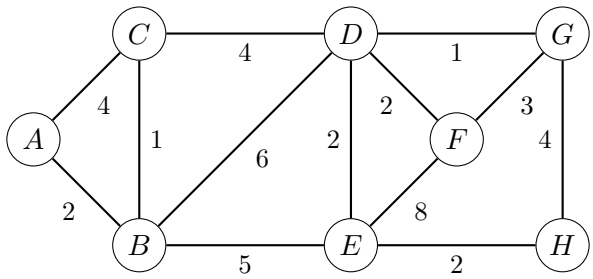
1 Weighted Graphs

A **weighted** graph is a graph $G = (V, E)$, where each edge e has a cost c_e associated with it. We typically draw this by writing c_e above the relevant edge

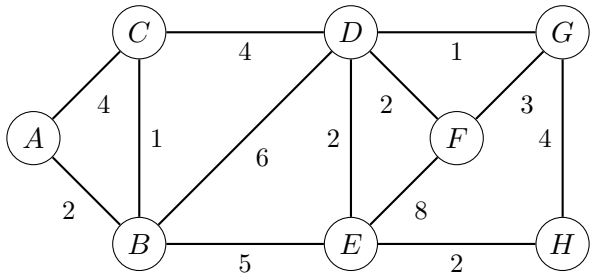
We say a path has **cost** equal to the sum of the costs of the edges in the path.

Question 1. What is the cost of the path $B - E - D - G$ in the following graph?

Question 2. What is the shortest (lowest-cost) path from A to D in the following graph?



1.1 Dijkstra's Algorithm



Dijkstra's Algorithm takes as input a weighted graph with positive edge weights and a designated vertex s to signify the "starting point." The output is a **tree** such that the unique path from s to any other vertex v in the tree is the shortest (lowest-cost) path in the original graph.

```
Dijkstra's (Single-Source, Shortest Path Tree) Algorithm
  for each vertex v do
    intree(v) = false
    parent(v) = N/A
    dist(v) = ∞
  dist(s) = 0
  while ∃ vertex u with intree(u) = false do
    u ← vertex with intree(u) = false and smallest dist(u)
    intree(u) = true
    for each vertex v ∈ adj[u] do
      if dist(v) > dist(u) + w(u, v) then
        dist(v) = dist(u) + w(u, v)
        parent(v) = u
```

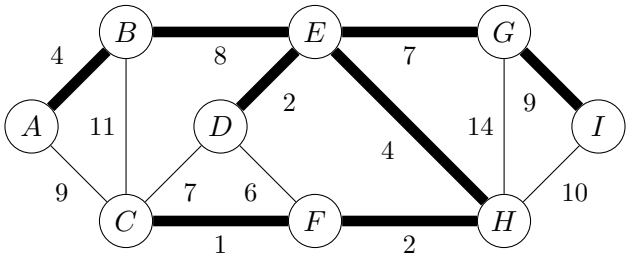
This algorithm keeps track of three pieces of information for each vertex: whether or not it's in the shortest path tree so far, which vertex either is its parent in that tree, or would be if it were added right now (using the best path found so far), and what the distance from the start vertex to it is (along edges viewed so far).

v	intree(v)	parent(v)	dist(v)
A		N/A	0
B			
C			
D			
E			
F			
G			
H			

2 Prim's Algorithm and Minimum Spanning Trees

Given a graph G, a **spanning tree** is a subgraph of G which is a tree containing every vertex of G.

The Minimum Spanning Tree problem is: Given a connected undirected graph $G = (V, E)$ with edge weights, find a subset of the edges which form a tree on the original nodes, such that the sum of the edge weights chosen is minimized. Here is a graph with a minimum spanning tree highlighted:

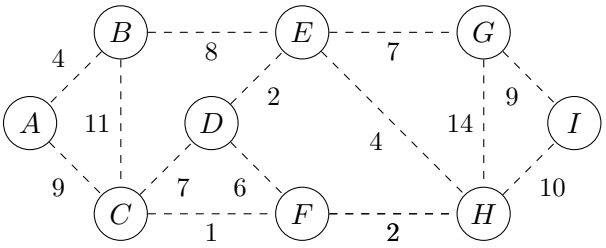


Note that while every connected weighted graph has a Minimum Spanning Tree, it is not the case that every connected weighted graph has a *unique* minimum spanning tree.

Question 3. Think about the edge (E, G). It is highlighted in the example. Could a correct minimum spanning tree for the graph *not* include that edge? Does this suggest an algorithm for finding the minimum spanning tree?

This demonstrates the **cut property** of Minimum Spanning Trees; for any subset of vertices S, let e be the minimum cost edge with exactly one endpoint in S. The Minimum Spanning Tree must contain e.

Using the cut property, identify which edges in the following graph constitute a minimum spanning tree.



v	intree(v)	parent(v)	dist(v)
A	F	N/A	0
B	F		∞
C	F		∞
D	F		∞
E	F		∞
F	F		∞
G	F		∞
H	F		∞
I	F		∞

3 Unweighted Interval Scheduling

Let's revisit a problem from the previous unit, with a slightly different approach: Fed up after CompSci 161, your friend has decided to change majors to one that grades based only on attendance. The only question is which classes to take next quarter? They all meet once a day at different times, *but are worth the same credits each.* Your friend's goal is to maximize the number of classes taken in the quarter without having to skip any lectures.

Problem Statement: we are given a set of n intervals, each of which has a start time s_i and a finish time f_i. Our goal is to select as large of a subset of the intervals such that no two selected intervals overlap.

We could call the algorithm from a few weeks ago, with $\forall_i\ v_i = 1$, but let's see if we can find a different way to solve this. It's possible that a simpler algorithm exists than the dynamic programming solution.

In fact, **one of the following algorithms will get the correct answer**. Decide which ones *don't work* by providing counter-examples. Don't worry (yet) about proving one that is correct.
Please note: for the homework and exam, you *do not* need to provide "not working" algorithms, and showing that other algorithms do not work *does not* demonstrate that yours does. The purpose of this part is to examine candidate algorithms and to think about the problem.

- Sign up for the class that begins earliest. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

- Sign up for the class that meets for the least amount of time. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

- Sign up for the class that conflicts with the fewest other classes. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

- Sign up for the class that ends earliest. Remove it and all overlapping classes from the set of available classes. Repeat this process until no classes remain.

Any of the above algorithms can be implemented in time $\mathcal{O}(n \log n)$ – a good exercise would be for you to write pseudo-code for it as part of your review. Unlike the dynamic programming algorithm, the correctness of this algorithm isn't as easy to see from the description. Right now, the "proof" that it is correct relies on that I told you one of the four was correct, and you've seen that the other three *aren't* correct. However, such a proof isn't valid.

Proving correctness: once we have an algorithm we believe is correct, we need to prove that it is. Each of the above algorithms can be described as "select some interval, remove conflicting intervals, and recursively solve the problem on the rest." We would like to prove that there is an optimal solution that includes the first interval selected.

Claim: There is an optimal solution that includes the first interval that we choose.
Note that I am not claiming that all optimal solutions do.

Would any optimal solution that includes our first interval also include any intervals that overlap with it?

What is left to do to prove that the rest of our algorithm is correct?

4 Scheduling with Deadlines

Let's examine a different algorithm for scheduling intervals. In the last lecture, each interval had pre-designated start and end times. Instead, let's consider a problem where each interval i is a task that must be completed; each has a designated time t_i, but we can designate any start time for it. Each interval also has a deadline d_i, which can be different for each interval.

We must assign each interval a start time in such a way that no two intervals overlap. Ideally, we would like to schedule everything to be finished before its deadline, but this is not always possible. We say the lateness l_i of a job is how late it is finished compared to its deadline, $s_i + t_i - d_i$, or 0 if it has been completed by the deadline. Our goal is to minimize the *maximum* lateness: the amount by which the most late job exceeds its deadline.

Examples: What is the optimal schedule for each of the following?

Example 1:

Time	1	2	3
Deadline	2	4	6

Example 2:

Time	1	2	3	4
Deadline	2	4	6	6

Some Possible Algorithms One of the following algorithms will correctly schedule the tasks. Decide which one you think it is, and show that the other two *do not* correctly schedule these.

- Sort the jobs by increasing time t_i; schedule them in that order.
- Sort the jobs by $d_i - t_i$; schedule them in that order.
- Sort the jobs by deadline d_i; schedule them in that order.

Proof of Correctness

Since every schedule (optimal or otherwise) includes every task, we cannot follow the same model proof as the covering/packing problems from last lecture. Instead, I will first cover two lemmas[1], and then I will use those to prove the overall theorem.

[1]You can think of a lemma as a "helper proof" – a statement that requires a proof for itself, but the overall statement is then used in your proof.

Lemma 1. When deciding start times, don't leave any gaps; $s_{i+1} = s_i + t_i$.

Lemma 2. Any schedule that doesn't agree with our algorithm has at least one pair of *consecutive* intervals $i, i+1$ that are *inverted* relative to our order.

We can now proceed to the full proof; we claim our output is a global optimal with this claim:

Claim: Any schedule with an inversion (relative to our output) can be modified to be more like our algorithm's output without making it worse.

5 Interval Coloring

Suppose your friend is working at the library and is in charge of allowing groups into the various study rooms. For this problem, all study rooms are interchangable. A total of n groups have requested to use a study room tonight; group i would like to use it from s_i to f_i. If two groups overlap in their request times, they cannot be placed in the same study room; furthermore, we cannot reject a group. Fortunately, we have a very large library with an infinite number of study rooms, although we would prefer to not use all of them if possible.

Give an efficient algorithm that will assign each group to a room (the rooms are numbered $1, 2, 3, \ldots$) in such a way that the number of rooms you use is minimized, and no two groups that overlap are assigned to the same room. Explain as best you can why your algorithm achieves the optimal number of rooms (we'll talk about how to formally prove this).

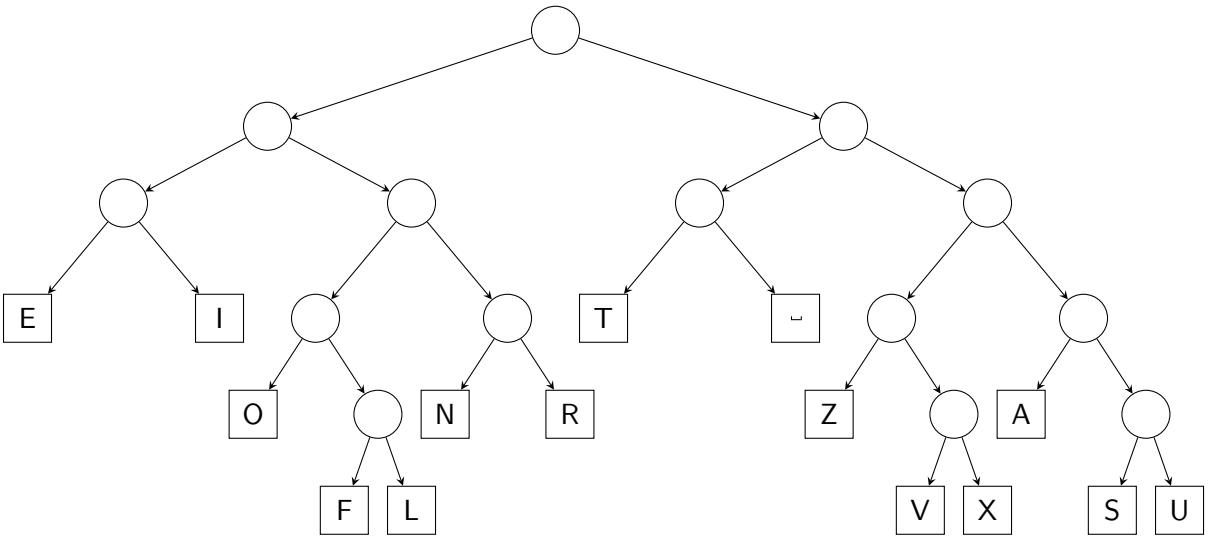
6 Huffman Trees

Question 4. Computers read bits, not characters. We translate each character to a string of bits which a computer can understand. First, think about what would be a good encoding. Then, identify problems and inefficiencies with the following encodings.

- a = 00000, b = 00001, c = 00010, ..., z = 11001
- a = 0, b = 1, c = 00, d = 01, e = 10, etc
- a = 00000, b = 00001, ..., v = 10101, w = 1100, x = 1101, y = 1110, z = 1111

Question 5. How could we use a binary tree to represent an encoding?

Consider the following prefix code tree:



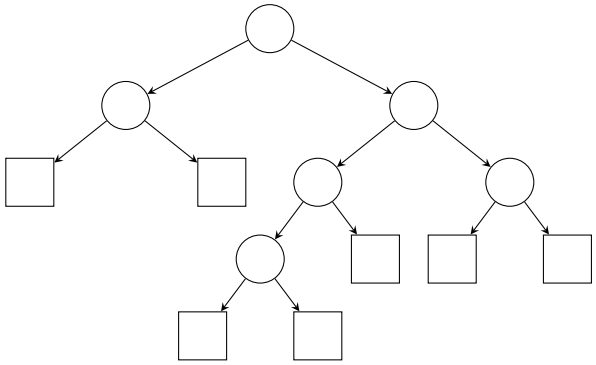
Question 6. What is the message that would be compressed as 110001001001011100010100, assuming a left-child to means '0' and a right-child means '1'?

Question 7. How would you encode the message "ANTEATERS" using the above tree?

Today's problem is as follows. We are given a text document; each letter i has a frequency $0 \leq f_i \leq 1$, with $\sum_i f_i = 1$. Our goal is to encode the text in such a way that each letter's code is a prefix of another code. Let b_i be the length of the encoding for letter i. Our goal is to find a valid encoding that minimizes $\sum_i f_i b_i$.

Suppose we want to encode a document with a prefix code that has the following frequencies, and further suppose we have to do so via the following tree (where the leaf nodes are represented as boxes):

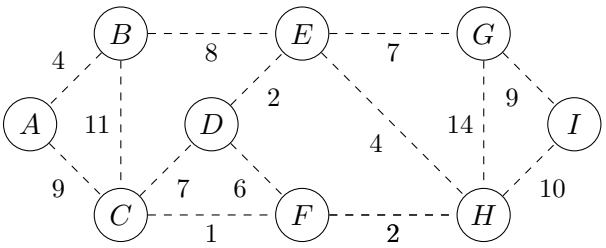
letter x	frequency f_x
F	21%
I	18%
A	6%
T	5%
L	23%
U	12%
X	15%



Question 8. Where should the letters go in order to minimize the average bit length of a compressed message?

Question 9. What would a Huffman tree for "`engineering useless rings`" be?

7 Kruskal's Algorithm



7.1 Union-Find Data Structure

When we ran Kruskal's Algorithm for Minimum Spanning Trees, we had the following sequence of edge considerations:

Edge	cost	Result
CF	1	keep
ED	2	keep
FH	2	keep
EH	4	keep
AB	4	keep
DF	6	reject
CD	7	reject
EG	7	keep
BE	8	keep
AC	9	reject
GI	9	keep
HI	10	reject
BC	11	reject
GH	14	reject

The graph was small enough that, as humans, we could look at it and see if the end-points were already in the same connected component. If we wanted to write this as a computer program, we need a way to track the following information:

1. Initially, all vertices are their own connected component.
2. We need to check efficiently if two vertices are in the same connected component, even if those two vertices were never explicitly merged
3. We need to efficiently merge two connected components, and all the vertices therein.