n) {
2      if(n == 0) return 0;
3      if(n == 1) return 1;
4
5      int res = fib(n-1) + fib(n-2);
6      return res;
7  }
```

Función recursiva

Podemos implementar la definición directamente...

```
1   int fib(int n) {
2       if(n == 0) return 0;
3       if(n == 1) return 1;
4
5       int res = fib(n-1) + fib(n-2);
6       return res;
7   }
```

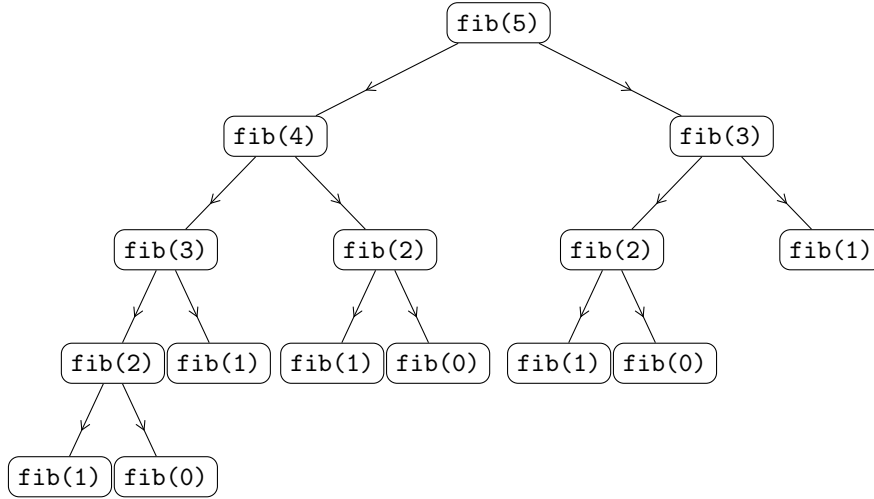
... pero es super lenta!

Función recursiva

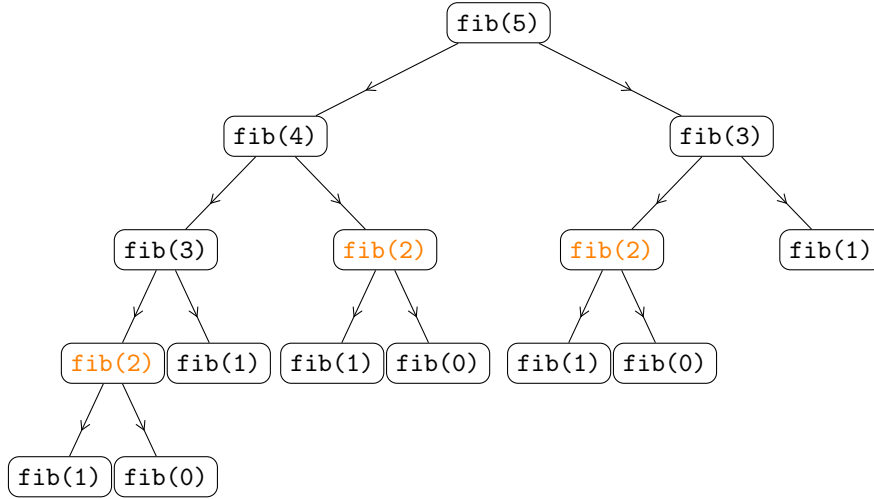
- Complejidad: $\mathcal{O}(\varphi^n)$ (Exponencial)
- $n = 45$ tarda ~ 17 segundos

[1] φ es el número de oro y vale $\varphi = 1.68\ldots$

Función recursiva



Función recursiva



Memoización

```
1   unordered_map<int, int> memo;
2
3   int fib(int n) {
4       if(n == 0) return 0;
5       if(n == 1) return 1;
6
7       if(memo.count(n)) return memo[n];
8       res = fib(n-1) + fib(n-2);
9       memo[n] = res;
10      return res;
11  }
```

... es mucho más rápida!

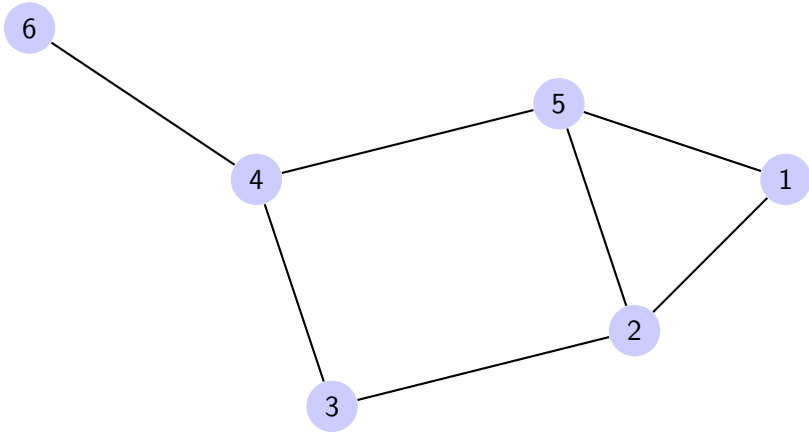
Función recursiva

- Complejidad: $\mathcal{O}(n)$ (Lineal).
- Computa `fib(`10^5`)` sin problemas.

Esto es la dp

- Pensar el problema como una función recursiva
- Computar cada valor a lo sumo una vez
- A la hora de implementar, *no siempre conviene hacer la recursión explícita*

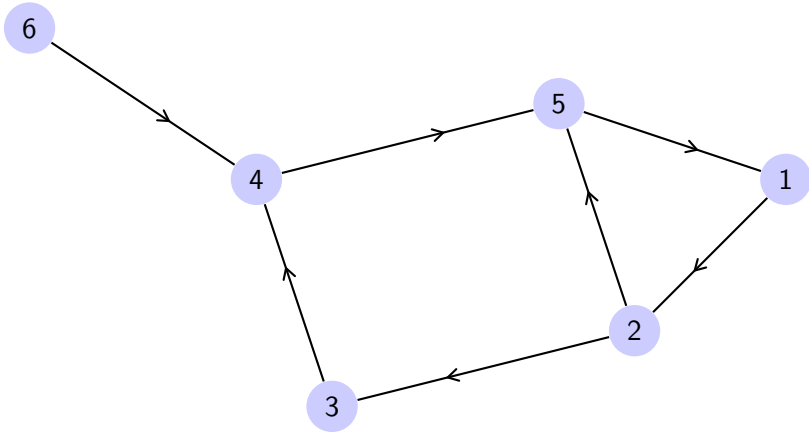
¿Qué es un grafo?



¿Qué es un grafo? – Variaciones

- *Dirigidos*
- Pesados

Dirigidos



¿Qué es un grafo? – Variaciones

- Dirigidos
- *Pesados*

¿Qué es un grafo? – Representación

En código, la forma más sencilla y eficiente de representar un grafo suele ser con *listas de adyacencia*

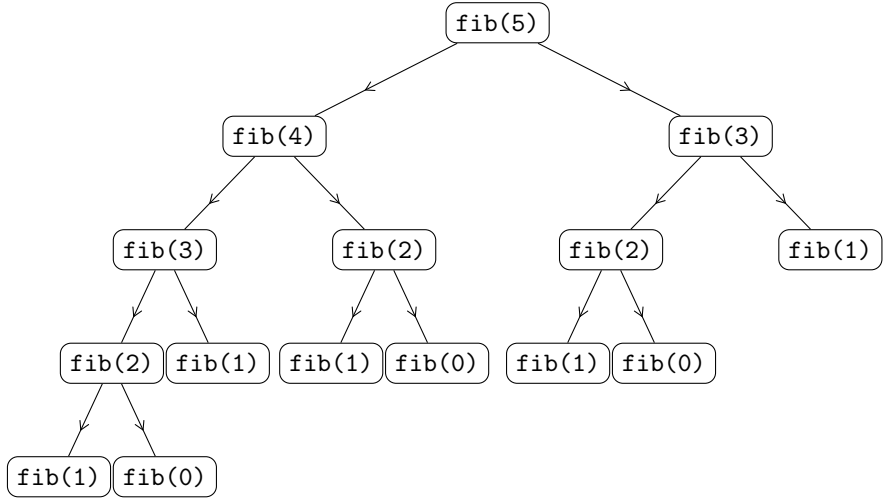
```
1  int main() {
2      int n, m;
3      cin >> n >> m;
4      vector<vector<int>> grafo(n);
5      for(int i = 0; i < m; ++i) {
6          int a, b;
7          cin >> a >> b;
8          grafo[a].push_back(b);
9          grafo[b].push_back(a);
10     }
11 }
```

¿Para qué sirven?

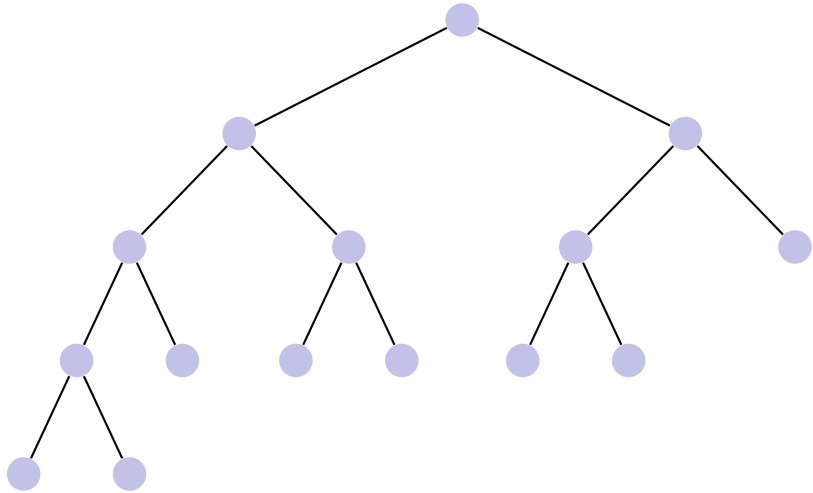
¿Para qué sirven?

- modelan muchisimos problemas (directa o indirectamente)
- nos van a servir para modelar la recursión

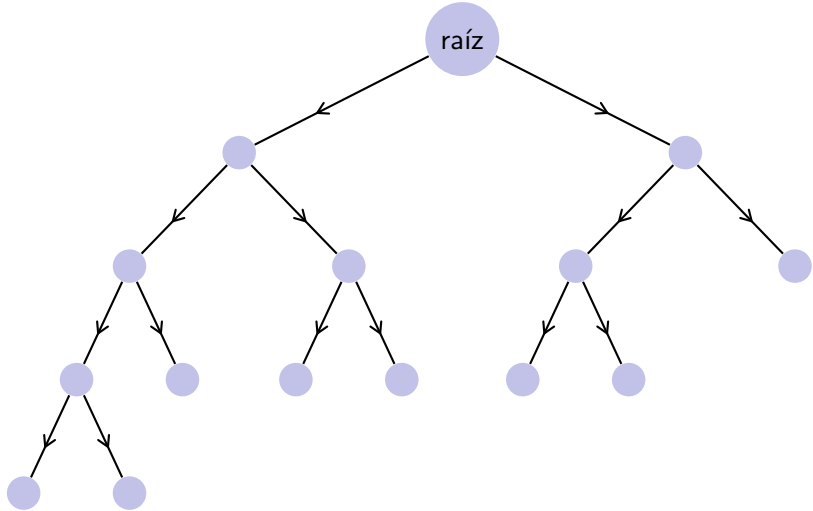
call graph de `fib`



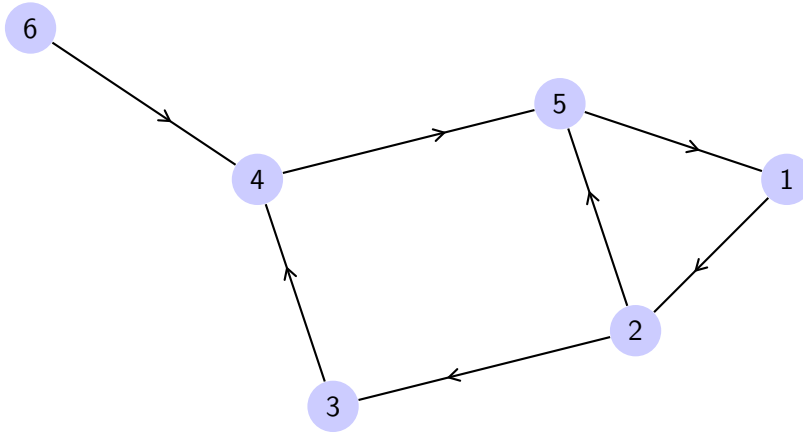
Árboles



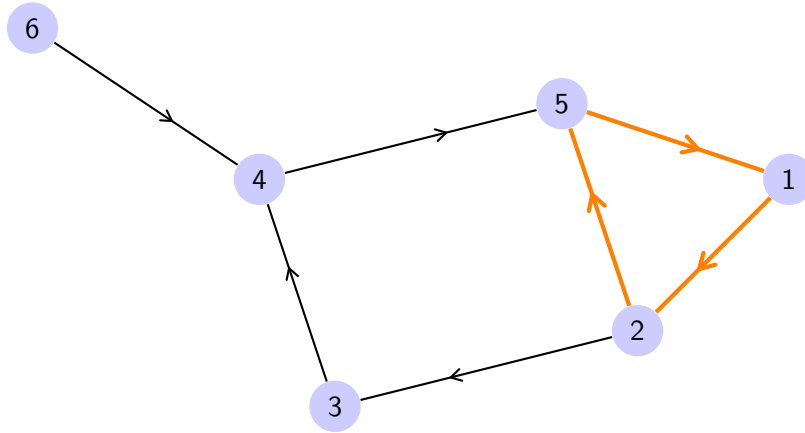
Árboles



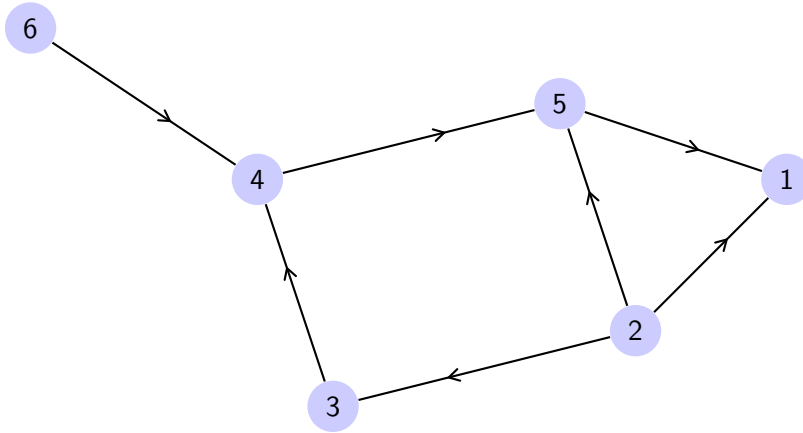
DAG = *Directed* Acyclic Graph



DAG = Directed *Acyclic* Graph



DAG = Directed *Acyclic* Graph



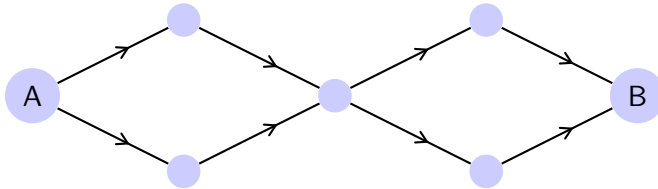
Dps en DAGs

Dps en DAGs – Cantidad de Caminos

Dado un DAG G con dos nodos $A, B \in G$, contar cuántos caminos hay entre A y B en G.
Contar la cantidad de caminos válidos.

- Opción 1: enumero los caminos de a uno $\Rightarrow$ exponencial

Dps en DAGs – Cantidad de Caminos



Dps en DAGs – Cantidad de Caminos

Dado un DAG G con dos nodos $A, B \in G$, contar cuántos caminos hay entre A y B en G.
Contar la cantidad de caminos válidos.

- Opción 1: enumero los caminos de a uno $\Rightarrow$ exponencial
- Opción 2: dp!

Cantidad de Caminos – Función de la Dp

`caminos(nodo)` = cantidad de caminos entre `nodo` y B

Cantidad de Caminos – Función de la Dp

$$\texttt{caminos(nodo)} = \text{cantidad de caminos entre nodo y B}$$

$$\texttt{caminos(nodo)} = \begin{cases} 1 & \text{Si nodo == } B \\ \sum_{\texttt{nodo} \to \texttt{v}} \texttt{caminos(v)} & \text{Si nodo != } B \end{cases}$$

```cpp
vector<vector<int>> grafo;
vector<int> memo(grafo.size(), -1);

int caminos(int nodo) {
    if(nodo == B) return 1;
    if(memo[nodo] != -1) return memo[nodo];

    int res = 0;
    for(int v : graph[nodo]) res += caminos(v);

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

Dps en DAGs – Cantidad de Descendientes

Dado un DAG G para cada nodo A, contar la cantidad de nodos a los que se puede llegar desde A (sus descendientes)

Dps en DAGs – Cantidad de Descendientes

Dado un DAG G para cada nodo A, contar la cantidad de nodos a los que se puede llegar desde A (sus descendientes)

Hacemos lo mismo

Dps en DAGs – Cantidad de Descendientes

Hacemos lo mismo:

$$\texttt{num\_descendientes(nodo)} = \text{cantidad de descendientes de } \texttt{nodo}$$
$$\texttt{num\_descendientes(nodo)} = 1 + \sum_{\texttt{nodo} \to \texttt{v}} \texttt{num\_descendientes(v)}$$

Dps en DAGs – Cantidad de Descendientes

Hacemos lo mismo:

$$\texttt{num\_descendientes(nodo)} = \text{cantidad de descendientes de } \texttt{nodo}$$
$$\texttt{num\_descendientes(nodo)} = 1 + \sum_{\texttt{nodo} \to \texttt{v}} \texttt{num\_descendientes(v)}$$

... pero esto no anda

Dps en DAGs – Cantidad de Descendientes

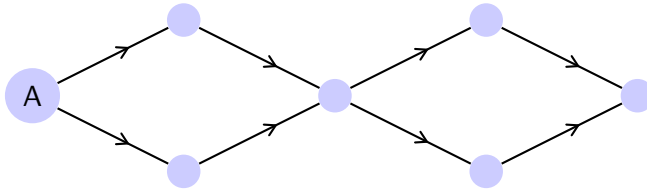
Hacemos lo mismo:

$$\texttt{num\_descendientes(nodo)} = \text{cantidad de descendientes de } \texttt{nodo}$$
$$\texttt{num\_descendientes(nodo)} = 1 + \sum_{\texttt{nodo} \to \texttt{v}} \texttt{num\_descendientes(v)}$$

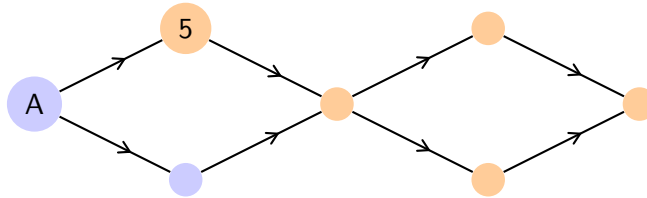
... pero esto no anda

¿Porqué?

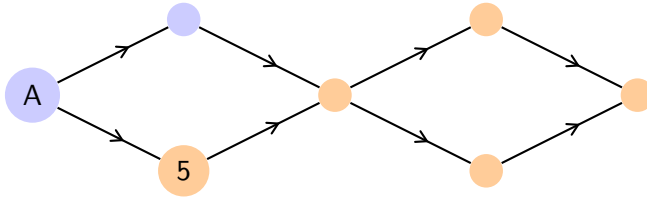
Dps en DAGs – Cantidad de Descendientes



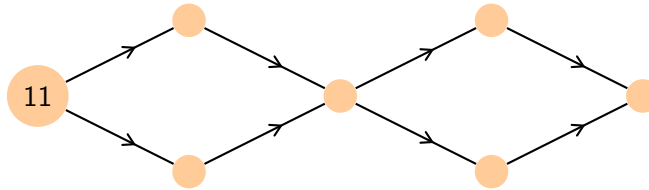
Dps en DAGs – Cantidad de Descendientes



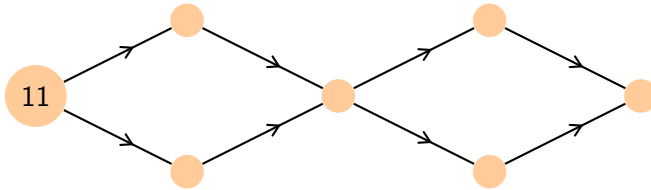
Dps en DAGs – Cantidad de Descendientes



Dps en DAGs – Cantidad de Descendientes



Dps en DAGs – Cantidad de Descendientes



Contamos doble los que son descendientes de dos de mis hijos.

¿Cómo lo arreglamos?

Tenemos que computar algo más fuerte: el *conjunto* de mis descendientes.

¿Cómo lo arreglamos?

$$\texttt{descendientes(nodo)} = \text{conjunto de descendientes de } \texttt{nodo}$$

$$\texttt{descendientes(nodo)} = \{\textit{nodo}\} \cup \bigcup_{\texttt{nodo} \to \texttt{v}} \texttt{descendientes(v)}$$

¿Cómo lo arreglamos?

```
1  vector<bitset<n>> memo;
2  bitset<n> descendientes(int node) {
3      if(memo[node].count() != 0) return memo[node];
4
5      memo[node] = true;
6      for(int v : graph[node]) {
7          memo[node] |= descendientes(v);
8      }
9      return memo[node];
10 }
```

Clausura Transitiva

- Complejidad: $\mathcal{O}(N \cdot E)$
- Nos computa además la *clausura transitiva*
- Se puede hacer con `bitset`, osea que es super rápido!

Esta idea se repite

En general, cuando hacemos una dp tenemos que hacer un balance entre:

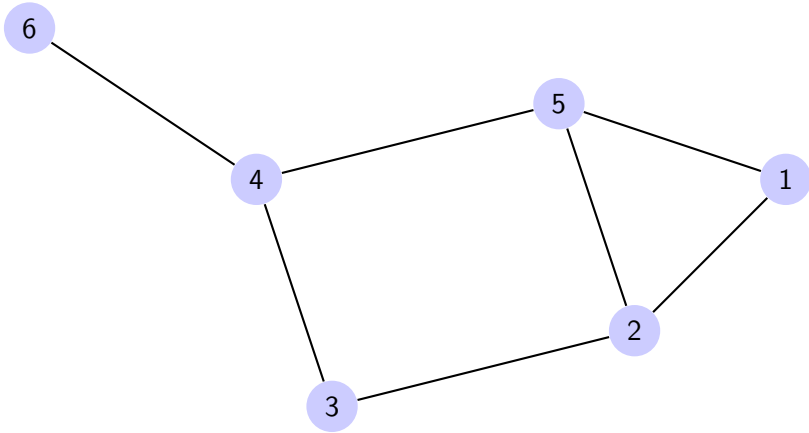
Lo que queremos computar en cada nodo

La información que tenemos de los nodos hijos

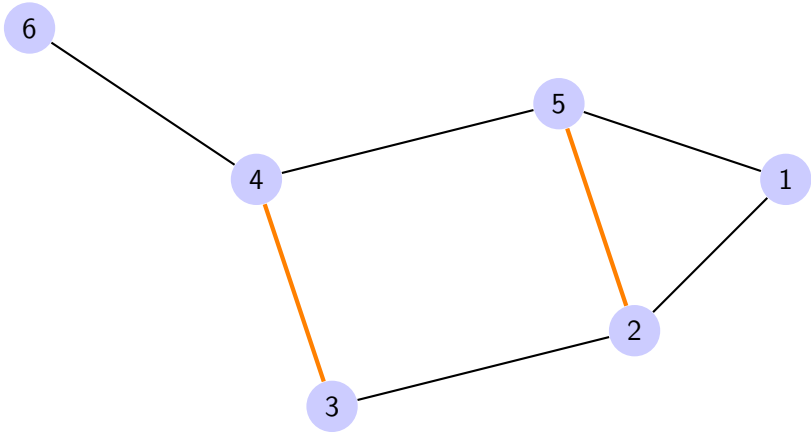
Otro Ejemplo – Maximum Matching

Dado un grafo G, encontrar el tamaño del mayor subconjunto de aristas que no contenga dos aristas incidentes en un mismo nodo

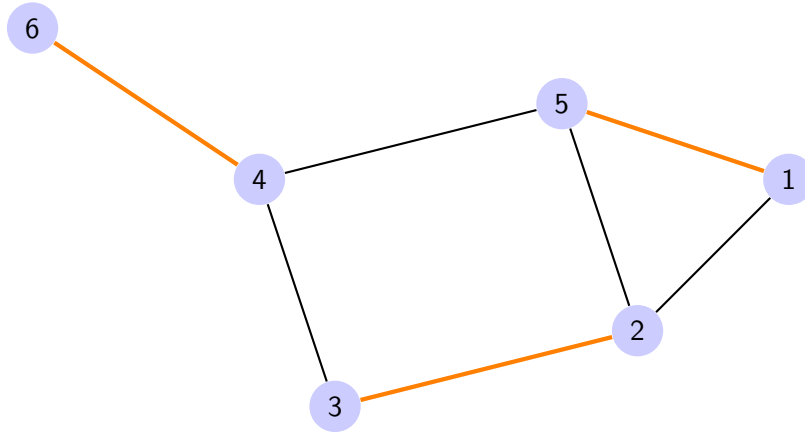
Otro Ejemplo – Maximum Matching



Otro Ejemplo – Maximum Matching



Otro Ejemplo – Maximum Matching



Maximum Matching en Árboles

Vamos a ver un caso particular del problema: maximum matching en árboles.

- El problema general es muy difícil.
- Para árboles sale con dp!

Maximum Matching en Árboles – Cuál es nuestra función recursiva?

- Intento 1: `mm(nodo)` = tamaño del maximo matching del subarbol de `nodo`.

  Falla, ya que no tenemos suficiente información para computar el resutado en un nodo dado el resultado en sus hijos

Vamos a tener que computar más información

- Intento 2: `mm(nodo)` va a devolver *dos* elementos:
  - `mm(nodo).matched`, el máximo matching si `nodo` está utilizado.
  - `mm(nodo).unmatched`, el máximo matching si `nodo` no está utilizado.

Eso si alcanza para computar el resultado en función a sus hijos.

Maximum Matching en Árboles

$$\texttt{mm(nodo).matched} = \sum_{\texttt{node}\to\texttt{v}} \max \begin{pmatrix} \texttt{mm(v).matched}, \\ \texttt{mm(v).unmatched} \end{pmatrix}$$

Maximum Matching en Árboles

$$\texttt{mm(nodo).matched} = \sum_{\texttt{node} \to \texttt{v}} \max \begin{pmatrix} \texttt{mm(v).matched}, \\ \texttt{mm(v).unmatched} \end{pmatrix}$$

$$\texttt{mm(nodo).unmatched} = \max_{v \to nodo} \left(1 + \texttt{mm(v).unmatched} + \sum_{nodo \to u, u \neq v} \texttt{mm(u).matched} \right)$$

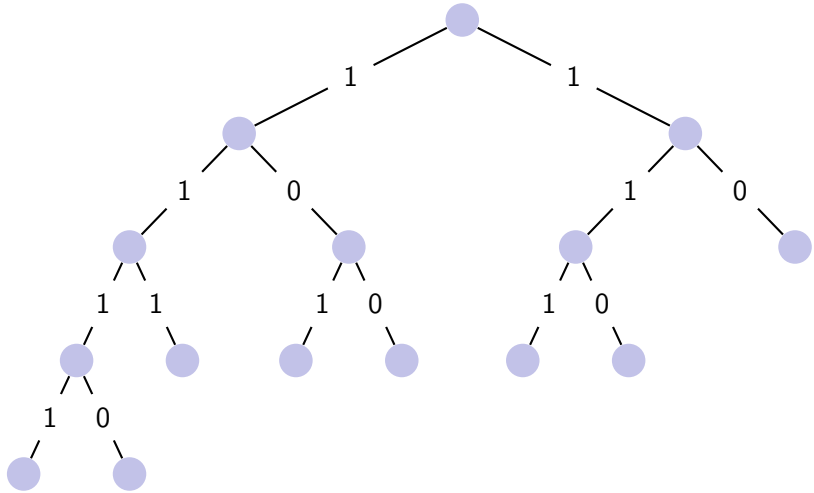
```
1  Dp mm(int node) {
2      Dp res;
3
4      res.matched = 0;
5      for(int v : graph[node]) res.matched +=
6          max(mm(v).matched, mm(v).unmatched);
7
8      int sum = 0;
9      for(int v : graph[node])
10         sum += mm(v).matched;
11
12     res.unmatched = 0;
13     for(int v : graph[node]) {
14         res.unmatched = max(res.unmatched,
15             1 + mm(v).unmatched + (sum - mm(v).matched)
16         );
17     }
18
19     return res;
20 }
```

Problema Ejemplo – 0-1-Tree

Dado un Árbol con los números 1 o 0 escritos en las aristas, decimos que un camino es *válido* si los números en sus aristas son decrecientes.

Contar la cantidad de caminos válidos.

Ejemplo



0-1-Tree – Approaches

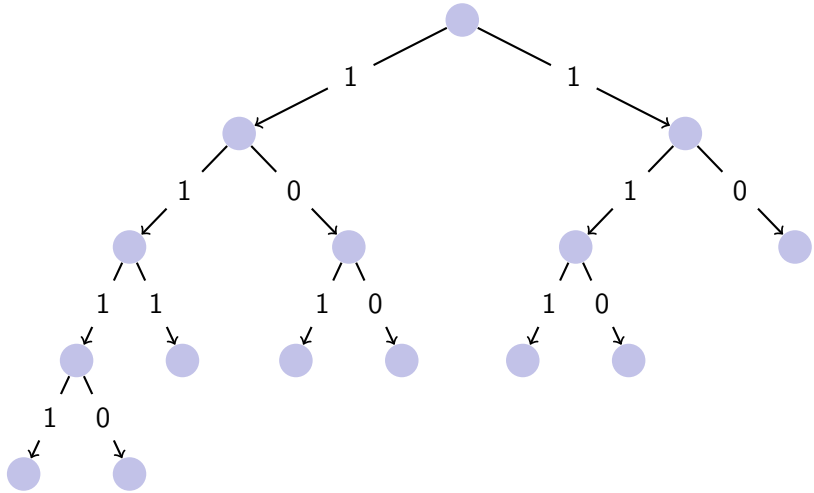
- Opción 1: Iterar todos los caminos y chequear si son válidos $\Rightarrow$ Complejidad $\mathcal{O}(n^3)$ a priori.
- Opción 2: dp! $\Rightarrow$ Complejidad $\mathcal{O}(n)$

0-1-Tree – Pasos para la dp

¿Qué necesitamos para la dp?

- Establecer el DAG: rooteamos el árbol.
- ¿Qué argumentos va a tener la función recursiva?
- ¿Qué va a devolver la función recursiva?

Ejemplo



0-1-Tree – Pasos para la dp

¿Qué necesitamos para la dp?

- Establecer el DAG: rooteamos el árbol.
- ¿Qué argumentos va a tener la función recursiva?
- ¿Qué va a devolver la función recursiva?

0-1-Tree – Pasos para la dp

¿Qué necesitamos para la dp?

- Establecer el DAG: rooteamos el árbol.
- ¿Qué argumentos va a tener la función recursiva?
- ¿Qué va a devolver la función recursiva?

```
1 struct Dp {
2     int numero_caminos;
```

```
1  struct Dp {
2      int numero_caminos;
3
4      // #caminos todos 0 que terminan en nodo
5      int ceros;
6      // #caminos decrecientes que terminan en nodo
7      int decrecientes;
```

```
1  struct Dp {
2      int numero_caminos;
3
4      // #caminos todos 0 que terminan en nodo
5      int ceros;
6      // #caminos decrecientes que terminan en nodo
7      int decrecientes;
8
9      // #caminos todos 1 que terminan en nodo
10     int unos;
11     // #caminos crecientes que terminan en nodo
12     int crecientes;
13 };
```

Implementación

Queda para practicar! Hay que pensar:

Implementación

Queda para practicar! Hay que pensar:

- Cómo calcular la dp en un nodo dado la dp en todos los hijos

Implementación

Queda para practicar! Hay que pensar:

- Cómo calcular la dp en un nodo dado la dp en todos los hijos
- Cómo extraer la información que nos pide el problema de la dp

Implementación

Queda para practicar! Hay que pensar:

- Cómo calcular la dp en un nodo dado la dp en todos los hijos
- Cómo extraer la información que nos pide el problema de la dp (esto último es fácil porque queda guardado en `Dp.numero_caminos`)

- Toposort: ¿para qué sirve?
- ¿Cuántas formas hay de poner dominós en un tablero de $1 \times n$? ¿Y de $2 \times n$?
- Queries de substring de máxima suma, con updates puntuales.

Fin

¿Preguntas?

Fin

¿Preguntas?

¡Gracias!