Introduction to Competitive Programming and Algorithms

Decal Lecture 6 - Graph Representations

Announcements

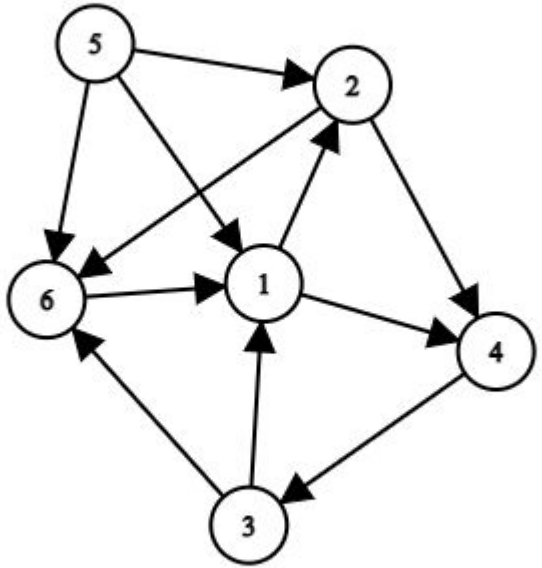
- We had problems with viewing test cases again…
  - Problem A from last week's problemset will no longer be based on correctness, only effort
- This week's problemset will also be based on effort rather than correctness!
  - Some of the problems can be pretty tough to implement

Types of Graphs

1. **Types of Graphs**
2. Graph Representations
3. Example Problems

What is a Graph?

- A collection of vertices (aka nodes) and edges
- Pairs of vertices are connected by edges
- Used to represent things and connections between them

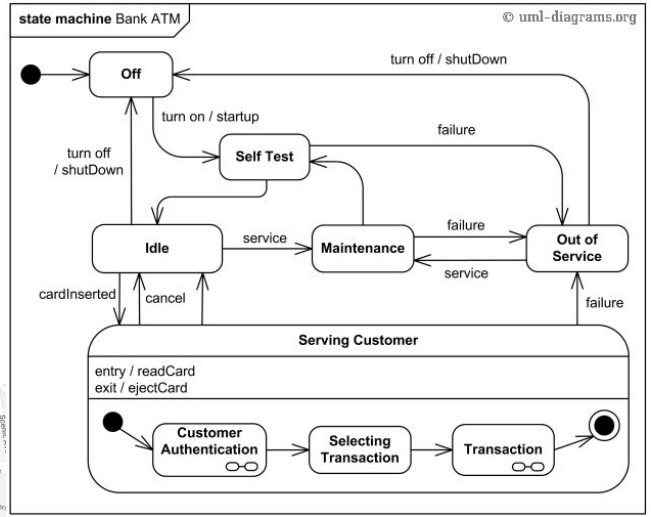


Solving Problems with Graphs

- Model the problem using an appropriate type of graph
- Model graphs using the right data structures in your code
- Consider various graph properties:
  - Weightedness
  - Directedness
  - Degree

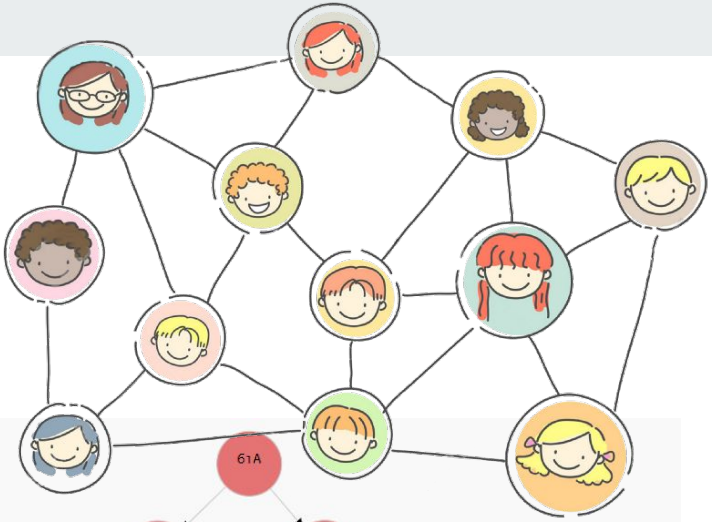
Graph Properties: Weightedness

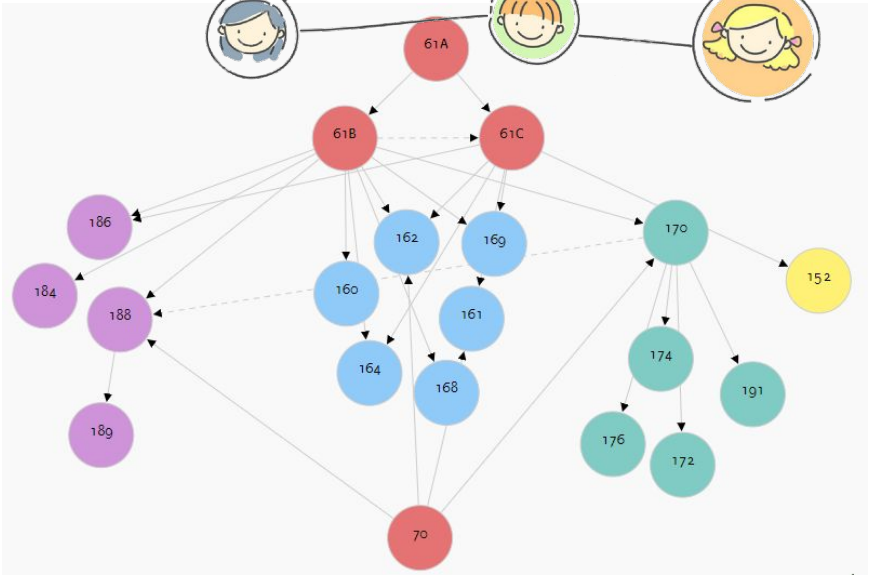
- Weight: associating some value to each edge
- Weighted or unweighted?
  - A map of Berkeley where some roads that are longer than others
  - A state diagram of an ATM machine. Count the number of transitions



Graph Properties: Directedness

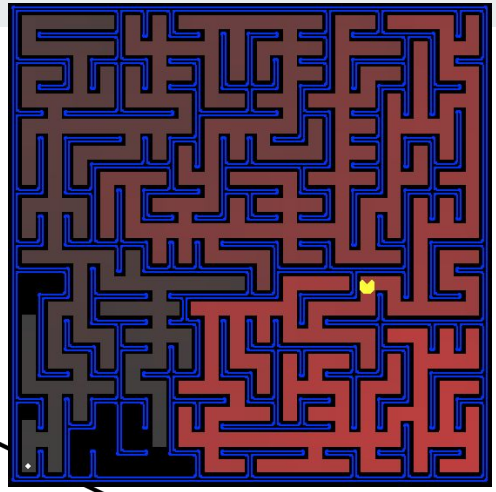
- Directedness: does A -> B also mean the same thing as B -> A?
- Directed or undirected?
  - A social network where both people have to agree before becoming friends
  - HKN's course guide, recommending courses to be taken in a specific order

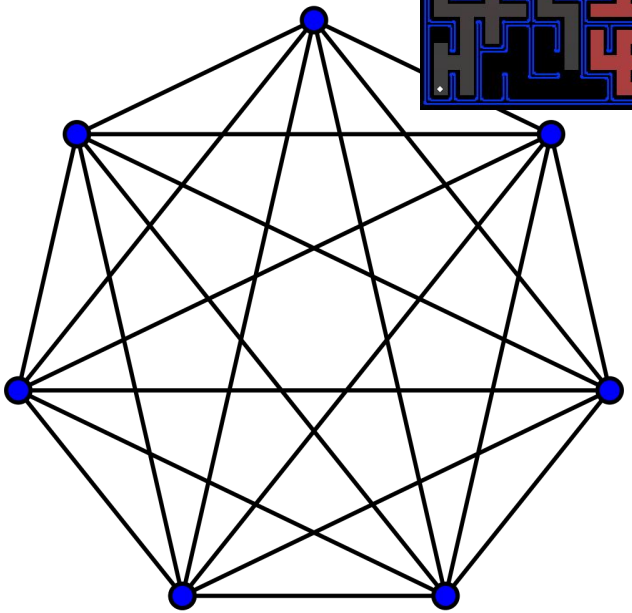




Graph Properties: Degree

- Degree: the number of incoming or outgoing edges a vertex can have
- Not always given guarantees, but sometimes can be useful
- Compare:
  - A maze, where at each step, pacman can move up, down, left, or right
  - A 7-gon, where you can go from any corner to any other corner





Data Structure Considerations

- Operations
  - Insert
  - Lookup
  - Neighbors
  - Delete
- Time complexity
- Space complexity

Graph Representations

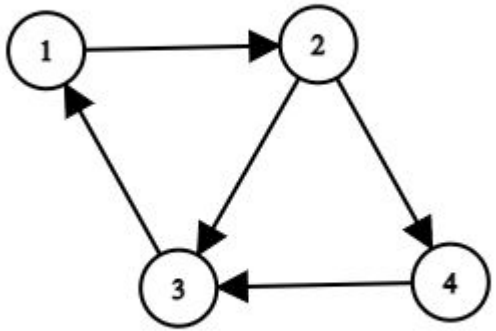
1. Types of Graphs
2. **Graph Representations**
3. Example Problems

Edge List

- List of pairs (i, j) that denote an edge between vertex i and vertex j
- Edges can be given in any order
- Pros
  - Great for sparse graphs
  - Simple to read and write
  - Fast edge insertions
- Cons
  - Slow to perform edge queries
  - Slow to iterate over neighbors
  - Bad for dense graphs

```
4    // total vertices (1, 2, 3, 4)
5    // total edges
1 2
2 4
4 3
3 1
2 3
```

```
[
    [1, 2],
    [2, 4],
    [4, 3],
    [3, 1],
    [2, 3]
]
```

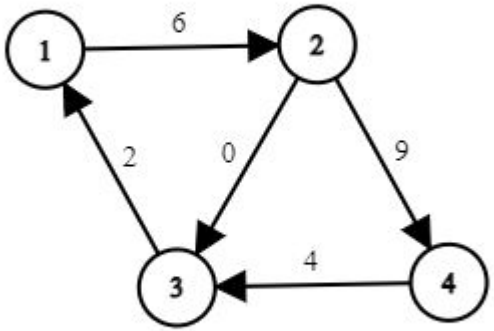


Edge List (Cont.)

- Weighted edge list
  - List of triplets (i, j, k) that denote an edge between vertex i and vertex j with weight k
- If directed, we can say that the edge [i, j] goes from i to j. Otherwise, we can just say that either way works, and have our code check both directions.
- This is the most common way you'll be given the graph!

```
4    // total vertices (1, 2, 3, 4)
5    // total edges
1 2 6
2 4 9
4 3 4
3 1 2
2 3 0
```

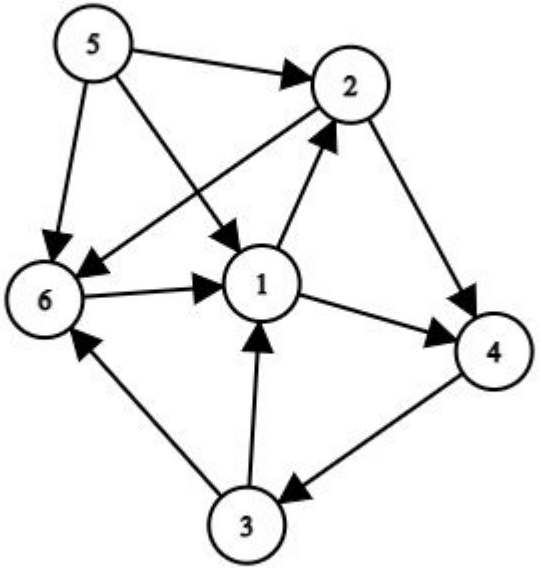
```
[
    [1, 2, 6],
    [2, 4, 9],
    [4, 3, 4],
    [3, 1, 2],
    [2, 3, 0]
]
```



Adjacency List

- List of lists that denotes edges from vertex i to all vertices in the i^{th} list
- Source vertices in fixed order
- Pros
  - Good for most graphs
  - Fast to iterate over neighbors
  - Fast edge insertions
- Cons
  - Bad for dense graphs, but better than edge list
  - Slow to perform edge queries

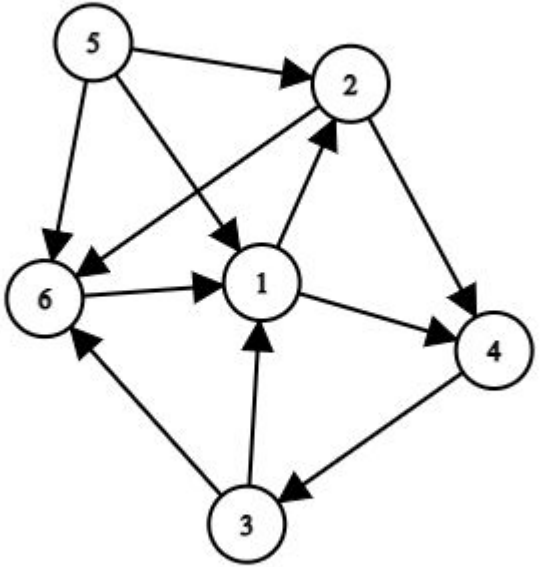
```
6    // total vertices (1, 2, 3, 4, 5, 6)
2 2 2 1 3 1 // length of each list
2 4
4 6
1 6
3
1 2 6
1
```




```
[
    [2, 4]
    [4, 6]
    [1, 6]
    [3]
    [1, 2, 6]
    [1]
]
```

Adjacency List Variations

- Weighted Adjacency List
  - Store a second adjacency list of edge weights for each edge
  - Alternatively, store pairs of destination/weight instead of just destination
- Adjacency Set
  - Use hash-based or tree-based sets for destination vertex instead of lists
  - Higher overhead for all operations, but faster edge queries



Edge List to Adjacency List

```python
n = int(input())
m = int(input())

# make a list of n empty lists
adj = [[] for _ in range(n + 1)]

# read edges
for _ in range(m):
  uv = input().split()
  u, v = int(uv[0]), int(uv[1])
  # add edge from u to v
  adj[u].append(v)
```

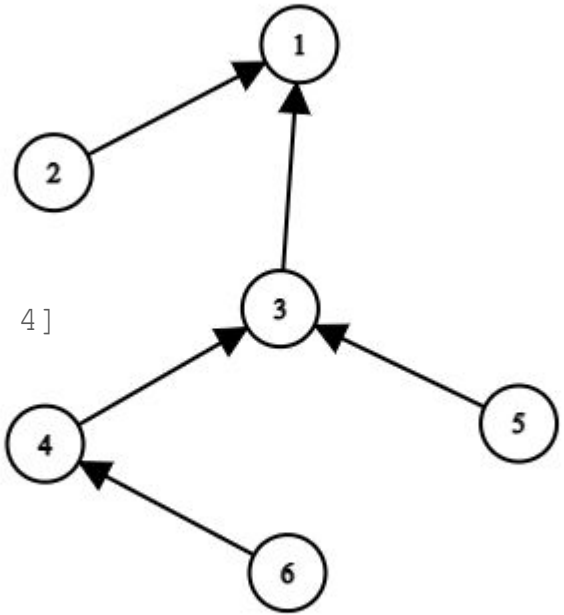
```
4    // total vertices (1, 2, 3, 4)
5    // total edges
1 2
2 4
4 3
3 1
2 3



adj = [
      []
      [2],
      [4, 3],
      [1],
      [3]
]
```

Adjacency Array

- Array that denotes edge from i to the element at index i
- Every vertex must have an outgoing edge
- Edges given in fixed order
- Pros
  - Great for trees and sparse graphs
  - Simple to read and write
  - Very memory efficient
  - Very fast for all operations
- Cons
  - Each vertex must have exactly 1 outgoing edge
- To add weights, just use a second array!

```
6    // total vertices (1, 2, 3, 4, 5, 6)
0 1 1 3 3 4
```

```
[0, 1, 1, 3, 3, 4]
```

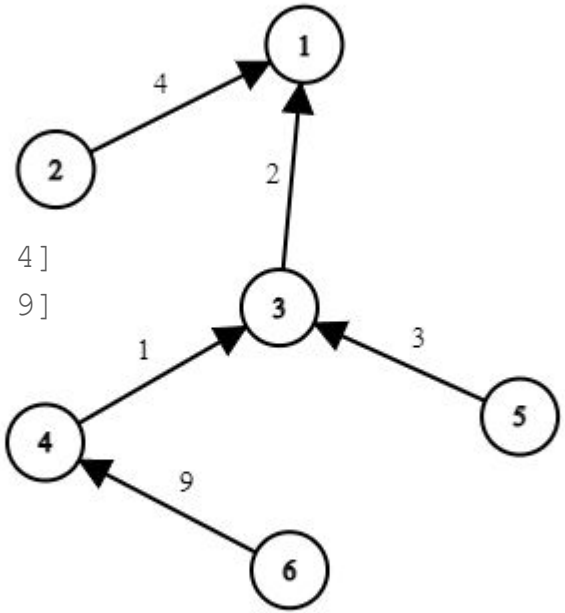


Adjacency Array (Cont.)

- Weighted adjacency array
  - Array that denotes edge from i to the element at index i
  - Second array that denotes weight of edge from index i
- Very clean way to represent trees

```
6    // total vertices (1, 2, 3, 4, 5, 6)
1 1 1 3 3 4
0 4 2 1 3 9
```

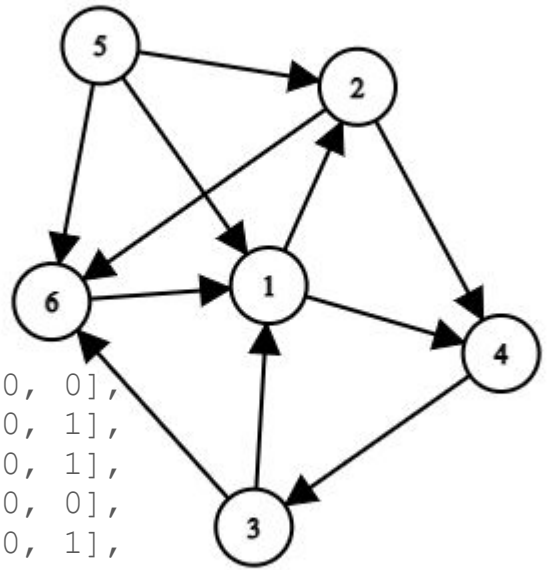
```
[1, 1, 1, 3, 3, 4]
[0, 4, 2, 1, 3, 9]
```



Adjacency Matrix

- Matrix where element in the ith row and jth index denotes the presence or absence of an edge between vertex i and vertex j
- Pros
  - Great for dense graphs
  - Fast edge queries
  - Fast edge insertions
- Cons
  - Slow to iterate over neighbors
  - Terrible for sparse graphs
  - Fixed worst case memory

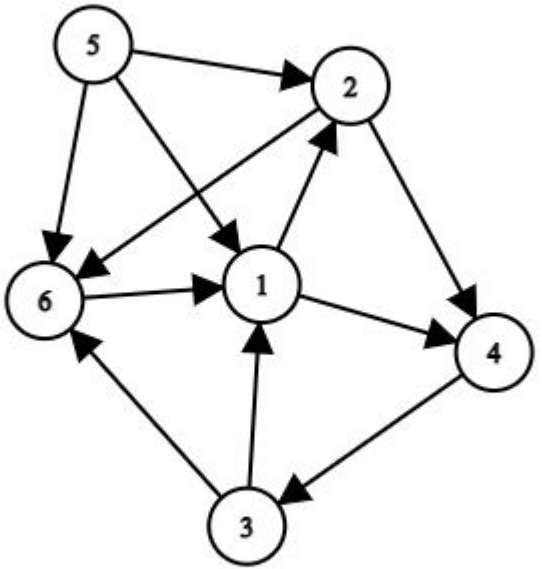
```
6    // total vertices (1, 2, 3, 4, 5, 6)
2 2 2 1 3 1 // length of each list
010100
000101
100001
001000
110001
100000
```




```
[
      [0, 1, 0, 1, 0, 0],
      [0, 0, 0, 1, 0, 1],
      [1, 0, 0, 0, 0, 1],
      [0, 0, 1, 0, 0, 0],
      [1, 1, 0, 0, 0, 1],
      [1, 0, 0, 0, 0, 0]
]
```

Adjacency Matrix (Cont.)

- Weighted Adjacency Matrix
  - Use integers or null or -1 instead of 0 and 1
- Store each row as integers/bitsets
  - Use bitwise operations to perform multiple reads/writes in a single operation
  - Potentially massive speedup!
- Usually problems that use adjacency matrices are uncommon, but still good to know



Other representations

- Sometimes, certain types of graphs may have other representations!
-

Example Problems

1. Types of Graphs
2. Graph Representations
3. **Example Problems**

Motivating Problem

- https://codeforces.com/problemset/problem/580/C

Kefa decided to celebrate his first big salary by going to the restaurant.

He lives by an unusual park. The park is a rooted tree consisting of n vertices with the root at vertex 1. Vertex 1 also contains Kefa's house. Unfortunaely for our hero, the park also contains cats. Kefa has already found out what are the vertices with cats in them.

The leaf vertices of the park contain restaurants. Kefa wants to choose a restaurant where he will go, but unfortunately he is very afraid of cats, so there is no way he will go to the restaurant if the path from the restaurant to his house contains more than m **consecutive** vertices with cats.

Your task is to help Kefa count the number of restaurants where he can go.

Input

The first line contains two integers, n and m ($2 \leq n \leq 10^5$, $1 \leq m \leq n$) — the number of vertices of the tree and the maximum number of consecutive vertices with cats that is still ok for Kefa.

The second line contains n integers $a_1, a_2, \ldots, a_n$, where each a_i either equals to 0 (then vertex i has no cat), or equals to 1 (then vertex i has a cat).

Next $n - 1$ lines contains the edges of the tree in the format "$x_i\ y_i$" (without the quotes) ($1 \leq x_i, y_i \leq n$, $x_i \neq y_i$), where x_i and y_i are the vertices of the tree, connected by an edge.

It is guaranteed that the given set of edges specifies a tree.

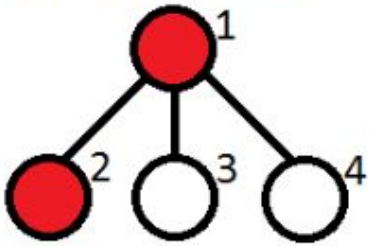
Output

A single integer — the number of distinct leaves of a tree the path to which from Kefa's home contains at most m consecutive vertices with cats.

Note

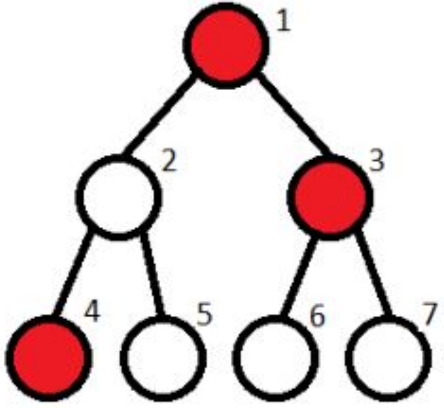
Let us remind you that a *tree* is a connected graph on n vertices and $n - 1$ edge. A *rooted* tree is a tree with a special vertex called *root*. In a rooted tree among any two vertices connected by an edge, one vertex is a parent (the one closer to the root), and the other one is a child. A vertex is called a *leaf*, if it has no children.

Note to the first sample test:



The vertices containing cats are marked red. The restaurants are at vertices 2, 3, 4. Kefa can't go only to the restaurant located at vertex 2.

Note to the second sample test:



The restaurants are located at vertices 4, 5, 6, 7. Kefa can't go to restaurants 6, 7.

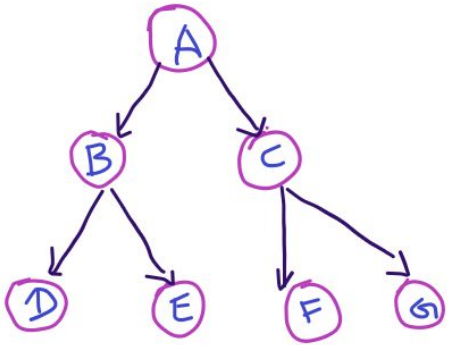
Depth First Search (DFS)

- An easy-to-implement graph traversal method
- “Depth-first” means that it will continue down a path until there is nothing left
    - Then it will backtrack

Depth First Search (DFS)

- An easy-to-implement graph traversal method
- “Depth-first” means that it will continue down a path until there is nothing left
  - Then it will backtrack
- Different from breadth first search (BFS) which traverses in level-order
- Start at the root, then do the following:
  - If we're looking at a leaf, stop
  - Otherwise, recursively call DFS on each of our children

Example

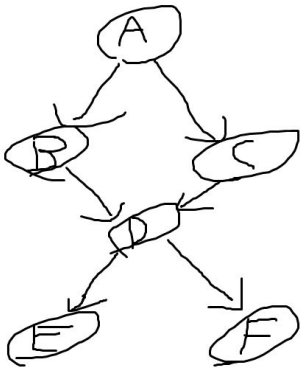


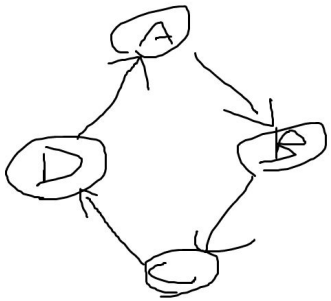
BFS - ABCDEFG

DFS - ABDECFG

One Last Thing!

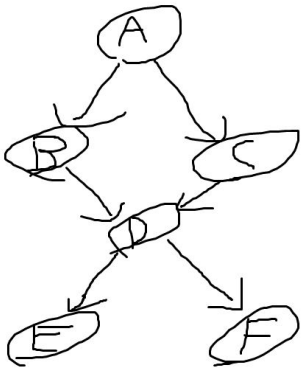
- What happens if there are two ways to get to a vertex?
- What if there's a cycle?
- Fortunately this question guarantees that we have a tree, so the above two statements won't be true. But what if we're given a general graph instead?

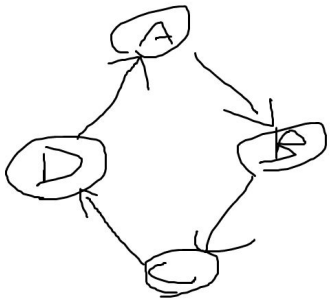




One Last Thing!

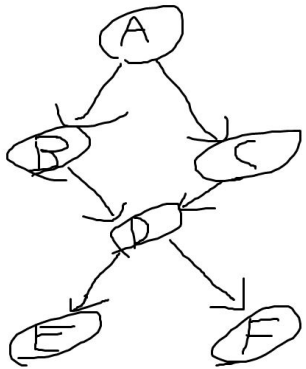
- What happens if there are two ways to get to a vertex?
  - We'll call DFS multiple times on that same vertex!
- What if there's a cycle?
  - We'll call DFS infinitely!
- We need to make sure that two adjacent vertices don't infinitely make DFS calls to each other!

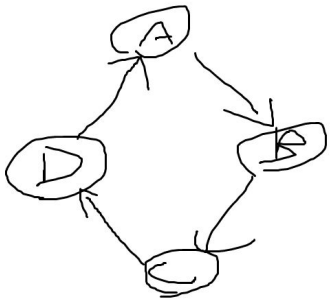




One Last Thing!

- We need to make sure that two adjacent vertices don't infinitely make DFS calls to each other!
- One way to do this is to maintain a boolean list called ***vis***
  - vis_i **= 1** if vertex ***i*** has already been visited
  - vis_i **= 0** if vertex ***i*** is unvisited
- Then, we *only* run DFS if we haven't visited it already! This guarantees we run DFS exactly once on each vertex.
- Works on all graphs (not just trees)





Implementation

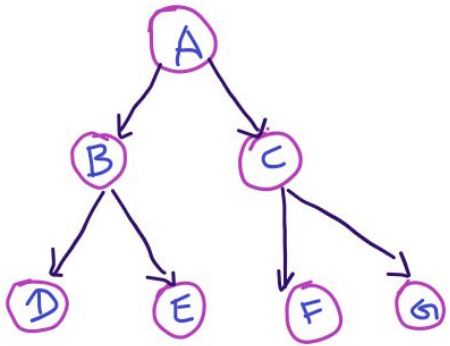
```
def dfs(curr):
    visited[curr] = 1
    for child in adjacency_list[curr]:
        if not visited[child]:
            dfs(child)
```

Tree DFS

- The visited list is not necessary if we're only working with a tree
    - Trees contain no cycles therefore there is a unique path from the root to each vertex

Tree DFS

- The visited list is not necessary if we're only working with a tree
    - Trees contain no cycles therefore there is a unique path from the root to each vertex
- Instead, it is enough to check that the vertex we are trying to visit is not our parent node



Tree DFS

- The visited list is not necessary if we're only working with a tree
  - Trees contain no cycles therefore there is a unique path from the root to each vertex
- Instead, it is enough to check that the vertex we are trying to visit is not our parent node

```
def dfs(curr, parent):
    for child in adjacency_list[curr]:
        if child != parent:
            dfs(child, curr)
```

Kefa decided to celebrate his first big salary by going to the restaurant.

He lives by an unusual park. The park is a rooted tree consisting of n vertices with the root at vertex 1. Vertex 1 also contains Kefa's house. Unfortunaely for our hero, the park also contains cats. Kefa has already found out what are the vertices with cats in them.

The leaf vertices of the park contain restaurants. Kefa wants to choose a restaurant where he will go, but unfortunately he is very afraid of cats, so there is no way he will go to the restaurant if the path from the restaurant to his house contains more than m **consecutive** vertices with cats.

Your task is to help Kefa count the number of restaurants where he can go.

Input

The first line contains two integers, n and m ($2 \leq n \leq 10^5$, $1 \leq m \leq n$) — the number of vertices of the tree and the maximum number of consecutive vertices with cats that is still ok for Kefa.

The second line contains n integers $a_1, a_2, \ldots, a_n$, where each a_i either equals to 0 (then vertex i has no cat), or equals to 1 (then vertex i has a cat).

Next $n - 1$ lines contains the edges of the tree in the format "$x_i\ y_i$" (without the quotes) ($1 \leq x_i, y_i \leq n$, $x_i \neq y_i$), where x_i and y_i are the vertices of the tree, connected by an edge.

It is guaranteed that the given set of edges specifies a tree.

Output

A single integer — the number of distinct leaves of a tree the path to which from Kefa's home contains at most m consecutive vertices with cats.

Solution

- Let’s call a vertex with cats a “***bad***” vertex
- We need to keep track of how many consecutive ***bad*** vertices we’ve visited

Solution

- Let's call a vertex with cats a "***bad***" vertex
- We need to keep track of how many consecutive ***bad*** vertices we've visited
- Add a third parameter to our dfs with the current count of ***bad*** vertices
- If ***count > m***, then we return from our dfs
    - We aren't allowed to use this path

Solution

- Right when we visit a vertex, we check if it is ***bad*** or not
- If it's bad:
  - Increment our ***count*** by one
- Otherwise
  - Reset the ***count*** to zero!

Check For Leaf Node?

- We need to see if the current vertex is a leaf so that we can increment our answer!
- A leaf node will not make any dfs calls
    - So if we did not make any other dfs calls from this vertex, we are at a leaf

```cpp
int main(){
    int n, m;
    cin>>n>>m;
    vector<vector<int>> adj(n+1);
    vector<int> cats(n+1), vis(n+1);
    for(int i = 1; i <= n; i++){
        cin>>cats[i];
    }
    for(int i = 0; i < n-1; i++){
        int a, b;
        cin>>a>>b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    ll ans = 0;
    function<void(int, int, int)> dfs = [&](int u, int cnt, int par){
        if(cats[u]){
            cnt++;
        }
        else
            cnt = 0;
        if(cnt > m)
            return;
        bool isLeaf = 1;
        for(int c: adj[u]){
            if(c != par){
                isLeaf = 0;
                dfs(c, cnt, u);
            }
        }
        if(isLeaf){
            ans++;
        }
    };
    dfs(1, 0, -1);
    cout<<ans<<endl;
    return 0;
}
```

Problem B

- https://codeforces.com/contest/1365/problem/D

Vivek has encountered a problem. He has a maze that can be represented as an $n \times m$ grid. Each of the grid cells may represent the following:

- Empty — '`.`'
- Wall — '`#`'
- Good person — '`G`'
- Bad person — '`B`'

The only escape from the maze is at cell (n, m).

A person can move to a cell only if it shares a side with their current cell and does not contain a wall. Vivek wants to block some of the empty cells by replacing them with walls in such a way, that all the good people are able to escape, while none of the bad people are able to. A cell that initially contains '`G`' or '`B`' **cannot be blocked** and **can be travelled through**.

Help him determine if there exists a way to replace some (zero or more) empty cells with walls to satisfy the above conditions.

It is guaranteed that the cell (n, m) is empty. Vivek can also block this cell.

Input

The first line contains one integer t $(1 \le t \le 100)$ — the number of test cases. The description of the test cases follows.

The first line of each test case contains two integers n, m $(1 \le n, m \le 50)$ — the number of rows and columns in the maze.

Each of the next n lines contain m characters. They describe the layout of the maze. If a character on a line equals '`.`', the corresponding cell is empty. If it equals '`#`', the cell has a wall. '`G`' corresponds to a good person and '`B`' corresponds to a bad person.

Output

For each test case, print "`Yes`" if there exists a way to replace some empty cells with walls to satisfy the given conditions. Otherwise print "`No`"

You may print every letter in any case (upper or lower).

Example

input

```
6
1 1
.
1 2
G.
2 2
#B
G.
2 3
G.#
B#.
3 3
#B.
#..
GG.
2 2
#B
B.
```

output

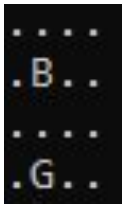
```
Yes
Yes
No
No
Yes
Yes
```

Solve The Maze Solution

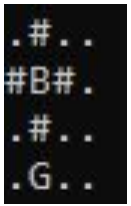
- Let’s first prevent all “bad people” from making it to the end
- What is the easiest way to guarantee that they cannot escape?
    - While also minimizing the chance that it prevents any “good people” from escaping?

Solve The Maze Solution

- Let's first prevent all "bad people" from making it to the end
- What is the easiest way to guarantee that they cannot escape?
    - While also minimizing the chance that it prevents any "good people" from escaping?
- We "box in" all bad people
- Let's say your grid is this:



Boxing a bad person would look like this:



Solve The Maze Solution

- Now we just have to make sure all the good people can still escape!

Solve The Maze Solution

- Now we just have to make sure all the good people can still escape!
- What if we dfs for each good person if they can reach the exit?

Solve The Maze Solution

- Now we just have to make sure all the good people can still escape!
- What if we dfs for each good person if they can reach the exit?
    - Too slow! In the worst case we have ***n*** * ***m*** people each taking a path order ***n*** * ***m***
    - In total: this would give us about 625 million operations

Solve The Maze Solution

- Now we just have to make sure all the good people can still escape!
- What if we dfs for each good person if they can reach the exit?
  - Too slow! In the worst case we have ***n*** * ***m*** people each taking a path order ***n*** * ***m***
  - In total: this would give us about 625 million operations
- Instead we do ***one*** dfs from the escape cell and then see if all good people become visited!
- The single dfs is O(***nm***)

Solve The Maze Solution

- Tricky Edge Case!
  - The goal itself can be blocked!
- Only do the dfs from the escape cell if it's not blocked!
  - Otherwise if it is blocked, then it is not possible for everyone to escape, output "no"

CSES: Building Roads

Byteland has n cities, and m roads between them. The goal is to construct new roads so that there is a route between any two cities.

Your task is to find out the minimum number of roads required, and also determine which roads should be built.

Input

The first input line has two integers n and m: the number of cities and roads. The cities are numbered $1, 2, \ldots, n$.

After that, there are m lines describing the roads. Each line has two integers a and b: there is a road between those cities.

A road always connects two different cities, and there is at most one road between any two cities.

Output

First print an integer k: the number of required roads.

Then, print k lines that describe the new roads. You can print any valid solution.

Constraints

- $1 \leq n \leq 10^5$
- $1 \leq m \leq 2 \cdot 10^5$
- $1 \leq a, b \leq n$

Example

Input:
```
4 2
1 2
3 4
```

Output:
```
1
2 3
```

CALSUCO '22: CALICO's Corporate Conundrum

Problem Statement

Find the largest number of unique disputes a single person is responsible for resolving within the company given the managers of all $\mathbf{N}$ of its employees. Each employee is identified using a number counting upwards from 0, with the last employee being assigned the number $\mathbf{N}$ - 1. The ith employee's manager $\mathbf{M}_i$ denotes the employee number of their direct superior. The CEO is assigned employee number 0 and is the only employee whose direct superior is themselves.

A dispute between two employees is resolved by the lowest-level manager that has authority over both of them, whether directly or indirectly. Any employee also has authority over themselves, meaning it's possible for a dispute to be resolved by someone involved in it. Two disputes are unique if they involve at least one different employee.

Input Format

The first line of the input contains a positive integer $\mathbf{T}$ denoting the number of test cases that follow. For each test case:

- The first line contains the single positive integer $\mathbf{N}$ denoting the number of employees present in the company.
- The second line contains the space-separated sequence of $\mathbf{N}$ non-negative integers $\mathbf{M}_0, \mathbf{M}_1, \ldots, \mathbf{M}_{N-1}$ denoting the employee number of each individual's manager.
  - $\mathbf{M}_0$ will always be zero to represent the CEO being their own direct superior.

Output Format

For each test case, output a single line containing the largest number of unique disputes a single person is responsible for resolving within the company.

CALSUCO '22: CALICO's Corporate Conundrum

Sample Test Cases

Main Sample Input

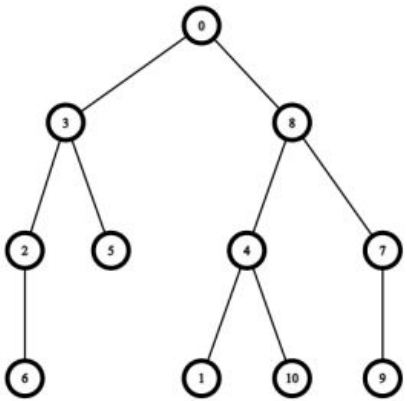
```
3
5
0 0 0 1 1
11
0 4 3 0 8 3 2 8 0 7 4
16
0 0 1 1 2 2 3 4 4 5 5 7 8 9 9 10
6
0 0 1 2 3 4
```

Main Sample Output

```
7
34
41
5
```

For Test Case #2:

The company structure looks like this:



The number of unique disputes each employee is responsible for is as follows:

- Employee 0: 34 disputes
- Employee 1: 0 disputes
- Employee 2: 1 dispute
- Employee 3: 5 disputes
- Employee 4: 3 disputes
- Employee 5: 0 disputes
- Employee 6: 0 disputes
- Employee 7: 1 dispute
- Employee 8: 11 disputes
- Employee 9: 0 disputes
- Employee 10: 0 disputes

The maximum number of disputes a single employee is responsible for is 34 (employee 0).

Atcoder ABC258: Triangle

Problem Statement

You are given a simple undirected graph G with N vertices.

G is given as the $N \times N$ adjacency matrix A. That is, there is an edge between Vertices i and j if $A_{i,j}$ is 1, and there is not if $A_{i,j}$ is 0.

Find the number of triples of integers (i, j, k) satisfying $1 \leq i < j < k \leq N$ such that there is an edge between Vertices i and j, an edge between Vertices j and k, and an edge between Vertices i and k.

Constraints

- $3 \leq N \leq 3000$
- A is the adjacency matrix of a simple undirected graph G.
- All values in input are integers.

Input

Input is given from Standard Input in the following format:

$$
\begin{aligned}
&N \\
&A_{1,1}A_{1,2}\ldots A_{1,N} \\
&A_{2,1}A_{2,2}\ldots A_{2,N} \\
&\vdots \\
&A_{N,1}A_{N,2}\ldots A_{N,N}
\end{aligned}
$$

Output

Print the answer.