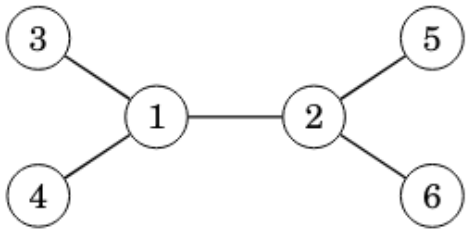
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: LOS CAMINOS MÁS LARGOS DE LOS NODOS DE UN ÁRBOL

---

1. Introducción

Digamos que tenemos un árbol y queremos calcular para cada nodo del árbol la longitud máxima de un camino que comienza en ese nodo. Esto puede verse como una generalización del problema del diámetro del árbol, porque la mayor de esas longitudes es igual al diámetro del árbol.

Como ejemplo, considere el siguiente árbol:



Sea $maxLength(x)$ la longitud máxima de una ruta que comienza en el nodo x. Por ejemplo, en el árbol anterior, $maxLength(4) = 3$, porque hay una ruta $4 \rightarrow 1 \rightarrow 2 \rightarrow 6$. Aquí hay una tabla completa de los valores:

node x	1	2	3	4	5	6
$maxLength(x)$	2	2	3	3	3	3

En la presente guía abordaremos como determinar para cada nodo del árbol la longitud máxima de un camino que comienza en ese nodo

2. Conocimientos previos

2.1. Recorrido en Profundidad (*DFS*)

Búsqueda en profundidad. Una Búsqueda en profundidad (en inglés *DFS* o *Depth First Search*) es un algoritmo que permite recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

2.2. Recorrido a lo Ancho (*BFS*)

En Ciencias de la Computación, Búsqueda a lo ancho (en inglés *BFS - Breadth First SearchRe*) es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A

continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

2.3. Programación Dinámica

La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.

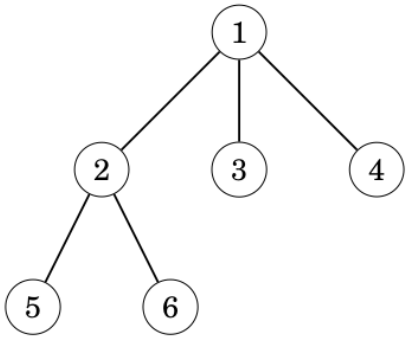
La teoría de programación dinámica se basa en una estructura de optimización, la cual consiste en descomponer el problema en subproblemas (más manejables). Los cálculos se realizan entonces recursivamente donde la solución óptima de un subproblema se utiliza como dato de entrada al siguiente problema. Por lo cual, se entiende que el problema es solucionado en su totalidad, una vez se haya solucionado el último subproblema. Dentro de esta teoría, Bellman desarrolla el Principio de Optimalidad, el cual es fundamental para la resolución adecuada de los cálculos recursivos. Lo cual quiere decir que las etapas futuras desarrollan una política óptima independiente de las decisiones de las etapas predecesoras. Es por ello, que se define a la programación dinámica como una técnica matemática que ayuda a resolver decisiones secuenciales interrelacionadas, combinándolas para obtener de la solución óptima.

3. Desarrollo

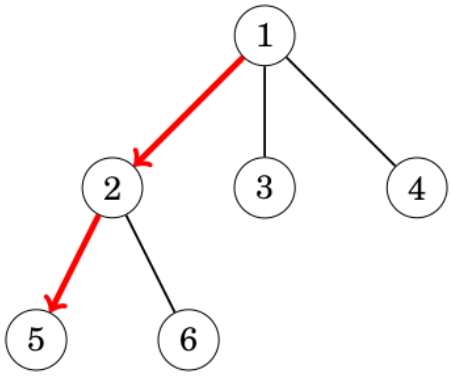
La idea trivial de la solución a este ejercicio sería realizar un DFS por cada nodo y para dicho ver al nodo más lejano que puede llegar y este sería el camino maslargo para el nodo por donde se comenzo el DFS. Esta idea el único inconveniente es el alto costo temporal para cuando el árbol tenga gran cantidad de nodos ya que su complejidad temporal esta dada por la expresión O($N(N+E)$) donde N sería la cantidad de nodos del árbol y E las aristas del mismo que por definición de un árbol sería $N-1$ por lo que sustituyen en la expresión inicial nos queda O($2N^2 - N$). Pero esta idea del DFS nos va a servir para ver la primera variante de solución.

3.1. Programación dinámica con DFS

También en este problema, un buen punto de partida para resolver el problema es rootear el árbol arbitrariamente: La primera parte del problema es calcular para cada nodo x la longitud

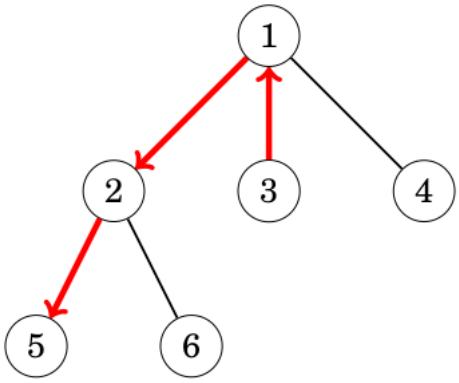


máxima de una ruta que pasa por un hijo de x. Por ejemplo, la ruta más larga del nodo 1 pasa por su hijo 2:



Esta parte es fácil de resolver en el tiempo O(N), porque podemos usar la programación dinámica.

Luego, la segunda parte del problema es calcular para cada nodo x la longitud máxima de una ruta a través de su padre p. Por ejemplo, la ruta más larga del nodo 3 pasa por su padre 1:

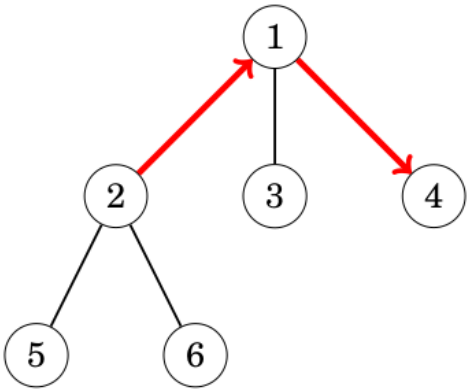


A primera vista, parece que debemos elegir el camino más largo desde p. Sin embargo, esto no siempre funciona, porque el camino más largo de p puede pasar por x. Aquí hay un ejemplo de esta situación:

Aún así, podemos resolver la segunda parte en el tiempo O (n) almacenando dos longitudes máximas para cada nodo x:

- $maxLength_1(x)$**:** La longitud máxima de una ruta desde x
- $maxLength_2(x)$**:** La longitud máxima de una ruta desde x en otra dirección que la primera ruta

Por ejemplo, en el anterioranterior, $maxLength_1(1) = 2$ usando la ruta $1 \to 2 \to 5$, y $maxLength_2(1) = 1$ usando la ruta $1 \to 3$.



Finalmente, si la ruta que corresponde a $maxLength_1(p)$ pasa por x, concluimos que la longitud máxima es $maxlength_2(p) + 1$, y de lo contrario la longitud máxima es la $maxLength_1(p) + 1$.

3.2. Usando BFS

Otra idea algorítmica para resolver este problema se basa en la utilización del algoritmo *BFS* con ciertas modificaciones y una estructura como vector o arreglo para almacenar para cada nodo el máximo camino donde él es uno de los extremos.

La idea algorítmica parte del díametro del árbol donde un árbol puede tener uno o varios díametros representados por todos aquellos caminos cuya longitud es máxima en todo el árbol y que comienza en un nodo u y terminan en un nodo v.

Para cualquier nodo z el máximo camino que lo comprende y comienza en él va terminar siempre en un nodo de tipo u o v. Por tanto basta con seleccionar un par (u, v) en el árbol y realizar un bfs partiendo sobre cada uno de ellos y ver para cada nodo del árbol de cual de estos dos nodos esta más lejos y ese será el máximo camino para el nodo. Para llevar esta idea vamos a tener dos elementos importantes:

1. Un vector o arreglo $path$ con la capacidad igual a la cantidad de nodos con un valor bien pequeño inicialmente. La idea de este vector o arreglo es que voy almacenar en la posición $path[i]$ la longitud máxima de un camino en el árbol donde uno de los extremos de dicho camino es el nodo i.
2. Implementar un *BFS* que retorne el último nodo visitado y que calcule la distancia desde el nodo que se comenzo el *BFS* hacia cualquier otro nodo i del árbol si dicha distancia es mayor que la que se tiene almacenada en el $path$ para el nodo i actualizar el valor de $path[i]$ con dicho valor.

Una vez visto estos detalles el procedimiento sería bien sencillo:

1. Realizar el *BFS* modificado desde cualquier nodo del árbol escogido de forma arbitraria este nos va devolver un posible nodo u.

2. Realizar el *BFS* modificado desde el nodo u obtenido en el paso anterior. Esta ejecucción del *BFS* nos va devolver un posible nodo v asociado con u cuyo camino entre ellos es un diámetro del árbol.

3. Realizar el *BFS* modificado desde el nodo v obtenido en el paso anterior.

Una vez realizados estos tres *BFS* en el vector o arreglo $path$ en la posición i tenemos el valor de la longitud del máximo que comienza o termina en el nodo i.

4. Implementación

4.1. C++

4.1.1. Programación dinámica con DFS

```
struct Node {
   vector<int> neighborns;
   int firstMaxPath; // Primer camino mas largo
   int secondMaxPath; // Segundo camino mas largo
   int nodeChildMaxPath; // Nodo hijo o padre que salio el camino mas largo
};

vector<Node> tree;

void dfsLongestPathsChild(int v, int p){
   tree[v].firstMaxPath = tree[v].secondMaxPath = 0;
   for (auto x : tree[v].neighborns) {
      if (x == p) continue;
      dfsLongestPathsChild(x, v);
      if (tree[x].firstMaxPath + 1 > tree[v].firstMaxPath) {
         tree[v].secondMaxPath = tree[v].firstMaxPath;
         tree[v].firstMaxPath = tree[x].firstMaxPath + 1;
         tree[v].nodeChildMaxPath = x;
      } else if (tree[x].firstMaxPath + 1  > tree[v].secondMaxPath) {
         tree[v].secondMaxPath= tree[x].firstMaxPath + 1;
      }
   }
}

void dfsLongestPathsParent(int v, int p){
   for (auto x : tree[v].neighborns) {
      if (x == p) continue;
      if (tree[v].nodeChildMaxPath == x) {
         if (tree[x].firstMaxPath < tree[v].secondMaxPath + 1) {
            tree[x].secondMaxPath = tree[x].firstMaxPath;
            tree[x].firstMaxPath = tree[v].secondMaxPath + 1;
            tree[x].nodeChildMaxPath = v;
         } else {
```

```
            tree[x].secondMaxPath = max(tree[x].secondMaxPath,tree[v].
                secondMaxPath + 1);
         }
      } else {
         tree[x].secondMaxPath = tree[x].firstMaxPath;
         tree[x].firstMaxPath = tree[v].firstMaxPath + 1;
         tree[x].nodeChildMaxPath = v;
      }
      dfsLongestPathsParent(x, v);
   }
}

void allLongestPathsTree(){
   dfsLongestPathsChild(1,-1);
   dfsLongestPathsParent(1,-1);
}
```

4.1.2. Usando BFS

```
struct Node {
   vector<int> neighborns;
   int maxPath = LONG_MIN;
};

int nnodes;

vector<Node> tree;

int bfs(int src) {
   int top;
   queue<int> q;
   vector<int> d(nnodes+1, -1);
   d[src] = 0;
   tree[src].maxPath = max(tree[src].maxPath, d[src]);
   q.push(src);
   while(!q.empty()) {
      top = q.front(); q.pop();
      for(int v: tree[top].neighborns) {
         if(d[v] == -1) {
            q.push(v);
            d[v] = d[top] + 1;
            tree[v].maxPath = max(tree[v].maxPath, d[v]);
         }
      }
   }
   return top;
}
```

```
void allLongestPathsTree(){
   int diam_end_1 = bfs(1);
   int diam_end_2 = bfs(diam_end_1);
   bfs(diam_end_2);
}
```

4.2. Java

4.2.1. Programación dinámica con DFS

```
public class Main {
   private class Node{
      public List<Integer> neighborns ;
      public int firstMaxPath; // Primer camino mas largo
      public int secondMaxPath; // Segundo camino mas largo
          public int nodeChildMaxPath; // Nodo hijo o padre que salio el
              camino mas largo

      public Node() {
         firstMaxPath =Integer.MIN_VALUE;
         secondMaxPath = Integer.MIN_VALUE;
         nodeChildMaxPath = Integer.MIN_VALUE;
         neighborns =new ArrayList<Integer>();
      }
   }

   private Node [] tree ;

   public void dfsLongestPathsChild(int v, int p){
      tree[v].firstMaxPath = tree[v].secondMaxPath = 0;
      for (int x : tree[v].neighborns) {
         if (x == p) continue;
         dfsLongestPathsChild(x, v);
         if (tree[x].firstMaxPath + 1 > tree[v].firstMaxPath) {
            tree[v].secondMaxPath = tree[v].firstMaxPath;
            tree[v].firstMaxPath = tree[x].firstMaxPath + 1;
            tree[v].nodeChildMaxPath = x;
         } else if (tree[x].firstMaxPath + 1  > tree[v].secondMaxPath) {
            tree[v].secondMaxPath= tree[x].firstMaxPath + 1;
         }
      }
   }

   public void dfsLongestPathsParent(int v, int p){
       for (int x : tree[v].neighborns) {
          if (x == p) continue;
          if (tree[v].nodeChildMaxPath == x) {
             if (tree[x].firstMaxPath < tree[v].secondMaxPath + 1) {
```

```
                tree[x].secondMaxPath = tree[x].firstMaxPath;
                tree[x].firstMaxPath = tree[v].secondMaxPath + 1;
                tree[x].nodeChildMaxPath = v;
            } else {
                tree[x].secondMaxPath = Math.max(tree[x].secondMaxPath,tree[v
                    ].secondMaxPath + 1);
            }
          } else {
            tree[x].secondMaxPath = tree[x].firstMaxPath;
                        tree[x].firstMaxPath = tree[v].firstMaxPath + 1;
                        tree[x].nodeChildMaxPath = v;
                 }
                 dfsLongestPathsParent(x, v);
       }
   }

   public void allLongestPathsTree(){
      dfsLongestPathsChild(1,-1);
      dfsLongestPathsParent(1,-1);
   }
}
```

4.2.2. Usando BFS

```
public class Main {
   private class Node{
      public List<Integer> neighborns ;
      public int maxPath;

      public Node() {
         maxPath =Integer.MIN_VALUE;
         neighborns =new ArrayList<Integer>();
      }
   }

   private Node [] tree ;
   private int nnodes;

   public int bfs(int src) {
      int top=0;
      Queue<Integer> q =new LinkedList<Integer>();
      int [] d = new int[nnodes+2];
      Arrays.fill(d, -1);
      d[src] = 0;
      tree[src].maxPath = Math.max(tree[src].maxPath, d[src]);
      q.add(src);
      while(!q.isEmpty()) {
         top = q.remove();
```

```
            for(int v: tree[top].neighborns) {
               if(d[v] == -1) {
                  q.add(v);
                  d[v] = d[top] + 1;
                  tree[v].maxPath = Math.max(tree[v].maxPath, d[v]);
               }
             }
          }
          return top;
       }

       public void allLongestPathsTree(){
          int diam_end_1 = bfs(1);
          int diam_end_2 = bfs(diam_end_1);
          bfs(diam_end_2);
       }
}
```

5. Aplicaciones

Dentro de las aplicaciones de hallar los caminos más largos de un árbol podemos mencionar:

1. **Redes de comunicación:** en una red de comunicación, el árbol de ruta más largo se puede utilizar para determinar la ruta más eficiente para transmitir datos o mensajes entre diferentes nodos. Al encontrar el árbol de ruta más largo, los administradores de red pueden optimizar el rendimiento de la red y minimizar los retrasos.
2. **Redes de transporte:** los árboles de caminos más largos se pueden aplicar a redes de transporte como carreteras o ferrocarriles. Al identificar los caminos más largos, los planificadores de transporte pueden determinar las rutas más eficientes para vehículos o trenes, reduciendo el tiempo de viaje y la congestión.
3. **Gestión de la cadena de suministro:** en la gestión de la cadena de suministro, se pueden utilizar árboles de ruta más largos para optimizar el flujo de bienes de los proveedores a los clientes. Al identificar los caminos más largos, los gerentes pueden identificar posibles cuellos de botella o ineficiencias en la cadena de suministro y realizar los ajustes necesarios para mejorar la eficiencia general.
4. **Gestión de proyectos:** los árboles de ruta más larga se utilizan comúnmente en la gestión de proyectos para determinar la ruta crítica de un proyecto. La ruta crítica es la ruta más larga a través de las actividades de un proyecto, e identificarla ayuda a los gerentes a programar tareas y asignar recursos de manera efectiva para garantizar la finalización oportuna del proyecto.
5. **Análisis de datos:** en el análisis de datos, los árboles de ruta más largos se pueden utilizar para identificar elementos influyentes o importantes dentro de un conjunto de datos. Al

encontrar los caminos más largos, los analistas pueden identificar variables o factores clave que tienen el impacto más significativo en el resultado que se analiza.

6. **Análisis de redes sociales:** los árboles de ruta más larga se pueden aplicar al análisis de redes sociales para identificar individuos o grupos influyentes dentro de una red. Al encontrar los caminos más largos, los analistas pueden determinar quién tiene más conexiones o influencia dentro de una red social, lo que puede ser útil para estrategias de marketing o para comprender la dinámica social.
7. **Investigación genealógica:** en la investigación genealógica, los árboles de caminos más largos se pueden utilizar para rastrear líneas ancestrales y determinar relaciones entre individuos. Al encontrar los caminos más largos, los investigadores pueden identificar ancestros comunes y construir árboles genealógicos.
8. **Enrutamiento de Internet:** los árboles de ruta más largos se utilizan en protocolos de enrutamiento de Internet como el Border Gateway Protocol (BGP) para determinar la mejor ruta para enrutar paquetes de datos entre diferentes redes. Al encontrar las rutas más largas, los enrutadores pueden tomar decisiones informadas sobre las rutas más eficientes para la transmisión de datos.
9. **Procesos de toma de decisiones:** los árboles de camino más largo se pueden utilizar en los procesos de toma de decisiones para evaluar múltiples opciones y determinar la opción más favorable u óptima. Al encontrar los caminos más largos, los tomadores de decisiones pueden evaluar los posibles resultados y consecuencias de diferentes opciones y tomar decisiones.
10. **Teoría de juegos:** los árboles de caminos más largos se pueden aplicar a la teoría de juegos para analizar las interacciones estratégicas entre jugadores. Al encontrar los caminos más largos, los analistas pueden identificar estrategias óptimas y predecir los posibles resultados de un juego o competencia.

6. Complejidad

En cuanto a la complejidad de los algoritmos analizamos en esta guía podemos mencionar que la primera variante utilizando programación dinámica con dos DFS hace que esta solución tenga una complejidad temporal de O($2(V + E)$) donde V es la cantidad de vértices del árbol y E las aristas, pero reglas del cálculo de complejidades de algoritmo la constante 2 se desprecia (en lo personal eso se ve muy bonito para las reglas pero para la programación competitiva es importante tener en cuenta para la cantidad de operaciones), otra sustitución dentro de la expresión es que la cantidad de aristas de un árbol es igual a la cantidad de vértices del árbol por tanto $E = V - 1$ y sustituyendo en la expresión final nos queda O($2(2V - 1)$).

En cuanto a la complejidad temporal de la segunda variante como se realiza 3 *BFS* la misma es O($3(V + E)$) que haciendo un análisis similar al anterior y sustiyendo E por su equivalente en función de V nos queda O($3(2V - 1)$).

Viendo ya complejidades de ambas variantes algorítmicas es evidente que la basada en programación dinámica es un tanto mas rápida que la basada en *BFS* aunque a favor de esta que su implementación es más facil de implementar y se basa en una idea mas asequible de entender.

7. Ejercicios

A continuación una lista de ejercicios que pueden ser resueltos aplicando los algoritmos analizados en la presente guía:

- CSES - Tree Distances I