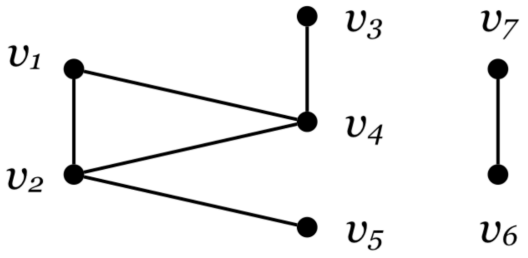
CS 280
Spring 2018

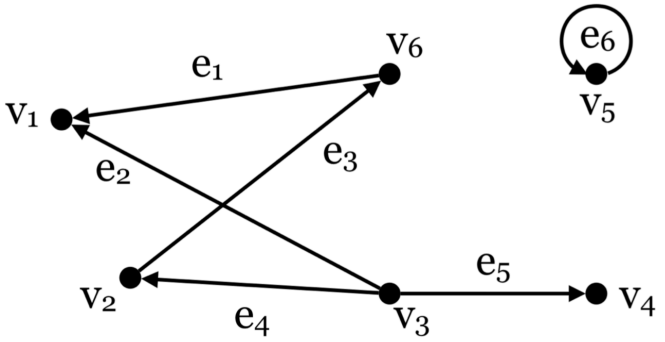
John Bowers, Professor
Mike Lam, Professor

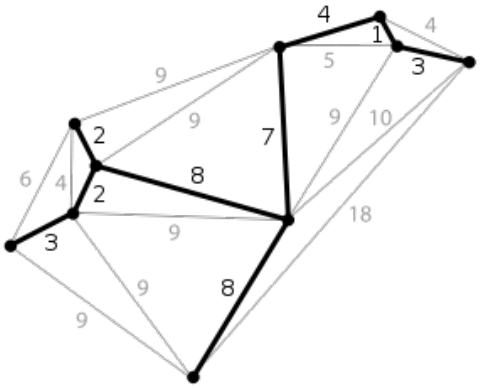
Graphs

Non-linear data structures

- Graph
  - Collection of nodes (vertices) and links (edges)
  - Directed vs. undirected
  - Weighted vs. unweighted
  - Connected vs. not connected





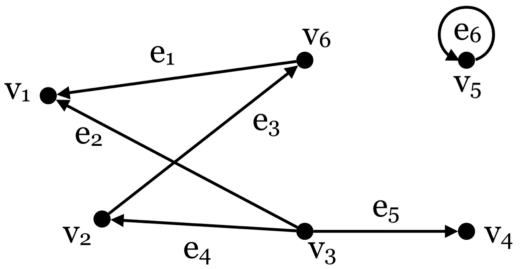


Graph implementations

- **Adjacency list**
  - Store a list of neighbors for each vertex.
- **Adjacency matrices**
  - Store a 2D matrix, where the (i, j)-entry is a 1 if there is an edge from vertex i to vertex j, or 0 otherwise.
- **Weight matrices**
  - Like an adjacency matrix, but the (i,j)-entry is the weight of the edge from i to j or infinity (Float.Positive_Infinity) if one doesn't exist.

Graph implementations

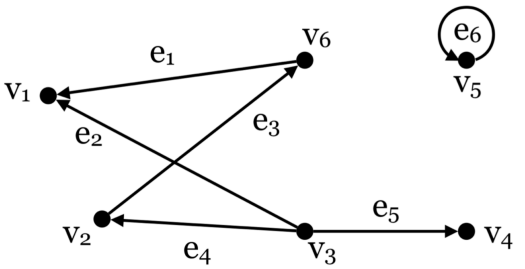
- **Adjacency list (generally more useful)**
  Space: O(V + E).
  Large graphs. Sparse graphs.




```
List<List<Integer>> adjList = new ArrayList<>();
for (int i = 0; i <= 6; i++) {
    adjList.add(new ArrayList<Integer>());
}
verts.get(6).add(1); // e1
verts.get(3).add(1); // e2
verts.get(2).add(6); // e3
verts.get(3).add(2); // e4
verts.get(3).add(4); // e5
verts.get(5).add(5); // e6
```

Graph implementations

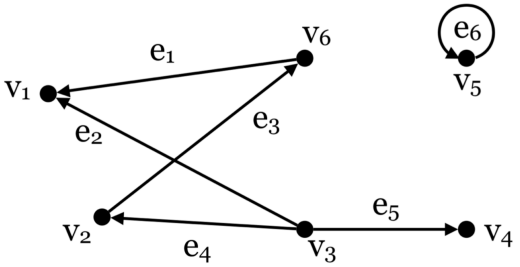
- **Adjacency matrix**
  Space: $O(V^2)$
  Good for small, dense graphs. V<1000.




```
int[][] adjMatrix = new int[7][7];
adjMatrix[6][1] = 1; // e1
adjMatrix[3][1] = 1; // e2
adjMatrix[2][6] = 1; // e3
adjMatrix[3][2] = 1; // e4
adjMatrix[3][4] = 1; // e5
adjMatrix[5][5] = 1; // e6
```

Graph implementations

- **Weight matrix**
  Adjacency matrix with weights.

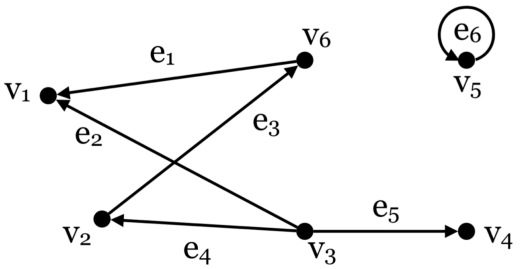



```
float[][] W = new float[7][7];
for (int i = 0; i < 7; i++) {
    for (int j = 0; j < 7; j++) {
        W[i][j] = Float.POSITIVE_INFINITY;
    }
}
W[6][1] = 1.5f; // e1
W[3][1] = 2.0f; // e2
W[2][6] = 1.f;  // e3
W[3][2] = -3.f; // e4
W[3][4] = 1.7f; // e5
W[5][5] = 9.0f; // e6
```

Weights:
e1: 1.5
e2: 2.0
e3: 1
e4: -3
e5: 1.7
e6: 9

Graph implementations

- **Flat Edge list.**
  Space: O(E)
  Sorted list of edges is useful for some algorithms.

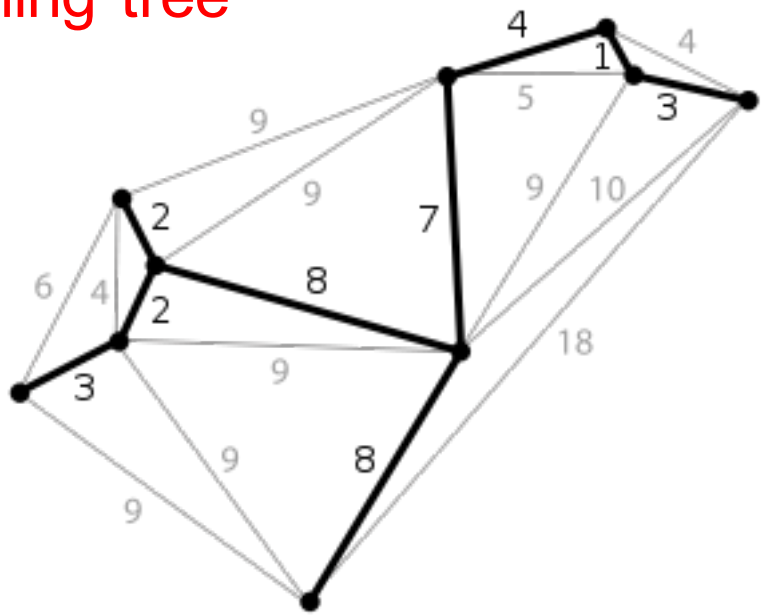



```
static class Edge implements Comparable<Edge>{
    int i, j;
    Float w;
    public Edge(int i, int j, float w) {this.i = i; this.j = j; this.w = w;}
    public int compareTo(Edge e) {
        return w.compareTo(e.w);
    }
}
public static void main(String[] args) {
    List<Edge> edges = new ArrayList<>();
    edges.add(new Edge(6,1,1.5f));
    edges.add(new Edge(3,1,2.f));
    edges.add(new Edge(2,6,1.f));
    edges.add(new Edge(3,2,-3.f));
    edges.add(new Edge(3,4,1.7f));
    edges.add(new Edge(5,5,9.0f));
    Collections.sort(edges);
}
```

Weights:
e1: 1.5
e2: 2.0
e3: 1
e4: -3
e5: 1.7
e6: 9

Specific kinds of graphs

- Simple graphs
- Fully-connected (complete) graphs
- Trees
- (Minimum) spanning tree



Takeaway

- No built-in structures
  - Be familiar with the different types of graphs
- Some problems may require you to store a graph
  - And some will timeout if you do!
    - Some graphs are infinitely large…

In-class problem

Flying Safely

Due to budget cuts, even spies have to use commercial airlines nowadays to travel between cities in the world. Although this mode of travel can be very convenient for a spy, it also raises a problem: the spy has to trust the pilot to make sure he is not in danger during the flight. And even worse, sometimes there is no direct flight between some pairs of cities, so that the spy has to take multiple flights to get to the desired location, and thus has to trust multiple pilots!

To limit the trust issues you are asked for help. Given the flight schedule, figure out the smallest set of pilots that need to be trusted, such that the spy can safely travel between all cities.

Input

On the first line one positive number: the number of test cases, at most 100. After that per test case:

- one line with two space-separated integers n ($2 \leq n \leq 1\,000$) and m ($1 \leq m \leq 10\,000$): the number of cities and the number of pilots, respectively.
- m lines with two space-separated integers a and b ($1 \leq a, b \leq n, a \neq b$): a pilot flying his plane back and forth between city a and b.

It is possible to go from any city to any other city using one or more flights. In other words: the graph is connected.

In-class problem

Input

On the first line one positive number: the number of test cases, at most 100. After that per test case:

- one line with two space-separated integers n ($2 \leq n \leq 1\,000$) and m ($1 \leq m \leq 10\,000$): the number of cities and the number of pilots, respectively.
- m lines with two space-separated integers a and b ($1 \leq a, b \leq n, a \neq b$): a pilot flying his plane back and forth between city a and b.

It is possible to go from any city to any other city using one or more flights. In other words: the graph is connected.

Output

Per test case:

- one line with one integer: the minimum number of pilots that need to be trusted such that it is possible to travel between each pair of cities.

Sample Input 1

```
2
3 3
1 2
2 3
1 3
5 4
2 1
2 3
4 3
4 5
```

Sample Output 1

```
2
4
```

Pseudocode

```
cases = int(input())

for case in range(cases):
    n, m = input().split()
    print(int(n)-1)
    remaining = [input() for line in range(int(m))]
```