NOI.PH Training: Graphs 1

Definitions and Basic Algorithms

Carl Joshua Quines

Contents

1 Introduction

There are lots of scenarios where you have a bunch of data, and a bunch of relationships between them. For example:

- There may be *cities*, and there may be *roads* connecting these cities.
- There are *courses* you need to take, but some courses have *prerequisites* which you need to take before you can take that course.
- There are *people* in a social networking site called Bacefook, and pairs of people have varying *relationships* (including "none").

There are many more examples, but the overarching theme is that there are bunch of *objects*, and some of those objects are *linked* together.

Such scenarios are best modeled by mathematical objects known as **graphs**. No, I don't mean the graph of an equation (like $x^{42} + y^{42} = 1$) on the plane—not that kind of graph. And actually, because of this name clash, I sometimes prefer using different names instead:

- For a graph of an equation on the plane, I would use **plot**.
- For a graph in the sense of this module, I would use **network**.

Other people sometimes use these too. But since we won't be discussing "plots" much, we'll simply use "graphs" throughout this module.

Abstractly, a graph is a mathematical object consisting of:

- objects called **nodes**, and
- pairs of nodes, called **edges**. (These are the "links" between objects.)

They're often augmented with additional data, such as the *lengths* of the edges. We'll get into these in more detail, as well as the ways to represent them on your program, and some basic algorithms on them such as *traversal*, i.e., "going through the nodes".

Why study graphs? Why do things "abstractly"? The idea is that, instead of thinking about cities and roads, or people and relationships, etc., we can strip away the unneeded parts of the scenario and focus on what really matters: the relationships between objects. **Abstraction** is the process of doing this. The idea is that graphs are applicable to so many things that by studying graphs on their own, we obtain results that automatically apply to all these things for free!

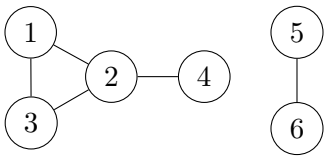
2 Definitions

Terms: graph, node, edge, loop, parallel edge, multigraph, simple

It is tradition that any introduction to graph theory begins with a bunch of definitions.[1] There are like, two dozen or so terms to define, but it's helped by the fact that most of the terms are intuitive. Also, you don't need to absorb all these terms immediately in one go; rather, you'll just come to memorize them naturally as you keep using them.

My best solution for introducing them in a memorable way is to include exercises along with the definitions, so make sure to try all the exercises before continuing reading. Doing the exercises will hopefully make the definitions come to you more easily.

This is a **graph**:



Informally, you can think of a graph as a set of **nodes** (represented as circles) that are connected by **edges** (represented as curves joining those circles). The name *graph* comes from the fact that we can represent it graphically.

The formal definition of a graph is useful because it lets us talk about graphs without drawing stuff. A graph G is an ordered pair of a set of nodes V and a set of edges E, where each edge is a set of two nodes.[2] For the above graph, we have

$$V = \{1, 2, 3, 4, 5, 6\} \qquad E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{5, 6\}\}.$$

The typical variables we'll use to represent nodes are u, v, and w. The edge $\{u, v\}$ is typically written as uv. So the edges uv and vu represent the same edge.

Example 2.1. Consider **Spratly Islands Tour:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/spratly-islands-tour. There are N islands arranged from north to south, and each island has a two-way bridge joining it to an island further north, and a two-way bridge joining it to an island further south.

This is a pretty blatant example of a graph problem: the islands are nodes, and the bridges are edges. Other similar ones are districts and roads, cities and airplane flights, bus stations and bus lines.

Exercise 2.2. Consider **Mutual Friendzone:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/mutual-friendzone-not-hacked, where there are people on a social networking site, and some pairs of these people are friends with each other. This network is naturally represented by a graph. What are its nodes and edges?

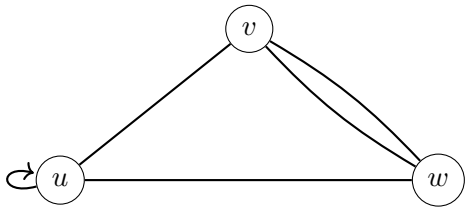
Example 2.3. Sometimes graphs can come up in unexpected situations. A classic example is the *wolf, goat, and cabbage* puzzle. A man wishes to cross a river and carry a wolf, a goat, and a cabbage. His boat can only carry himself and one of these items at a single time. He cannot

[1] By "tradition", I mean every graph theory book I've read begins with these definitions as the first section.
[2] The letter "V" was chosen because **nodes** are alternatively called **vertices**.

leave the wolf alone with the goat, or the goat alone with the cabbage. How can he do this?

The organized way to solve this is to represent the puzzle using a graph. Let the nodes denote which items are on which sides, like {w, g}, {c} to represent the wolf and goat on the first side, and the cabbage on the second side. Then draw an edge between two nodes if the man can use his boat to transfer between them. Can you use this graph to solve the problem?

Right now, our formal definition exclude graphs like these:



Node u has an edge joining it to itself, which goes against an edge being a set of two nodes. These kinds of edges are called **loops**. There are multiple edges joining v and w, which goes against the fact that the edges form a set. These kinds of edges are **parallel edges** or multiedges.

In a **multigraph** or complex graph, both loops and parallel edges are allowed.[3] Graphs where these aren't allowed are called **simple graphs**. Unless stated otherwise, the term *graph* will refer to both multigraphs and simple graphs. (Be careful with this: many authors use *graph* only for simple graphs.)

Example 2.4. Read the statement of **Lito Lapida's Lost Lapida:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/lito-lapida. The constraint $a \neq b$ means that the graph doesn't have any loops, but nowhere was it stated the graph can't have parallel edges. In fact, the sample input shows a graph that *does* have multiple edges. So the graphs in the problem are multigraphs.

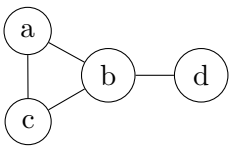
Unless otherwise stated, **do not assume that a graph problem only deals with simple graphs**. If in doubt, ask.

Example 2.5. Consider **City Map:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/city-map. The condition "either connected by one bridge, or no bridge at all" means that parallel edges aren't allowed. "No bridge connects a house to itself" means that loops aren't allowed. So all the graphs in this problem are simple graphs.

2.1 Degree

Terms: adjacent, neighbor, incident, degree ($\deg(v)$)

Consider this graph:



[3] This amounts to changing all the sets in the formal definition to multisets. A multiset is a set where repeated elements are allowed.

Two nodes joined by an edge are called **adjacent**. Here, nodes a and b are adjacent. An adjacent node is also called a **neighbor**, so the neighbors of c are a and b. An edge is **incident** to each of the nodes is joins. Here, node b is incident to edge bd.

The number of edges incident to a node v is called its **degree**, which we'll write as $\deg(v)$. Node b is incident to ab, bc, and bd, so $\deg(b) = 3$. In a multigraph, loops are counted twice. For a simple graph, the degree of a node is also the number of its neighbors.

We now have enough terms to state and prove our first theorem:

Theorem 2.6 (Handshaking lemma)**.** The sum of degrees of each node is even.

Proof. Since each edge has two endpoints, the sum of degrees is exactly twice the number of edges:

$$\sum_{v \in V} \deg(v) = 2|E|.$$

In particular, this sum is even. □

This theorem is equivalent to the fact that there are an even number of nodes with odd degree. This theorem is also one of the reasons why we want to count loops twice when defining the degree of a node.

Exercise 2.7. Show that any graph has an even number of nodes with odd degree.

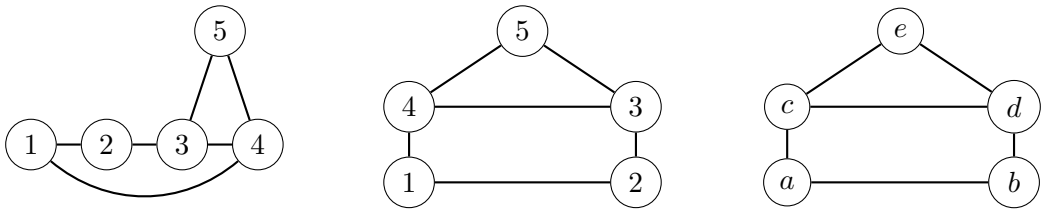
Exercise 2.8. A graph has 10 nodes and 13 edges. What is the average (mean) degree of the nodes?

Exercise 2.9. Prove that any simple graph with at least two nodes has two nodes of the same degree.

2.2 Isomorphism

Terms: isomorphic, complement

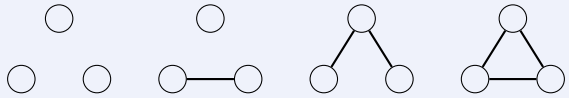
A graph can be drawn differently but still be the same graph. Consider these three graphs:



The two graphs on the left share the same set of nodes and the same set of edges, the layout is just different. They are the same graph. The two graphs on the right share a different set of nodes, but if we make the changes $1 \to a, 2 \to b, 3 \to d, 4 \to c, 5 \to e$, then we can see that the same edges connect the same nodes. They are also the same graph.

So all three graphs are **isomorphic** to each other. You can remember this word through the prefix *iso*, meaning *same*, and the root word *morph*, meaning *form*. Isomorphic graphs have the same form. For much of this module, we will treat graphs that are isomorphic as the same.

Example 2.10. There are 4 different simple graphs with 3 nodes:



If we wanted to be more specific, we'd say "4 *nonisomorphic* simple graphs".

Exercise 2.11. There are two different simple graphs with 4 nodes and 3 edges. Draw them.

Exercise 2.12. Convince yourself some two of these graphs are isomorphic:

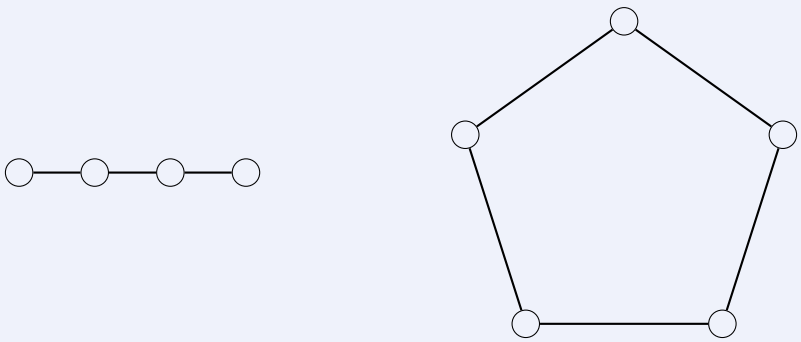
[missingfig: http://mathworld.wolfram.com/images/eps-gif/PetersenGraphEmbeddings_1000.gif]

In fact, all of these graphs are isomorphic. This graph is named the **Petersen graph**. The drawing to the left is the most common one. As Knuth said, it "serves as a counterexample to many optimistic predictions about what might be true for graphs in general."

The **complement** $\overline{G}$ of a simple graph G is the simple graph with the same nodes, and two nodes are adjacent in $\overline{G}$ if and only if they are not adjacent in G.

Example 2.13. These two graphs are complements of each other:

Exercise 2.14. Convince yourself that these two graphs are isomorphic to their complement:



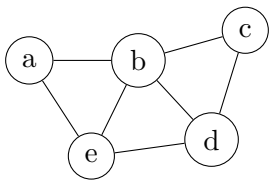
There is exactly one other simple graph with 5 nodes that is isomorphic to its complement. Find it.

2.3 Paths and cycles

Terms: walk, length, trail, path, circuit, cycle

There are six related terms in this section, but the ones most worth remembering are path, cycle, and length. To define them, however, we need to use the other terms.

Consider this graph:



A **walk** is a sequence of alternating nodes and edges $v_0, e_1, v_1, e_2, \ldots, e_\ell, v_\ell$, such that edge e_i is incident to v_{i-1} and v_i. Its **length** is ℓ, the number of edges. In the above graph, one walk is

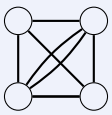
$$b, bc, c, cd, d, db, b, bd, d, de, e$$

This walk has length 5. In a simple graph, since there aren't parallel edges, we can specify a walk using a sequence of nodes instead. So the above walk could be written as *bcdbde*.

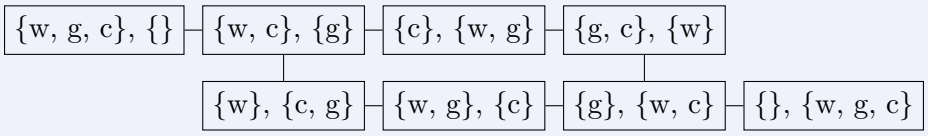
A **trail** is a walk that doesn't repeat any edges. An example of a trail is *abcdbe*; its length is 5. A **path** is a trail that doesn't repeat any nodes. An example of a path is *abcde*; its length is 4.

Exercise 2.15. Prove that if two nodes have a walk joining them, then they also have a path joining them.

Exercise 2.16. Find a trail in the following graph that passes through all of the edges:



Example 2.17. In Example 2.3, you were asked to draw a graph to represent a puzzle. You should've gotten something like this:



Solving the problem is now equivalent to finding a path from the node {w, g, c}, {} to the node {}, {w, g, c}, which we can easily see from the diagram. In fact, there are two such paths!

A **circuit** is a trail that begins and ends at the same node. So for example, *bcdbeab* is a circuit; its length is 6. A **cycle** is a circuit that doesn't repeat any nodes, except for the beginning and end node. For example, *abcdea* is a cycle; its length is 5.

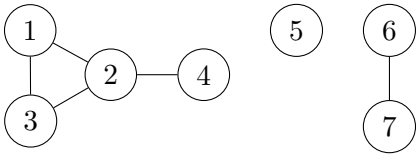
Exercise 2.18. Prove that any cycle must contain at least three nodes.

Exercise 2.19. Prove that if an edge is contained in some circuit, it's also contained in some cycle.

2.4 Connectivity

Terms: connected, disconnected, component

Consider this graph:



Two nodes are **connected** if there is some path joining them. nodes 1 and 4 are connected, while nodes 5 and 7 are **disconnected**.

Note that if u and v are connected, and v and w are connected, then u and w are connected as well. This means that we can partition the graph into several **components** such that any two nodes in the same component are connected, and any two nodes in different components are disconnected. The above graph has three components: $\{1, 2, 3, 4\}, \{5\}, \{6, 7\}$.

Example 2.20. Consider **El Filibusterismo:** https://www.hackerrank.com/contests/pset-4/challenges/el-filibusterismo. This problem is about finding the number of nodes in the component belonging to a given node.

The terms connected and disconnected can also be used to describe graphs. A graph that has multiple components is **disconnected**, and a graph that has exactly one component is **connected**. In other words, a graph is connected if we can visit any node from any other node by following a series of edges.

Example 2.21. Read the statement of **Kapuluan ng Kalayaan:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/spratly-islands. The constraint for S is that it "enables people from one island to visit any other island by taking a series of ferries." This is another way to say that S should be *connected*.

Unless otherwise stated, **do not assume that a graph problem only deals with connected graphs**. If in doubt, ask.

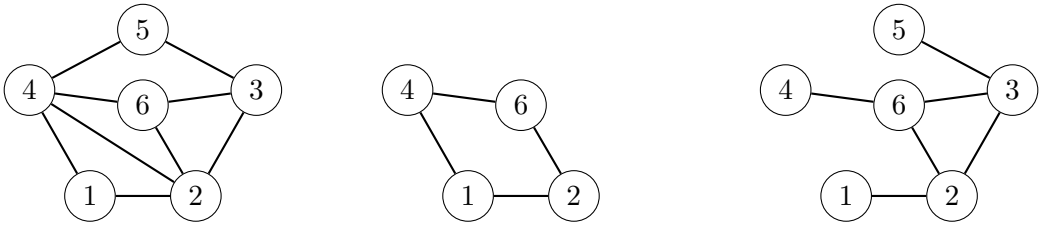
Exercise 2.22. Find a disconnected simple graph with n nodes and $\binom{n-1}{2}$ edges.

Exercise 2.23. Convince yourself that if a simple graph G is disconnected, then its complement $\overline{G}$ is connected.

2.5 Subgraphs

Terms: subgraph, contains, spanning

Consider these three graphs, and call the first graph G:



We say that a graph H is a **subgraph** of G if its nodes are a subset of G's nodes and its edges are a subset of G's edges. We also say that G **contains** H. The two graphs to the right are all subgraphs of G.

A **spanning** subgraph is one that has the same node set. The third graph above is a spanning subgraph of G.

Example 2.24. We can redefine a component as a *maximal connected subgraph*. That is, it is a subgraph where we cannot add any more nodes without it being disconnected.

Exercise 2.25. A spanning subgraph where every node has degree 1 is called a **perfect matching**, because it perfectly matches nodes to each other, leaving no node unmatched.

Consider the Petersen graph, defined in Exercise 2.12. Find any perfect matching, and remove all its edges. Convince yourself that no matter which matching you pick, you'll always end up with two disconnected cycles of length 5.

2.6 Directed graphs

Terms: arrow, directed, undirected

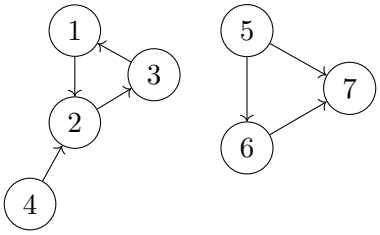
Our current conception of graphs doesn't deal with all the cases we want to use them. In particular, sometimes we want our edges to have some sort of orientation or direction:

Example 2.26. Consider **The Cheapest Reid:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/the-cheapest-reid, where the input format gives "a route from city a to city b (but not necessarily the other way around)."

This is a blatant example of a graph theory problem, where the routes should represent edges. Unlike our previous examples, where edges can be crossed in either direction, here, the edges need some notion of *direction*.

Other similar concepts are one-way roads, following someone on Twitter, or representing people who love each other. (Because loving someone doesn't mean they love you back!)

We can represent such graphs by changing our edges to **arrows**:

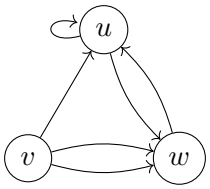


Formally, a **directed graph** (or digraph) is a set of nodes V, and a set of arrows (or directed edges) E, where each arrow is an *ordered pair* of distinct nodes. For the above graph, we have

$$V = \{1, 2, 3, 4, 5, 6, 7\} \qquad E = \{(1, 2), (2, 3), (3, 1), (4, 2), (5, 6), (6, 7), (5, 7)\}.$$

We can again represent an arrow (u, v) more succinctly with $\overrightarrow{uv}$, or simply uv. Note that uv and vu represent different arrows.

Again, the formal definition here disallows directed graphs like:



Node u has an arrow joining it to itself, which goes against an arrow being a pair of distinct nodes. These kinds of arrows are called **directed loops**. There are multiple arrows joining v and w, which goes against the fact that the arrows form a set. These kinds of arrows are called **multiple arrows**.

Note that the arrows between u and w are *not* multiple arrows. This is because the arrows uw and wu are different arrows. A **directed multigraph**, complex digraph, or multidigraph, is one where directed loops and multiple arrows are allowed; otherwise we call it a **simple directed graph**.

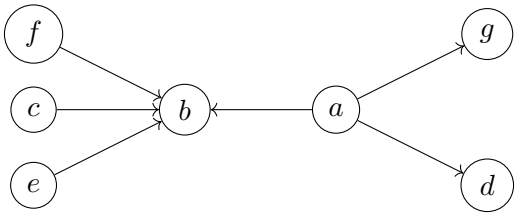
Many of the concepts from **undirected** graphs apply here too. Isomorphism and subgraphs have something similar. You have **directed paths** and **directed cycles**, which are like paths or cycles but all the arrows have to point in the same direction.

Exercise 2.27. Consider a directed graph where if uv is an arrow, then vu is also an arrow. Convince yourself that this is equivalent to an undirected graph: directed paths correspond to paths, etc.

2.6.1 Arrows

Terms: head, tail, outdegree ($\deg^+(v)$), indegree ($\deg^-(v)$)

Consider the following graph:



An arrow has two ends: its **beginning** and its **end**. The arrow uv has its beginning at u and its end at v. These are also called the source and the target, or the start and the end, or the source and the destination, or the tail and the head.

For a directed graph, we say that u and v are **adjacent** if uv is an arrow. Note that the order matters here! Here, a and b are adjacent, but b and a are not. We also say that b is adjacent to a, but a is not adjacent to b.

We can talk about a node being **incident** to either the beginning of an arrow or the end of an arrow. Here, a is incident to the beginning of ab, and b is incident to the end of ab.

The **outdegree** of a node is the number of arrows it is incident to the beginning of. It is denoted by $\deg^+(v)$, so in the above graph, $\deg^+(a) = 3$. This is because a forward arrow is like a "positive" direction from a node.[4]

The **indegree** of a node is the number of arrows it is incident to the end of. It is denoted by $\deg^-(v)$, so in the above graph, $\deg^-(b) = 2$. Again, the negative sign is because an arrow going into a node is the negative direction.

Here's the analogue of the handshake lemma:

Theorem 2.28 (Directed handshake lemma)**.** The sum of the indegrees is the same as the sum of the outdegrees.

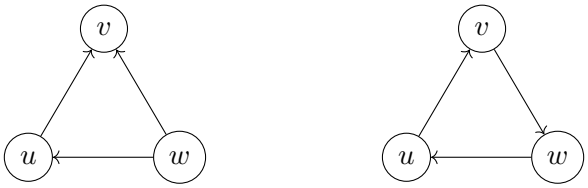
Exercise 2.29. Prove the above theorem. Both numbers are equal to $|E|$.

Exercise 2.30. A directed graph where each node v has $\deg^+(v) = 1$ is called a **functional graph**. This is because the arrows of a graph represent a function from $V \to V$.

Convince yourself that any functional graph has a directed cycle. What can you say about a directed graph where $\deg^-(v) = 1$ for each node v?

2.6.2 Connectivity

Connectivity, however, doesn't have a direct analogue. Consider the following graph:



The above shows two different directed graphs. If you take either one, and change all the arrows to undirected edges to get an undirected graph, then you get a connected graph.

But the first graph doesn't have a directed path from u to w, while the second one does. In fact, the second one has a directed path between any ordered pair of nodes!

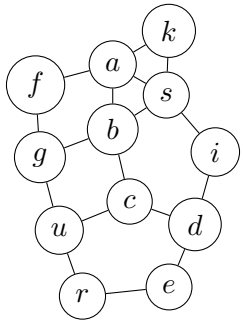
This is a problem, because suddenly, defining "connected components" doesn't work any more. This is because we have two different kinds of connectivity, and they have two different names. We'll discuss more of this in a later module on Connectivity.

[4] If you're a physics person, you can also think of how we draw the field lines *away* from a proton, which is *positively* charged.

2.7 Weighted graphs

Terms: distance ($d(v, w)$), weight, shortest path problem

The paths in a graph give us a natural notion of distance. Consider the following graph:



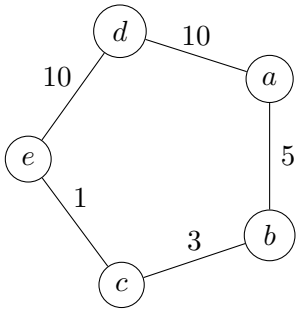
We define the **distance** between two nodes u and v to be the length of the shortest path[5] joining u and v, and denote this by $d(u, v)$. In this graph, for example, the shortest path between a and e is $abcde$, which has length 4. So $d(a, e) = 4$. We can also define distance for directed graphs by using directed paths instead.

This notion of distance would be more useful if we can make some edges "longer" than others.

Example 2.31. Consider **The Cheapest Reid:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/the-cheapest-reid. As mentioned earlier, we can model this as a directed graph. The input format states each route "costs the couple a total of c pesos," where c can differ for each route. You're looking for the cheapest cost to go from city a to city b.

Suppose you go use the directed path $axyb$. Then its total cost would be the cost of ax, plus the cost of xy, plus the cost of yb. So the cost of a directed path would be the sum of the costs of each arrow.

We can represent such a concept by associating a number with each edge of a graph, called its **weight**:



For example, edge ab has weight 5, and edge bc has weight 3. The length of a path is now the sum of the weights of each edge, so abc has length 8. A graph where each edge has a weight is called a **weighted graph**, and one where it doesn't is an **unweighted graph**.

The shortest path from a to e isn't ade, which has weight 10, but $abce$, which has weight 9. Thus, the distance $d(a, e) = 9$. Finding the distance between two nodes in a graph is known

[5] Technically, I should say *a* shortest path since there can be multiple shortest paths. For example, in the example graph, $abcde$ and $aside$ are both shortest paths from a to e.

as the **shortest path problem**, and we'll use different algorithms to solve it in the weighted and unweighted cases.

Note that a graph can be directed and weighted, directed and unweighted, undirected and weighted, or undirected and unweighted. Unless otherwise stated, we assume a graph is undirected and unweighted. We won't be dealing much with weighted graphs this module, but it's good to recognize when a graph is weighted and to know how to represent them.

Example 2.32. Let's restate the problem **Kapuluan ng Kalayaan:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/spratly-islands in terms of graphs. The islands and routes are nodes and edges, and the constraints tell us that it's a simple graph.

The subset of routes corresponds to a subgraph. The subgraph should include all the islands (it's *spanning*) and we should also be able to visit any island from any other island (it's *connected*). The average cost of the routes in the subgraph should also be as small as possible.

In other words, given a simple graph, we need to find the spanning connected subgraph with the smallest average weight.

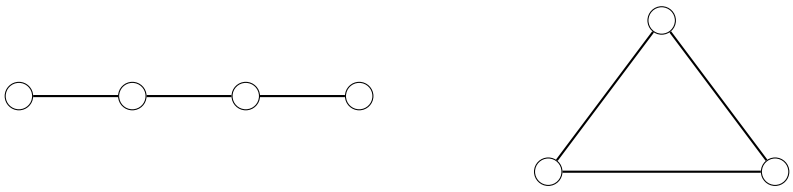
Exercise 2.33. Restate the problem **Burning Bridges [Redux]:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/burning-bridges-x in terms of graphs, like we did above.

2.8 Special graphs

Terms: path (P_n), cycle (C_n), complete (K_n), bipartite, complete bipartite ($K_{m,n}$), planar

Finally, we make our last few definitions by giving examples of specific, special kinds of graphs.

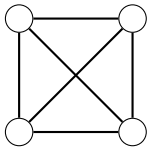
The first two are the **path graph** and the **cycle graph**. The path graph, denoted by P_n, has n nodes all in the same path. The cycle graph, denoted by C_n, has n nodes all in the same cycle. We often call C_3 the triangle and C_4 the square.

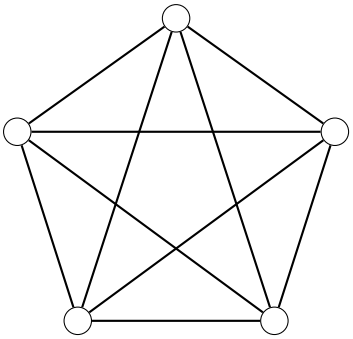


The above shows P_4 and C_3. Note that P_n has n nodes, but $n-1$ edges. In general, the subscript for all our notation will be related to the number of nodes. We will leave C_1 and C_2 as undefined.

Exercise 2.34. Is P_4 the only simple graph with 4 nodes and 3 edges? Is C_3 the only simple graph with 3 nodes and 3 edges?

The next is called the **complete graph**, denoted by K_n. It is the simple graph with n nodes, where each pair of distinct nodes is adjacent.





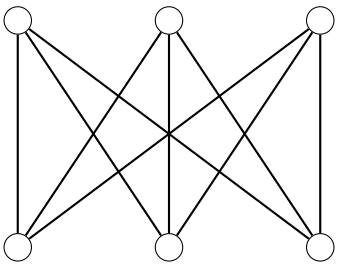
The above shows K_4 and K_5. Note that K_3 is the same as C_3, K_2 is the same as P_2, and K_1 is a graph with one node and no edges.

Example 2.35. Consider a valid solution to **.:** https://www.hackerrank.com/contests/noi-ph-2019-finals-2/challenges/mr-kupido By considering people as nodes, and drawing an arrow from one person to another if they love them, we form a directed graph. By turning every arrow into an undirected edge, you form a complete graph.

Exercise 2.36. How many edges does K_n have? Is K_n the only simple graph with n nodes and that many edges?

Exercise 2.37. Prove that any simple graph with n nodes is a spanning subgraph of K_n.

The next is a more general class of graphs called **bipartite graphs**. A graph is bipartite if its nodes can be partitioned into two sets X and Y, such that each edge connects a node in X to a node in Y. In particular, all the neighbors of a node in X are in Y, and all the neighbors of a node in Y are in X:



The above shows the typical way to draw a bipartite graph: all the nodes in one set are on one side, and all the nodes in the other set are on the other. This allows us to easily see that the graph is bipartite.

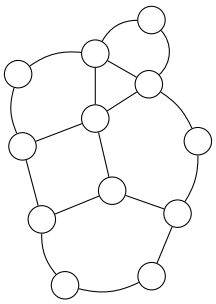
Exercise 2.38. For which n is P_n bipartite? For which n is C_n bipartite?

A **complete bipartite graph** is one where each node of X is joined to each node of Y. If $|X| = m$ and $|Y| = n$, we denote this graph by $K_{m,n}$. The above graph is $K_{3,3}$.

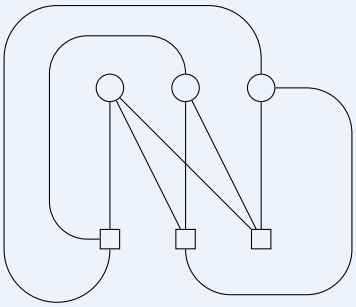
Exercise 2.39. How many edges does $K_{m,n}$ have?

Exercise 2.40. Describe $\overline{K_{m,n}}$.

Finally, a **planar graph** is one that we can draw on the plane such that any two edges intersect only at their endpoints. For example, this is a planar graph:



Example 2.41. A classic puzzle is the **three utilities puzzle**. We draw three houses and three utilities on a sheet of paper. We need to draw a curve joining each house to each utility; nine curves in total. No two curves are allowed to intersect, except at their endpoints. Is this possible?



From a graph theory perspective, the graph formed is $K_{3,3}$. (Check that!) The problem is asking us if $K_{3,3}$ is planar. In fact, this graph is *not* planar: that is, it is impossible to satisfy the constraints in the problem.

Another example of a graph that is not planar is K_5. It turns out that, in some sense, K_5 and $K_{3,3}$ are the "smallest" non-planar graphs in that every other non-planar graph "includes" them, for some definition of "includes" that we won't talk about here.

Exercise 2.42. While K_5 and $K_{3,3}$ aren't planar, K_4 and $K_{2,3}$ are. Prove this.

3 Representation

With all those definitions out of the way, we can finally start talking about programming! We first deal with the problem of representing a graph. There are several different ways that a graph can be represented, and the choice of which one to use depends on the application.

Note. VisuAlgo has a bunch of cool visualizations of graphs and their representations. I strongly recommend that you check it out here https://visualgo.net/en/graphds. It may be helpful to refer to the visualizations while you're reading this section.

3.1 Questions about graphs

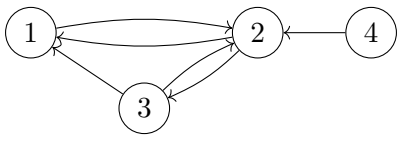
What do we want out of a representation? It helps to consider the kind of problems that we want to solve using graphs, and think about how we want to solve them:

- In the *shortest path problem*, we want to find the shortest path between two nodes. If we want to build the shortest path node-by-node, that means we need to quickly answer "what are the nodes adjacent to a given node?"
- Consider the *Hamiltonian cycle problem*, we want to find a cycle that passes through all the nodes, or the *Eulerian circuit problem*, we want to find a circuit that passes through all the edges. Again, we're building a path or a cycle, so we want to answer "what are the nodes adjacent to a given node?"
- Say we have a *graph coloring problem*, where we want to color the nodes with different colors, such that no two adjacent nodes have the same color. Suppose we're building a solution node-by-node. Then once we've colored a node, the next thing we want to do is color its neighbors, and color their neighbors, and so on. Again, the question we want to answer is "what are the nodes adjacent to a given node?"

From this small sample of problems, the most common question we want our graph representation to answer is this: given a node, enumerate all of its neighbors. Since this question is so common, we want a representation that quickly answers this question.

3.2 Adjacency list

What's the simplest solution to this problem? Well, if we want to enumerate the neighbors of each node quickly, why don't we just store a list of all of each node's neighbors? Consider the following directed graph:



Here's a list of each node and all the nodes it can go to:

node	adjacent
1	$\{2\}$
2	$\{1, 3\}$
3	$\{1, 2\}$
4	$\{2\}$

The list of these lists is called the **adjacency list**. Given the adjacency list, answering our question is easy! If we want to enumerate all the neighbors of node v, we simply find the vth list, and then loop through it. To program this, we typically use an array of vectors. For example:

Example 3.1. In this problem, we take a directed graph in input. The first line is two space-separated integers n and m, the number of nodes and arrows. The nodes will be $1, 2, \ldots, n$. The next m lines consist of two space-separated integers a and b, indicating that there is an arrow going from a to b.

The next line is an integer q, the number of queries. Each subsequent line consists of a single integer v. For each query, we print the neighbors of node v.

```cpp
const int N = 1001; // pick something at least maximum n + 1
vector<int> g[N]; // adjacency list

int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        g[a].push_back(b); // adds an arrow a->b
    }

    int q;
    cin >> q;
    for (int i = 0; i < q; i++) {
        int v;
        cin >> v;
        for (auto u : g[v]) { // u goes over each of v's neighbors
            cout << u << ' ';
        }
        cout << '\n';
    }
}
```

Exercise 3.2. Convince yourself that the adjacency list uses $\mathcal{O}(n + m)$ memory for a graph with n nodes and m edges.

Exercise 3.3. Does this code still work when we have a directed loop? What about multiple arrows?

Some notes:

- The above code is for *directed* graphs. Suppose, instead, that our graph was **undirected**, and that each line in the input was an (undirected) edge instead of an arrow. Then instead of just `g[a].push_back(b);`, we would also do `g[b].push_back(a);`. Make sure you see

why the program still works in this case!

- What if we wanted to support **weighted** graphs instead? The easiest solution is to do something like:

```
vector<pair<int, int>> g[N];
// add an arrow a->b with weight w:
g[a].push_back({b, w});
// iterating over v's neighbors:
for (auto [u, w] : g[v]) {
    // arrow v->u has weight w
}
```

The `auto [u, w]` syntax is a common way to do unpacking in C++, that is, split a `pair` or `tuple` into its parts and assign them to variables.

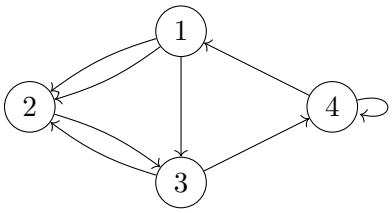
- When there are multiple test cases with different graphs, you need to **reset** the adjacency list:

```
// reset before an n-node graph
for (int i = 1; i <= n; i++) {
    vector<int>().swap(g[i]); // clear vector g[i]
}
```

For speed purposes, you want to clear vectors using the swap trick demonstrated above. Note that `memset(g, 0, sizeof(g));` also works, but beware that this is much slower and much more wasteful, especially if you have a lot of test cases with a small total number of nodes.

3.3 Adjacency matrix

The adjacency list is the representation you're going to be using the most. There are a handful of algorithms, however, that work better with a different graph representation. The adjacency matrix quickly answers the question "Given two nodes, are they adjacent?" Consider the following graph:



Say that a graph has n nodes. Then its **adjacency matrix** is the $n \times n$ matrix, whose entry in the ith row and jth column in the adjacency matrix is the number of arrows (or edges) going from i to j:

$$\begin{matrix} 0 & 2 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \end{matrix}$$

For example, there's a 2 in the entry in the 1st row and 2nd column, because there are 2 arrows going from node 1 to node 2.

Exercise 3.4. In a simple graph, convince yourself that everything on the main diagonal is 0, and every other entry is either 0 or 1. (The main diagonal of a matrix goes from its upper left to its lower right.)

Exercise 3.5. What do the column sums and row sums of an adjacency matrix correspond to?

Some problems ask us to take input through an adjacency matrix, or to output a graph by printing its adjacency matrix. For example, **City Map:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/city-map asks you to output a graph by printing its adjacency matrix.

To implement an adjacency matrix, we use a two-dimensional array:

```
const int N = 1001;
int g[N][N];
// set everything to 0
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        g[i][j] = 0;
// add an arrow a->b
g[a][b] += 1;
// iterating over v's neighbors:
for (int u = 1; u <= n; u++) {
    // g[v][u] is the number of arrows v->u
    // do something
}
```

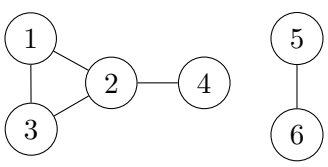
For **undirected** graphs, the adjacency matrix is symmetric about its main diagonal (meaning `g[i][j] = g[j][i]` for all i, j).

We can't really represent **weighted** graphs, unless they're also simple. In this case, we can simply make `g[u][v]` 0 if there's no edge joining u and v, and if there is, we set it to its weight. Of course, this only works if the weight is positive; if the weight can be zero or negative, we have to pick a different default value.

Compared to adjacency lists, adjacency matrices always use $\mathcal{O}(n^2)$ memory, and take $\mathcal{O}(n)$ time to enumerate all the neighbors of a given node, which makes them inconvenient for most purposes. The advantage is that an adjacency matrix takes $\mathcal{O}(1)$ time to check if two nodes are adjacent, while an adjacency list takes $\mathcal{O}(n)$ time (why?).

3.4 Edge list

While the adjacency list is the representation you'll *use* the most, the edge list is the representation you'll *encounter* the most. The **edge list** is simply a list of edges. This graph:



has this edge list, which directly comes from the formal definition of the graph:

$$\{\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{5,6\}\}.$$

Most graph problems will give their input using an edge list. The input format will say something like "The next e lines contain two space-separated integers a and b, meaning there is a two-way flight between cities a and b."

If for some reason you need to implement an edge list, we simply use a vector of pairs to store the edges:

```
1 vector<pair<int, int>> g;
2 // add an arrow a->b
3 g.push_back({a, b});
4 // iterating over v's neighbors:
5 for (auto [v_, u] : g) {
6     if (v != v_) continue;
7     // arrow v->u
8     // do something
9 }
```

For **undirected** graphs, we can choose to add both directions or not, and we modify line 6 accordingly.

Exercise 3.6. Convince yourself that the adjacency list uses $\mathcal{O}(m)$ memory for a graph with n nodes and m edges.

Exercise 3.7. Show that lines 5 to 8 run in $\mathcal{O}(m)$, if m is the number of edges.

For **weighted** graphs, we can use a `vector<tuple<int, int, int>>` instead, and store an arrow from a to b with weight w as a tuple (w,a,b). The advantage of (w,a,b) over (a,b,w) is that we can easily sort tuples by weight: because the `sort` function in C++ sorts lexicographically, sorting the vector will sort the tuples by increasing w.

The only common graph algorithm that uses an edge list is called Kruskal's algorithm. In practice, I have never used an edge list for any application other than Kruskal's algorithm or a modification of it, so don't worry about edge lists too much.

4 Traversal

Graph *traversal* is a fancy word for searching a graph. It's an algorithm that goes through all the nodes in a graph in a natural order.

While we *could* go through the nodes of the graph by increasing node number, that doesn't really make sense, because it doesn't refer to the "structure" of the graph. The thing about graph traversal is that it goes through the nodes by moving through the edges, so you're only checking nodes that are connected to the node you search from.

Note. VisuAlgo again has cool animations of graph traversal. I strongly recommend that you check it out here https://visualgo.net/en/dfsbfs. The animation makes it much easier to understand what each algorithm *does*!

4.1 Depth-first search

The first kind of traversal is depth-first search, or DFS. To do this, we start at some node. Then we keep going through the neighbors of its nodes, making sure not to repeat any nodes, until we reach a node that doesn't have any unvisited neighbors. We then *backtrack* and try the next possibilities.

Supposing that the graph is stored in an adjacency list in `g`:

```
bool visited[N];
for (int i = 1; i <= n; i++) visited[i] = false;

void dfs(int v) {
    visited[v] = true;
    for (auto u : g[v]) { // u is v's neighbor
        if (!visited[u]) dfs(u);
    }
}
```

Then, calling `dfs(i)` starts a depth-first search at node i.

> **Exercise 4.1.** Why is it important that we check if u hasn't been visited yet?

> **Exercise 4.2.** Convince yourself that DFS runs in $\mathcal{O}(n+m)$, where n and m are the number of nodes and edges, respectively.

If you'll remember the Backtracking 1 module, then you'll notice how similar this is to our backtracking code:

```
def backtrack(candidate):
    if candidate is done:
        do something
    else:
        for each possible next_candidate:
            backtrack(next_candidate)
```

In fact, one of the exercises in the Backtracking 1 module compares backtracking to DFS. The idea is that backtracking does DFS on a graph of all possible configurations.

As explained in this module, we can convert any backtracking solution to an iterative solution using a stack. Similarly, we can convert DFS to be done using a stack as well:

```
bool visited[N];
for (int i = 1; i <= n; i++) visited[i] = false;
stack<int> st;

st.push(i); // node to start DFS in
while (!st.empty()) {
    v = st.top(); st.pop();
    visited[v] = true;
    for (auto u : g[v]) {
        if (!visited[u]) st.push(u);
    }
}
```

Exercise 4.3. Compare this code to the recursive version of DFS, and convince yourself they do the same thing. Note that v and u are the same variables in both.

4.2 Breadth-first search

The second kind of traversal is breadth-first search, or BFS. In BFS, we visit all adjacent nodes first *before* visiting the neighbors of these adjacent nodes. This means we're visiting nodes in order from nearest to farthest: first we visit nodes that are one edge away, then we visit nodes that are two edges away, and so on.

Implementation-wise, we just take the DFS code and replace the stack with a queue:

```
bool visited[N];
for (int i = 1; i <= n; i++) visited[i] = false;
queue<int> qu;

qu.push(i); // node to start DFS in
while (!qu.empty()) {
    v = qu.front(); qu.pop();
    visited[v] = true;
    for (auto u : g[v]) {
        if (!visited[u]) qu.push(u);
    }
}
```

Exercise 4.4. Convince yourself that BFS runs in $\mathcal{O}(n+m)$, where n and m are the number of nodes and edges, respectively.

4.3 Applications

Some applications:

- In each step, a DFS or BFS only visits nodes that are adjacent. Therefore, at the end of a search, the nodes visited are exactly the nodes in the same connected component as the root. This has applications in finding nodes in the same connected component.
- By going through each unvisited node, and visiting all the nodes in its connected component, we can also count the number of connected components, label them, find their sizes, and so on.
- Since a BFS visits nodes from nearest to farthest, in an unweighted graph, it can also compute distances from the source node.
- We can use BFS or DFS to find cycles. When we visit a node v, that means that it has one neighbor we already visited: the neighbor where we just came from, say, u. Suppose that one of v's neighbors other than u has already been visited. Then that means that the graph must have a cycle: do you see why?
- We can also use BFS or DFS to check if the graph is bipartite. Note that a graph is bipartite if we can divide it into X and Y, such that no two nodes from X are adjacent and no two nodes from Y are adjacent.

  Well, we can just start from a node and start dividing them into X and Y: we know all the neighbors of a node in X has to be in Y, and all the neighbors of a node in Y has to be in X. So we label the first node arbitrarily, then label each of its neighbors in turn.

  At the end, if two adjacent nodes have the same label, we know the graph isn't bipartite. Otherwise, it is. Do you see why?

There are lots more stuff that we can do with BFS and DFS, and this is just a sampler.

Example 4.5. Consider a problem like **Yumamma II:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/yumamma-ii. We need to find the shortest path between Yumamma and the onion. Finding the shortest path in a grid should be a signal that we should try converting it to a graph. In general, **for anything involving motion in a grid, a good idea would be to convert it to a graph.**

How do we do this? Consider a graph where there's one node for each of Yumamma's possible positions in the grid. Then, draw an edge between two nodes if Yumamma can move between them. The problem is now equivalent to finding the shortest path between Yumamma's starting node, to any of the four possible ending positions, which we can now solve with a BFS!

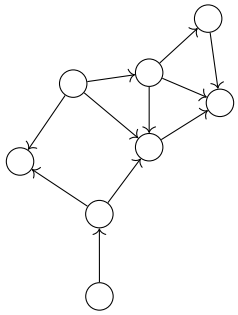
Implementation-wise, to convert a grid to a graph, it helps to have some way to convert the node (r, c) to a single integer, and then convert it back. One way would be to use the function $(r, c) \to 1001r + c$. That way, given $n = 1001r + c$, then the inverse function would be

$$n \to \left(\left\lfloor \frac{n}{1001} \right\rfloor, n \bmod 1001 \right),$$

so we can recover r and c. Then, you can use the typical techniques to work with grids. This will be discussed in more detail in Data Structures 1.

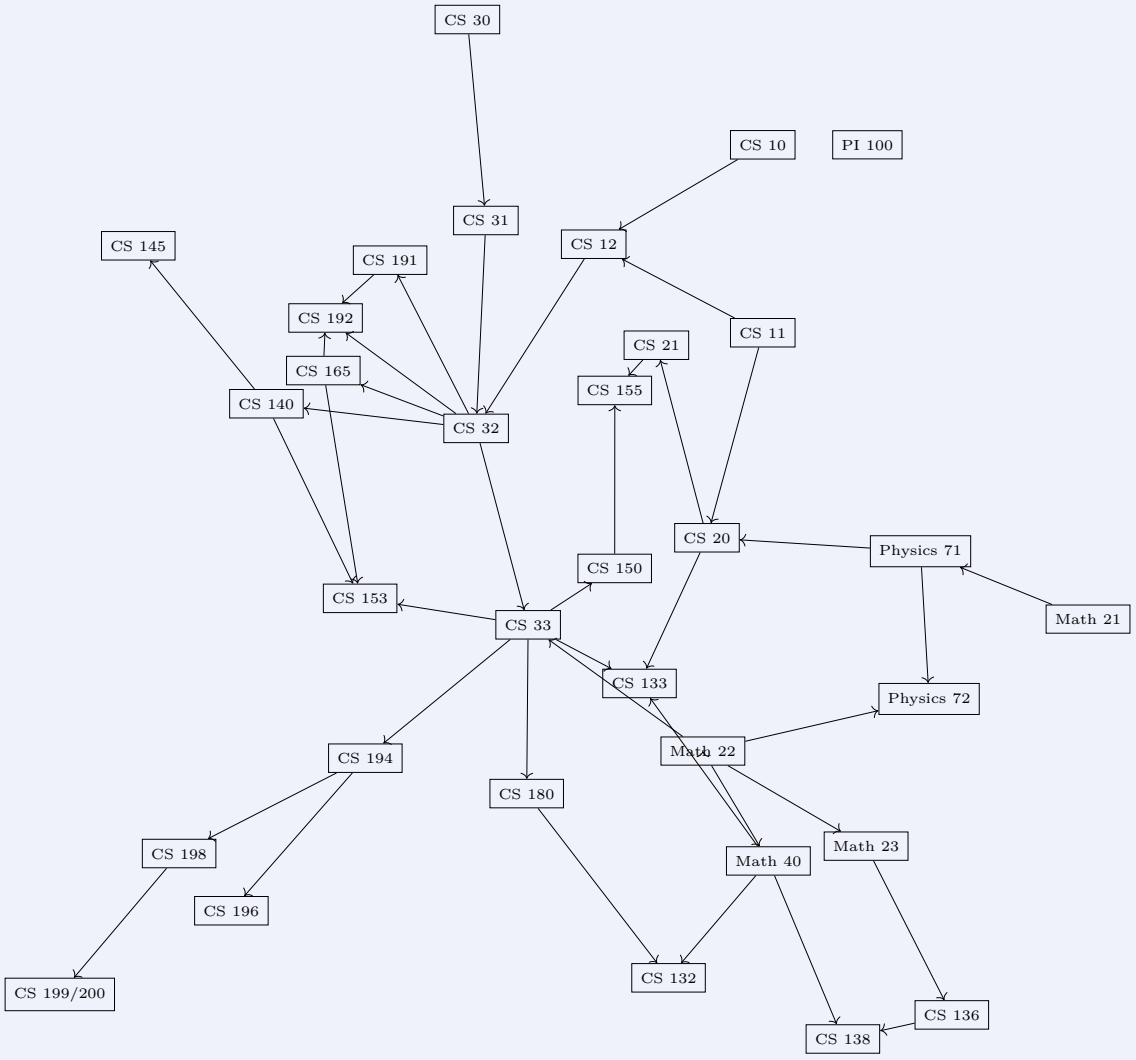
5 Directed acyclic graphs

A **directed acyclic graph**, or a DAG, is a directed graph that does not have any directed cycles:



DAGs usually arise in cases where we have to do one thing before another thing:

Example 5.1. Consider enrolling for subjects in college. Some subjects have other subjects as prerequisites. Consider the graph where subjects are nodes, and where we draw an arrow from a subject's prerequisite to its subject.



We wouldn't want this graph to have any circular dependencies, otherwise no one would be able to take these subjects! So this directed graph can't have any cycles, and this is an example of a DAG.

Example 5.2. Consider doing DP on states. When we process a state, we need to process all of the states that it depends on, like the states that "come before it". The fact that states can't have circular dependencies means that this graph has to be a DAG.

One important property of a DAG is that it has a *minimal node*, or a node that has indegree 0:

Exercise 5.3. Convince yourself that a DAG has a node v such that $\deg^-(v) = 0$.

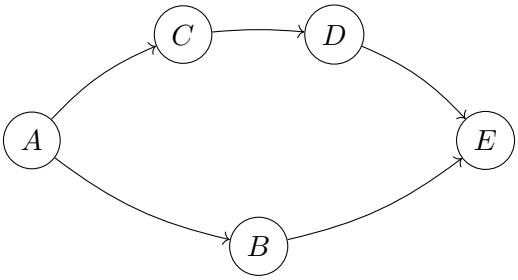
Note that if we reverse all the arrows of a DAG, we also get another DAG. So in the reverse DAG, there's a node u such that $\deg^-(u) = 0$. In the original DAG, then, this corresponds to a node where $\deg^+(u) = 0$. So any DAG has a node with indegree 0, and a node with outdegree 0.

5.1 Toposort

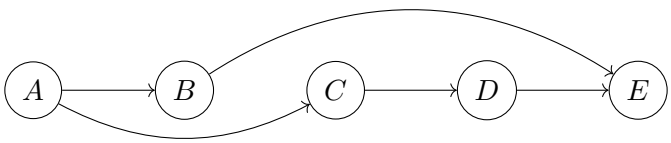
Because a DAG does not have any circular dependencies, there has to be some way to order it such that all the arrows "point forward":

Example 5.4. Go back to the example of choosing subjects in college. Say you have a list of subjects that you need to take in order to graduate. Then there's some order you can take these subjects, so by the time you take a subject, you've already taken all its prerequisites.

Such an ordering is called a **topological sorting**, or a toposort. A single DAG can have multiple toposorts. For example, this graph:



has A, B, C, D, E as one toposort, and A, C, D, B, E as another. If you order the nodes in a graph as in a toposort, and draw all the arrows:



then all of the arrows have to "point forward". Note that **all DAGs have a toposort**, which we can prove by finding such a toposort using an algorithm.

There are multiple ways to do this, but the simplest way is to do a DFS:

```cpp
bool visited[N];
for (int i = 1; i <= n; i++) visited[i] = false;
vector<int> ts; // stores reversed toposort

void dfs(int v) {
    visited[v] = true;
    for (auto u : g[v]) {
        if (!visited[u]) dfs(u);
    }
    ts.push_back(v); // add v to the toposort
}

for (int i = 1; i <= n; i++) {
    if (!visited[i]) dfs(i);
}
```

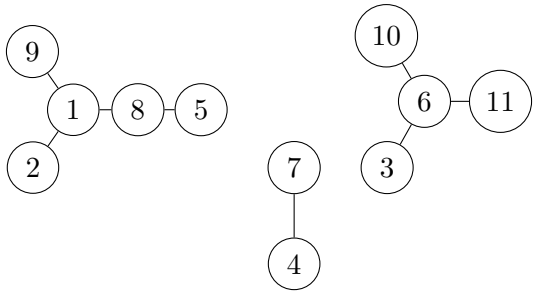
After running the above code, `ts` will now store a reverse toposort. By reversing it, you then get a toposort.

Exercise 5.5. Prove that the above algorithm really *does* give a toposort. This is equivalent to proving that the children of each node appear in the toposort after that node.

Exercise 5.6. What is the complexity of the above algorithm?

6 Trees

A **forest** is a simple graph without any cycles. A **tree** is a connected forest. So this is a forest, and each connected component is a tree:



6.1 Properties

Trees have a lot of equivalent definitions. A graph is a tree if any of the following is true:

- It is a simple connected graph without any cycles.
- It is a simple connected graph, but removing any edge would make it disconnected.
- It is a simple graph without any cycles, and adding any edges would make it have a cycle.
- It is a simple graph where any two nodes are connected by exactly one path. (This is the definition you'll see the most in problem statements.)
- It is a simple connected graph with n nodes and $n - 1$ edges.
- It is a simple graph with n nodes, $n - 1$ edges, and no cycles.

Exercise 6.1. Prove that each of these imply the graph is a tree, and that if a graph is a tree then each of these have to be true. (For this problem, the definition of a tree is a *simple connected forest*.)

Exercise 6.2. For which of these can we remove the word "simple"?

Here are some more properties of trees:

- They have at least two nodes of degree one.
- They are bipartite graphs.
- Every connected graph has a subgraph that is both spanning and a tree. This subgraph is called, aptly, a **spanning tree**.

Exercise 6.3. Prove this last property! As a hint, you can try using DFS or BFS.

6.2 Rooted trees

We consider a tree **rooted** if we single out a specific node as the **root** node.

A node adjacent to the root is called its **child**, while the root is called its **parent**. Similarly, all of a node's neighbors, except its parent, are called its children.

Aptly, the child of a child is called a grandchild, and the parent of a parent is called a grandparent. The general terms are ancestors and descendants. Kind of like a family tree!

Each node in a tree produces a subgraph, consisting of itself and all its descendants. This subgraph is also a rooted tree, and is called the **subtree** rooted at that node.

The **depth** of any node in a rooted tree is its distance to the root. So the depth of the root is 0, the depth of the root's children is 1, and so on. The **height** of a rooted tree is the maximum depth of any node in that tree.

6.3 Binary trees

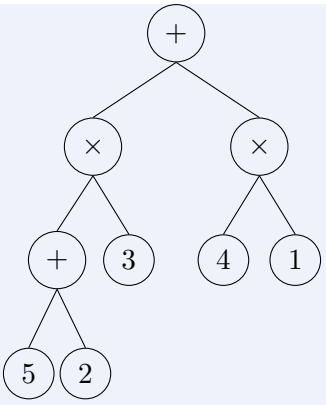
A rooted tree is called a **k-ary tree** if each node has at most k children. When $k = 2$, the most common case, we call it a **binary tree**.

In a binary tree, we often order the children of each parent too. We call one child the **left child** and one child the **right child**.

There are special ways to traverse a binary tree, called **preorder**, **inorder**, and **postorder**. These are all forms of DFS on a given tree. Here are what they looks like:

```
int left[N];  // left[i] is index of i's left child
int right[N]; // right[i] is index of i's right child

void preorder_dfs(int v) {
    // process v
    preorder_dfs(left[v]);
    preorder_dfs(right[v]);
}

void inorder_dfs(int v) {
    inorder_dfs(left[v]);
    // process v
    inorder_dfs(right[v]);
}

void postorder_dfs(int v) {
    postorder_dfs(left[v]);
    postorder_dfs(right[v]);
    // process v
}
```

Example 6.4. We can consider evaluating an arithmetic expression like $(5 + 2) \times 3 + 4 \times 1$ as a tree:



Suppose that our processing for a node is printing it. Then the result of each of the traversals would be:

- Preorder: $+ \times + 5\,2\,3 \times 4\,1$
- Inorder: $5 + 2 \times 3 + 4 \times 1$
- Postorder: $5\,2 + 3 \times 4\,1 \times +$

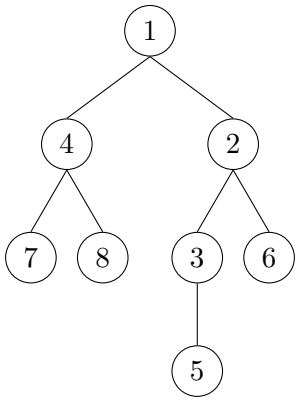
These are known as *prefix*, *infix*, and *postfix* notation.

We can also traverse a tree in **levelorder**. This amounts to traversing all the nodes that are depth 0, then all the nodes that are depth 1, then all the nodes that are depth 2, and so on.

Exercise 6.5. Convince yourself that a BFS from the root gives a levelorder traversal.

6.4 Parent array

While we can store a rooted tree using an adjacency list, like the same way we store other graphs, there are more compact ways of storing a tree. One example is the **parent array**, where we store the parent of each node. For example, the rooted tree



has the parent array

i	1	2	3	4	5	6	7	8
`par[i]`	-1	1	2	1	3	2	4	4

The value of `par[i]` is the parent of node i. So for example, since the parent of 7 is 4, we have `par[7] = 4`. Since 1 is the root and has no parent, we use a value like 0 or -1 to mark that it

is the root.

One very useful application of a parent array is generating random trees:

Exercise 6.6. Prove that any array `par` where `par[1] = -1` and `par[i] < i` is a parent array of a tree with root at 1. This gives us an easy way to generate random trees.[a]

[a] The trees aren't necessarily sampled uniformly among all trees though.

Sometimes, a tree is also given in a problem statement using its parent array.

7 Union-find

Union-find or disjoint set union is a data structure that stores a partition of a set. This means that it stores several subsets of a given set such that *each element* in the set is in *exactly one* subset. So this is a partition of the set $\{1, 2, \ldots, 10\}$:

$$\{1, 4, 5, 6\}, \{2, 3\}, \{7, 9, 10\}, \{8\}.$$

The union-find data structure stores a partition, and supports two operations.

The first is **find**, which determines which subset an element is part of. The find function returns a representative element of its subset. So for example, in the above partition, `find(1)` and `find(4)` would return the same thing, while `find(2)` and `find(7)` would return different things.

The second is **union**, which combines the elements of two subsets. Calling `unite(2, 7)` on the above partition would give us

$$\{1, 4, 5, 6\}, \{2, 3, 7, 9, 10\}, \{8\}.$$

Then calling `unite(1, 8)` would give us

$$\{1, 4, 5, 6, 8\}, \{2, 3, 7, 9, 10\}.$$

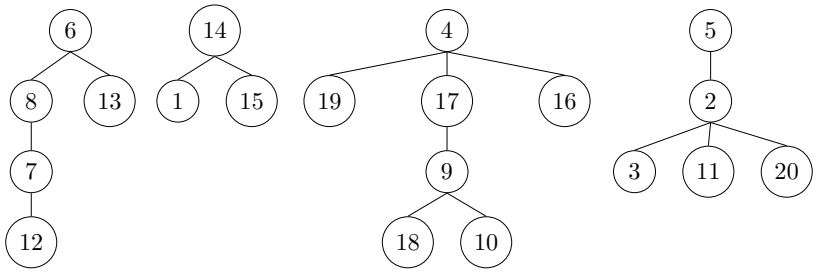
Calling `unite(3, 9)` would not make any changes. If we now call `unite(5, 10)`, the two subsets would now combine into one big set.

7.1 Implementation

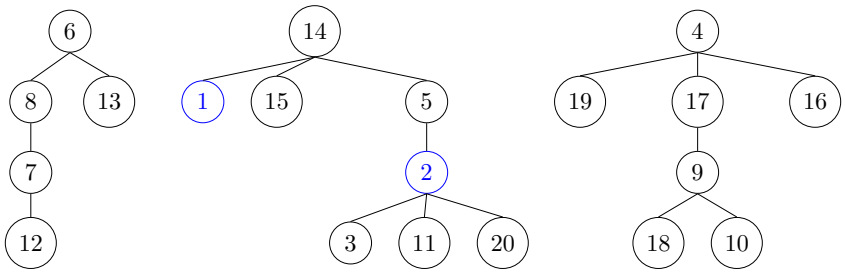
Note. Again, VisuAlgo has cool animations of this, which you can check out here https://visualgo.net/en/ufds.

The idea of implementing union-find is that the elements in each subset are stored in a tree.

Each element is a node, and if two nodes are in the same subset, they belong to the same tree. We then make `find` return the element at the root of this tree. To implement `unite`, we simply combine the two trees by making the root of one tree as the child of the other tree's root. For example, if we have



and we call `unite(1, 2)`, we get:



Initially, each node is in its own subset, so each node will be the root of its own tree. This is easily implemented with a parent array:

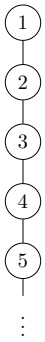
```
int par[N]; // parent array
for (int i = 1; i <= n; i++) par[i] = -1;

int find(int v) {
    if (par[v] == -1) return v; // u is the root
    return find(par[v]);
}

void unite(int u, int v) {
    u = find(u); // find u's root
    v = find(v); // find v's root
    if (u != v) // if they're from different subsets:
        par[u] = v; // set u's parent to v
}
```

Note that we use `unite` because `union` is a reserved word. (Some people like to use `onion` or `merge`.)

Exercise 7.1. Convince yourself that the `find` command here works.

7.2 Optimizations

In the worst case, you can end up with really, really bad-looking trees:



Here, calling `find` would take $\mathcal{O}(n)$ time, if there are n nodes. This is because the worst case is that you get really, really long chains like the above. So `unite`, which also calls find, would also take $\mathcal{O}(n)$ time.

We can optimize `find` by replacing the parent of each node with the root. This way, calling `find` would take long at first, but the second time you call it would be faster:

```
int find(int v) {
    if (par[v] == -1) return v;
```

```
    return par[v] = find(par[v]);
}
```

This is called **path compression**, because you're compressing the path from a node to its root.

We can further optimize `unite` by making sure we only merge the "smaller" tree to the "larger" tree, so that long chains like the above are avoided in the first place.

To do this, we use `par` to store the negative *depth* of each tree, so that we only merge trees with smaller depth to trees with larger depth. Then we update the depth of the new tree.

```
int unite(int u, int v) {
    u = find(u); // -par[u] is u's depth
    v = find(v);
    if (u == v) return;
    if (-par[u] < -par[v]) { // v is deeper
        par[u] = v;
    } else if (-par[v] < -par[u]) {
        par[v] = u;
    } else { // same depth
        par[v] = u;
        par[u]--; // u's depth increases by one
    }
}
```

This is called **union by rank**, since you're uniting trees based on their rank.

> **Exercise 7.2.** Instead of `par` storing the negative *depth* of each tree, it can also store the negative *size* of each tree, or the number of nodes. Then, if we always merge trees with smaller size to trees with larger size, we get what's called **union by size**.
>
> Implement this. Make sure to update the size of the new tree after merging them. Union by size is also useful if we want a function to find the size of a tree.

If you're having trouble debugging, I suggest simply creating a separate array for the depth or size. This way, you can keep the sentinel `-1` value, and you don't have to negate the depth or the size.

If we're combining both optimizations, then **we have to change the check in `find` to `par[v] < 0`**. This is easy to miss when implementing path compression and union by rank for the first time, so make sure to remember this!

The complexity of both `find` and `unite` after combining both optimizations is $\mathcal{O}(\alpha(n))$, a function that's called the inverse Ackermann function. For all practical purposes its value is less than 5, so you can treat it as a constant when you're calculating complexity. The proof of this is way outside the scope of this handout.

A typical union-find interpretation combining both optimizations, with much more concise code, looks like:

```
int par[N]; // parent array
for (int i = 1; i <= n; i++) par[i] = -1;
```

```
3
4  int find(int v) {
5      return par[v] < 0 ? v : par[v] = find(par[v]);
6  }
7
8  void unite(int u, int v) {
9      u = find(u); v = find(v);
10     if (u == v) return;
11     if (par[u] == par[v]) par[u]--;
12     if (par[u] > par[v]) swap(u, v);
13     par[v] = u;
14 }
```

Exercise 7.3. Convince yourself that `unite` does the same thing in this fancy implementation.

I wouldn't recommend memorizing it; but if you understand what it does then you'll probably end up memorizing it anyway.

8 Problems

Discussion policy:

- If it's worth [2★] or [3★], you can ask for hints, give hints to others, and discuss the problem on Discord.
- If it's worth more, you can ask for hints on Discord, and wait for one of the trainers to give a hint. Do not give other people hints.

This is a pretty mathy week: even the coding problems will be more math-flavored than algorithms-flavored. Expect to be working more on paper than debugging on your computer.

8.1 Warmup problems

W1 City Map: https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/city-map

W2 Path in a Maze: https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/path-in-a-maze

W3 Onion Find: https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/onion-find

W4 Basic Tree Traversals: https://www.hackerrank.com/contests/noi-ph-2015-camp-cramers/challenges/basic-tree-traversals

W5 Basic Tree Reconstruction 1: https://www.hackerrank.com/contests/noi-ph-2015-camp-cramers/challenges/basic-tree-reconstruction-1

W6 Ordering Tasks: UVa 10305

8.2 Non-coding problems

8.2.1 Debugging set

Each of the following programs either has a mistake (possibly several mistakes), is too slow under normal time limits, or uses too much memory under normal memory limits. For each problem, identify what is wrong with the code, and how to fix it. In cases where you think the algorithm is incorrect, it would also be great if you could find a counterexample.

All students are required to solve [20★].

N.D1 [5★] The following code attempts to solve the following problem: *Given a simple undirected graph with $n \leq 100\,000$ nodes and $e \leq 200\,000$ edges, and two nodes s and t, find the length of the shortest path from s to t (if it exists).*

```cpp
#include <bits/stdc++.h>
using namespace std;
const int N = 1'00'000;
const int INF = N;

vector<int> adj[N];
int memo[N];
int dist(int s, int t) {
    if (s == t) return 0;
    if (memo[s] != -1) return memo[s];
    memo[s] = INF;
    for (int i : adj[s]) {
        memo[s] = min(memo[s], 1 + dist(i, t));
    }
    return memo[s];
}

void solve() {
    int n, e, s, t;
    cin >> n >> e >> s >> t;
    s--, t--;
    for (int i = 0; i < n; i++) {
        memo[i] = -1;
        adj[i].clear();
    }
    while (e--) {
        int i, j;
        cin >> i >> j;
        i--, j--;
        adj[i].push_back(j);
        adj[j].push_back(i);
    }
    int res = dist(s, t);
    if (res < INF)
        cout << res << '\n';
    else
```

```
        cout << "N/A\n";
}

int main() {
    int z;
    for (cin >> z; z--; solve());
}
```

N.D2 [5★] The following code attempts to do a DFS, for a graph with $n \leq 100\,000$ nodes.

```
bool visited[N];
void init() {
    for (int i = 1; i <= n; i++) visited[i] = false;
}

void dfs(int v) {
    visited[v] = true;
    cout << "now visiting node: " << v << '\n';
    for (int u : g[v]) {
        if (!visited[u]) dfs(u);
    }
    cout << "finished visiting node: " << v << '\n';
    visited[v] = false;
}
```

N.D3 [5★] The following code attempts to find the shortest path between u and v, in an unweighted graph with adjacency list `adj`, with $n \leq 100\,000$ nodes.

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100'011;
const int INF = 1 << 29;

vector<int> adj[N];
int dist[N];
int n;
int shortest_path(int u, int v) {
    for (int i = 0; i < n; i++) dist[i] = -1;
    deque<int> queu;
    queu.push_back(v);
    dist[v] = 0;
    while (!queu.empty()) {
        int i = queu.front();
        queu.pop_front();
        for (int j : adj[i]) {
            if (dist[j] == -1) {
                dist[j] = dist[i] + 1;
                queu.push_front(j);
```

```
            }
        }
    }
    return dist[u] == -1 ? INF : dist[u];
}
```

N.D4 [5★] The following code attempts to find the shortest path between u and v, in an unweighted graph with adjacency list `adj`, with $n \leq 100\,000$ nodes.

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100'011;
const int INF = 1 << 29;

vector<int> adj[N];
int n;

map<pair<int,int>,bool> memo;
bool reaches_end(int d, int u, int v) {
    if (d < 0) return false;
    if (u == v) return true;
    if (memo.count({d, u})) return memo[{d, u}];
    bool ans = false;
    for (int i : adj[u]) {
        if (reaches_end(d - 1, i, v)) {
            ans = true;
            break;
        }
    }
    return memo[{d, u}] = ans;
}

int shortest_path(int u, int v) {
    memo.clear();
    for (int d = 0; d < n; d++) {
        if (reaches_end(d, u, v)) return d;
    }
    return INF;
}
```

N.D5 [5★] The following code attempts to find the shortest path between u and v, in an unweighted graph, with $n \leq 100\,000$ nodes.

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100'011;
const int INF = 1 << 29;

vector<int> adj[N];
```

```cpp
int dist[N], n;

int shortest_path(int u, int v) {
    for (int i = 0; i < n; i++) {
        dist[i] = INF;
    }

    vector<int> queue;
    queue.push_back(u);
    dist[u] = 0;
    for (int pos = 0; pos < queue.size(); pos++) {
        int i = queue[pos];
        if (i == v) return dist[i];
        for (int j = 0; j < n; j++) {
            if (adj[i][j] && dist[j] >= INF) {
                dist[j] = dist[i] + 1;
                queue.push_back(j);
            }
        }
    }
    return INF;
}
```

8.2.2 Math problems

No need to be overly formal in your answers; as long as you're able to convince me, it's fine!

All students, solve at least [45★]. Red problems are recommended.

Veterans: problems worth less than [4★] don't count, and you're required to solve at least one of the problems worth at least [13★].

N1 [5★] Exercise 5.5

N2 [18★] Exercise 6.1 ([3★] each)

N3 [3★] Let G be a simple graph isomorphic to $\overline{G}$. If n is the number of its nodes, show that $n \equiv 0$ or $1 \pmod 4$.

N4 Suppose the ith node of a graph G has degree d_i. We call the sequence $d_1, d_2, \ldots, d_n$ the **degree sequence** of G.

(a) [5★] Prove that a sequence of non-negative integers $d_1, d_2, \ldots, d_n$ is the degree sequence of some graph if and only if $d_1 + d_2 + \cdots + d_n$ is even. (Note that this statement is not true for simple graphs.)

(b) [13★] Prove that a sequence of positive integers $d_1, d_2, \ldots, d_n$ is the degree sequence of some tree if and only if $d_1 + d_2 + \cdots + d_n$ is $2n - 2$.

(c) [17★] Prove that a sequence of non-negative integers $d_1 \geq d_2 \geq \ldots \geq d_n$ is the degree sequence of some simple graph if and only if

$$d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n$$

is the degree sequence of a simple graph as well. This is known as the Havel–Hakimi theorem. (The statement itself should suggest an algorithm for constructing such a graph, but proving it works is much harder.)

N5 [2★] How many edges does K_n have, in terms of n? If a simple connected graph has n nodes and this many edges, does it have to be K_n?

N6 [3★] Is $K_{m,n}$ the only simple graph with $m+n$ nodes and that many edges? Is $K_{m,n}$ the only simple *bipartite* graph with $m+n$ nodes and that many edges?

N7 Adjacency matrices have cool mathematical properties, particularly if you multiply them.

(a) [3★] Let G_1 and G_2 be two graphs with the same nodes $1, 2, \ldots, n$. Let A be the adjacency matrix of G_1 and B be the adjacency matrix of G_2. Show that the entry of AB in the ith row and jth column is the number of ways to get from i to j by going first along an edge of G_1, and then along an edge of G_2.

(b) [5★] Let A be the adjacency matrix of a graph G. Show that the entry of A^k in the ith row and jth column is the number of walks from the ith node to the jth node of length k.

(c) [2★] Prove that the number of triangles in an undirected graph G is one-sixth the trace of A^3. The *trace* of a square matrix is the sum of the entries on the main diagonal.

(d) [8★] Let $\mathcal{P}$ be the set of $n \times n$ matrices that have exactly one entry of 1 in each row and each column and 0s elsewhere. Prove that G_1 and G_2 are isomorphic if and only if there exists some matrix P in $\mathcal{P}$ such that $PA = BP$.

In particular, this means that A and B have the same determinant, trace, characteristic polynomial, eigenvalues, and so on. (Don't worry if you don't know what any of these mean.) This opens up the study of a field called *spectral graph theory*.

N8 [3★] Prove that any simple connected graph with n nodes has a connected subgraph with k nodes, for all positive integers $k < n$.

N9 [3★] A **k-partite graph** is one whose node set can be partitioned into k subsets so that each edge connects a node to a node in a different subset. A **complete k-partite graph** is a simple k-partite graph where each node is adjacent to every node not in the same subset, usually denoted $K_{n_1,n_2,\ldots,n_k}$ if it has n_i nodes in the ith subset. Show that it has

$$\frac{(n_1 + n_2 + \cdots + n_k)^2 - (n_1^2 + n_2^2 + \cdots + n_k^2)}{2}$$

edges.

N10 [5★] Show that, for any three nodes u, v, w, then $d(u,v) + d(v,w) \geq d(u,w)$. This is known as the **triangle inequality**.

N11 Let the **diameter** of a graph be the maximum value of $d(u,v)$ over all pairs of nodes u and v. Call a path the **longest path** if its length is the diameter.

(a) [2★] Prove that a tree with n nodes has diameter at most 2 if and only if there exists a node v with $\deg(v) = n - 1$.

(b) [5★] Take an arbitrary node u in a tree. Let v be such that $d(u,v)$ is a maximum over all nodes v. Let w be such that $d(v,w)$ is a maximum over all nodes w. Show that $d(v,w)$ is the diameter of the tree.

(c) [8★] Show that any two longest paths share a node in common.

(d) [3★] Let n be a positive integer. Give an example of a graph with n nodes where all pairs of longest paths share exactly one node in common, and prove that it has this property.

N12 We can always color the nodes of a graph such that no two adjacent nodes are the same color. The **chromatic number** of a graph is the minimum number of colors needed to do so. For a graph G, prove that these conditions are equivalent:

(a) G is bipartite.

(b) [2★] G has chromatic number 2.

(c) [5★] G is connected, and for every node a and edge bc, $d(a,b) \neq d(a,c)$.

(d) [8★] G has no cycles of odd length.

N13 [3★] Prove that a forest with n nodes has c connected components if and only if it has $n - c$ edges.

N14 [13★] Suppose that, for the union find problem, all of the unions are performed before all of the finds. Suppose there are q operations. Show how to solve this problem in $\mathcal{O}(n+q)$ time.

(The find operation can return any element in the subset, but it must return the same thing for every element in the subset. This shows how asking queries in between adds to the difficulty of the union-find problem.)

N15 [13★] Construct a worst-case scenario for union find without any optimizations. That is, for any positive integer q, give a series of q operations with a running time of $\Theta(qn)$, and prove that it has this running time.

8.3 Coding problems

For each **S** category except the last, you'll get a [5★] bonus for the first problem that you solve after not having thought about it for a year.

First timers, solve at least [60★]. Many of the red problems are classics, so you're strongly encouraged to **solve at least one red problem per S category.**[6] Note that if you solve the easiest red problem in each **S** category, you'll get [51★].

Veterans, solve at least [100★]. Strongly suggest you solve, or at least *try* to solve, all the problems in the last **S** category: they are worth [61★] in total.

C1 [5★] **LoL Tournament:** Problem L in https://cjquines.com/files/icpcmanila2016.pdf

C2 [5★] Write a program that takes in an integer n and generates a random tree with n nodes. It should output $n-1$ lines, each with two space-separated integers denoting an edge of the tree. Number the nodes $1, 2, \ldots, n$.

C3 [8★] Problem **N4c** easily leads to an algorithm that, given a sequence of positive integers a sequence of non-negative integers $d_1 \geq d_2 \geq \ldots \geq d_n$, outputs a simple graph with n nodes where the ith node has degree d_i, or states that no such graph exists.

This algorithm is called the **Havel–Hakimi algorithm**. Implement it. It should take in the sequence of positive integers from input in however way you want, and output the graph by printing its adjacency matrix.

These problems all involve graph traversal and its applications.

S1 [2★] **[2017] Never Forget the C:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2017-never-forget-the-c

S2 [3★] **Kuro and Walking Route:** https://codeforces.com/problemset/problem/979/C

S3 [3★] **Fox and Names:** https://codeforces.com/problemset/problem/510/C

S4 [5★] **Yumamma II:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/yumamma-ii

S5 [8★] **Backup Bridges:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/backup-bridges

S6 [8★] **Dima and Horses:** https://codeforces.com/problemset/problem/272/E

S7 [8★] **Labelling Cities:** https://codeforces.com/problemset/problem/794/D

These problems involve trees: tree traversal, rooted trees, operations on trees, and so on. While tree problems can fall under many categories, like tree DP, data structures, centroid decomposition, and so on, these problems don't require any complicated algorithms.

S8 [2★] **Basic Tree Reconstruction 2:** https://www.hackerrank.com/contests/noi-ph-2015-camp-cramers/challenges/basic-tree-reconstruction-2

[6] By classics, I mean they're the kind of problem you can bring up in a conversation and people will remember it. LoL Tournament, for example, is a problem I recognized even after having thought about it for a year. Many NOI.PH graph problems are also classics in this sense.

S9 [2★] **Basic Tree Reconstruction 3:** https://www.hackerrank.com/contests/noi-ph-2015-camp-cramers/challenges/basic-tree-reconstruction-3

S10 [3★] **Full Binary Tree Queries:** https://codeforces.com/problemset/problem/960/D

S11 [5★] **Maximal Tree Diameter:** https://www.hackerrank.com/contests/hourrank-19/challenges/maximal-tree-diameter

S12 [5★] **Chef and Trees:** https://www.codechef.com/problems/MAXDIST

S13 [8★] **Tree Cake:** https://www.codechef.com/problems/TREECAKE

S14 [8★] **Afraid of the Dark:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/afraid-of-the-dark

S15 [8★] **TreeLand Journey:** https://www.codechef.com/problems/JRNTREE

S16 [13★] **Product of Diameters:** https://www.codechef.com/problems/TREEDIAM

These problems all involve union find and its multiple variants.

S17 [2★] **In or Out:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/in-or-out

S18 [3★] **War:** UVa 10158

S19 [3★] **Cargo Train to Busan:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/cargo-train-to-busan

S20 [5★] **Almost Union-Find:** UVa 11987

S21 [8★] **The Tide is High:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/the-tide-is-high

S22 [8★] **Carps?:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/r-u-g

S23 [13★] **A Little Bit Frightening:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/in-fact-it-was

---

These problems are all *constructive*, in the sense that you need to find the construction that gives what you want. They all feel "mathematical" in nature, in that they can easily appear in a math contest.

S24 [3★] **Cutting Figure:** https://codeforces.com/problemset/problem/193/A

S25 [3★] **Maze:** https://www.codechef.com/problems/MAZE

S26 [5★] **Mister B and Flight to the Moon:** https://codeforces.com/problemset/problem/819/E

S27 [8★] **BiCycles:** https://www.codechef.com/problems/BCYCLES

S28 [13★] [2018] **The Wedding Guests:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2018-the-wedding-guests

---

These problems are also constructive, but in a different way: it's asking you to construct a *graph* with certain properties. They have roughly the same "feel" in that they share many of the same main ideas.

S29 [3★] **Graph And Its Compliment:** https://codeforces.com/problemset/problem/990/D

S30 [5★] **A Mist of Florescence:** https://codeforces.com/problemset/problem/989/C

S31 [5★] **Fox and Minimal path:** https://codeforces.com/problemset/problem/388/B

S32 [8★] **Chefs and Voting for best friend:** https://www.codechef.com/problems/CHEFVOTE

S33 [13★] **Mr. Kupido:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/mr-kupido

S34 [13★] **PWNED BY ANONYMOUS PH:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/mutual-friendzone

---

These problems are a bit hard to categorize, but they are my favorite problems in this set. They are all fun to think about, all unique, and all very "mathematical". These problems definitely belong in the Graph 1 module, and can be solved with the concepts introduced here.

S35 [5★] **Network Connections:** UVa 793

S36 [5★] **Mutual Friendzone:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/mutual-friendzone-not-hacked

S37 [8★] **[2015] Rigid Trusses:** https://www.hackerrank.com/contests/noi-ph-2019-preselection/challenges/2015-rigid-trusses

S38 [13★] **Historical Junctions:** https://www.codechef.com/problems/HISTJUNK

S39 [13★] **We Are Number One:** https://www.hackerrank.com/contests/noi-ph-practice-page/challenges/lazytown-4

S40 [17★] **Suns and Rays:** https://codeforces.com/problemset/problem/316/F3

Acknowledgments

Much of the later material is essentially copied off of the 2018 material from Week 3, so thanks to Kyle and Hadrian for writing that! Thanks to Kevin for help with the problem set!