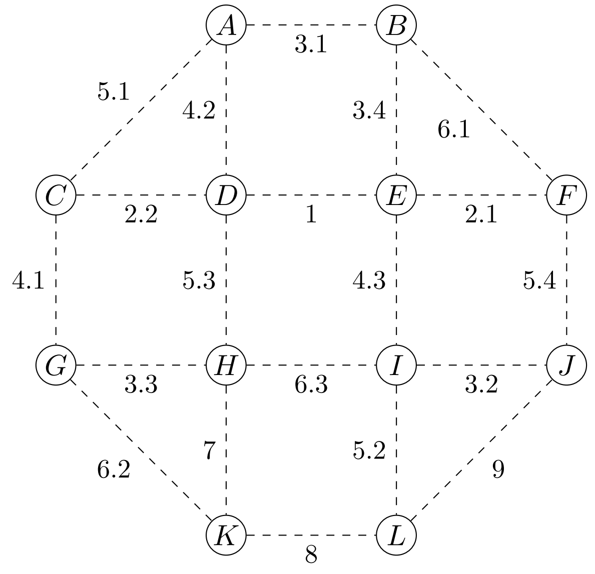
1 Jarnik's Algorithm and Minimum Spanning Trees

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

The Minimum Spanning Tree problem is: Given a connected undirected graph $G = (V, E)$ with positive 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. For purposes of CompSci 161, unless stated otherwise, we will assume all edge weights are distinct.

Something you might find interesting about the MST problem: there is a deterministic algorithm for this (by Pettie and Ramachandran in 2002) whose running time is optimal, but we do not know what that running time is.



Jarnik's Algorithm:

- Choose an arbitrary starting vertex s
- Build output tree T one edge at a time.
  - At start, one-vertex tree only s
- Until we have a tree, add an edge:
  - Edge must connect T to $G - T$
  - Select min-weight edge e that does so.
  - Add e to T.

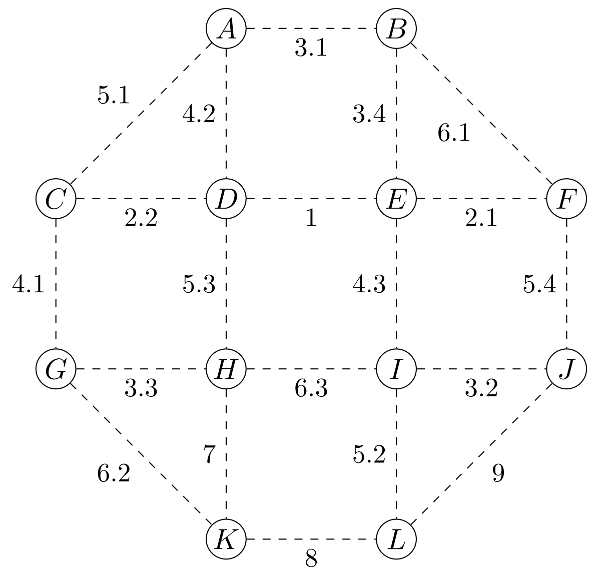
1.1 Correctness of Jarnik's Algorithm

Some lemmas apply to other MST algorithms, too.

Question 1. If C is any cycle and e is its heaviest edge, then any tree T that includes e cannot be a minimum spanning tree. This is the **cycle property** of Minimum Spanning Trees.

Question 2. If we cut the vertices of the graph into any two (non-empty) subsets X and $G - X$, and e is the lightest edge with endpoints in both subsets, then any tree T that avoids e cannot be the minimum spanning tree. This is the **cut property** of Minimum Spanning Trees.

Let's go through a run of Jarnik's Algorithm again and discuss why *every edge chosen is correct.*



Jarnik's Algorithm:

- Choose an arbitrary starting vertex s
- Build output tree T one edge at a time.
  - At start, one-vertex tree only s
- Until we have a tree, add an edge:
  - Edge must connect T to $G - T$
  - Select min-weight edge e that does so.
  - Add e to T.

Question 3. For any two vertices x and y, the path from x to y through the minimum spanning tree has the minimum possible weight for its heaviest edge. This is the **path property** of Minimum Spanning Trees.

2 Unweighted Interval Scheduling

Reading: Erickson §4.2, G/T §10.2

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.

Suggested reinforcement after lecture: take your notes and attempt to write this proof as if it were a homework question. It will look different from how it was presented in lecture! This is a good exercise, similar in concept to warm-up 1.

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?

3 Scheduling with Deadlines

Reading: Erickson §4.1 is a similar problem. A good exercise for you is to read the book chapter up until the statement of Lemma 4.1, then try to prove that lemma. When you complete that, read the proof and then attempt 4.2.

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.

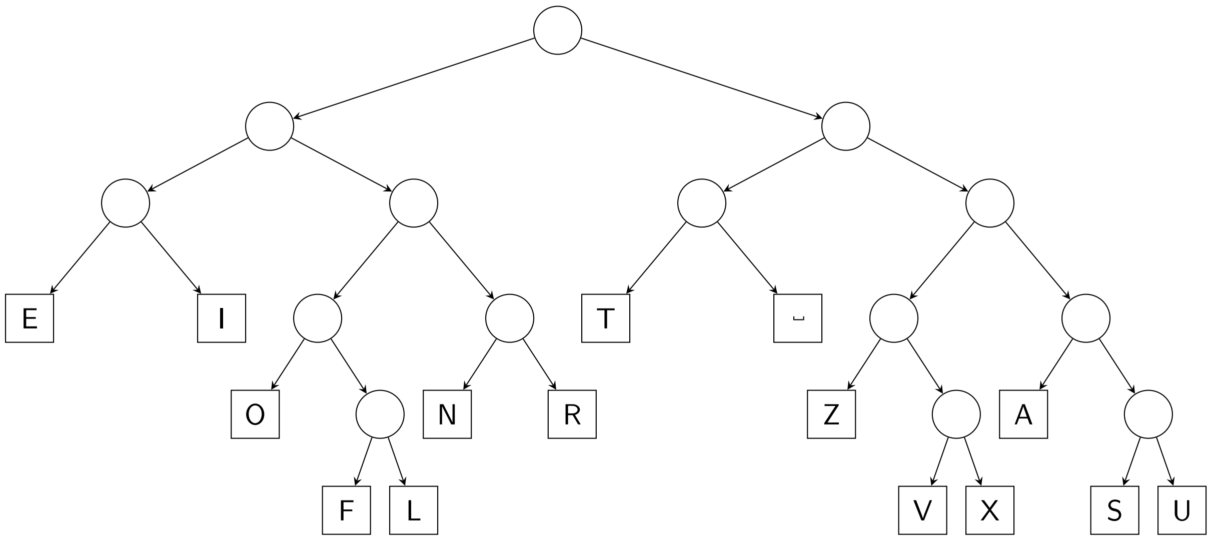
We can now proceed to the full proof; we claim our output is a global optimal with this claim:

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.

4 Huffman Trees

Reading: Erickson §4.4, G/T 10.3

Consider the following prefix code tree:

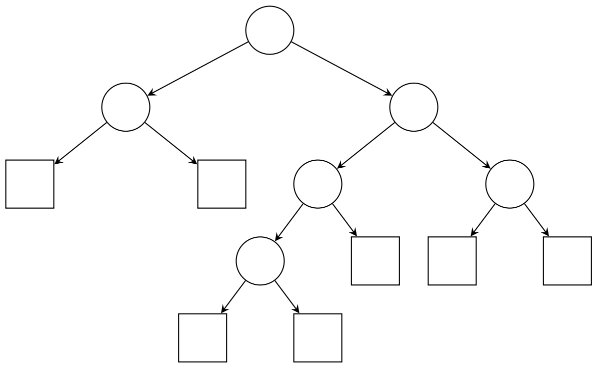


Question 4. What is the message that would be compressed as 1100010010010111000100100, assuming a left-child to means ‘0’ and a right-child means ‘1’?

Question 5. How would you encode the message “ANTEATERS” using the above tree?

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

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 $\sum_i f_i b_i$.

We will discuss how to build a tree that does this.

Question 7. All non-leaf nodes in the optimal tree have exactly two children. Why is this true?

Question 8. The two characters with minimum frequency should be at maximum depth. Why is this true?

Question 9. What algorithm is suggested by the above two lemmas?

Question 10. Draw a Huffman tree for the phrase "`engineering useless rings`"

5 Fractional Knapsack

Reading: G/T §10.1

Consider the general Knapsack problem: we have a set S of n items; each item has a positive benefit b_i and a positive weight w_i. We can carry at most some bound W, and we wish to take some (or all) of the items to gain the maximum possible benefit, subject to the constraint that the total weight we take is at most W.

During a discussion section during the Dynamic Programming portion of class, you saw an algorithm to solve a version of this problem called the **0-1 Knapsack Problem**. In this, each item had to either be taken or not taken; this represents that the items are things like your laptop, where having half a laptop is not worth half the value of a full laptop.

In today's problem, we can take a **fraction** of each item. That is, for each item, we decide some amount x_i to take, up to w_i. That is, we take up to the full amount of each item, which are presumed to be infinitely divisible – a not-perfect representation for things like food and beverage or the dust of valuable metals. This also assumes that the value is linear – having half of the available material is worth half the total value, one-quarter of the available material is worth one-quarter of its total value, and so on.

More formally, decide for each item a value x_i with $0 \leq x_i \leq w_i$ such that $\sum_i x_i \leq W$ with the goal of maximizing the sum $\sum_i b_i(x_i/w_i)$.

Example: Suppose we can carry $W = 10$ and have the following items available to us:

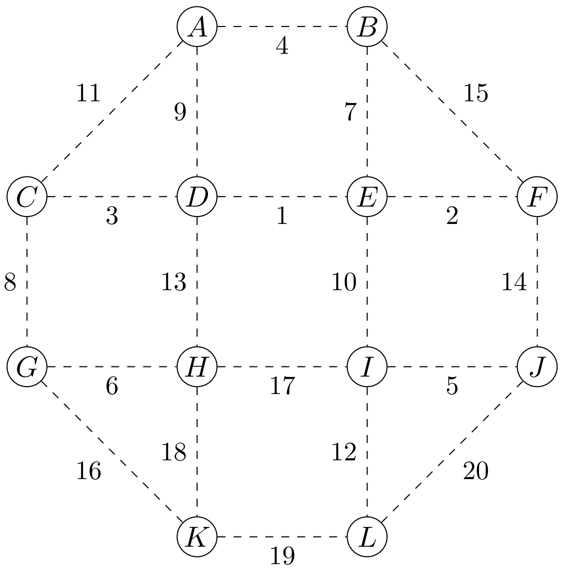
Item :	1	2	3	4	5
Weight:	4	8	2	6	1
Benefit:	12	32	40	30	50

Problem: find an efficient algorithm that will allow us to, given the general input, will find the optimal fractional knapsack solution. Prove that your algorithm finds the optimal amount.

6 Kruskal's Algorithm and the Union-Find Data Structure

Kruskal's Algorithm proceeds as follows.

```
Each vertex is a tree
for each edge $e = (u, v)$, non-decreasing do
    if $e$ connects two different trees then
        accept edge $e$
    else
        reject edge $e$
```



6.1 Union-Find Data Structure

If we run Kruskal's Algorithm for the above graph, it is small enough that, as humans, we can 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:

We will need an efficient data structure with two functions:

1. Given a vertex, in which tree is it?
2. Given two vertices in different trees, connect their trees.

This structure's constructor will need to start at the state of n disjoint trees of a single node each.

7 Greedy Algorithms on Trees

The Maximum Matching problem is as follows. You are given an undirected graph $G = (V, E)$. A *matching* is a subset of the edges such that each *vertex* is included at most once. One way to think of this is if the vertices are students, the edges are pairs of willing partners, and the goal is to create teams of size two. Students may work on their own, but no one can have two partners. A maximum matching[2] is the largest number of edges from an input graph you can select without a vertex being repeated.

In any case, suppose our input graph *is an undirected tree*. It might not be a binary tree. Give an efficient algorithm to compute the maximum matching. Justify the correctness of your algorithm.

The Maximum Independent Set problem is as follows. Given an undirected graph $G = (V, E)$, a subset $I \subseteq V$ of the vertices is called an *independent set* if no two vertices in set I share an edge. One way to picture this is vertices are people, edges represent conflicts, and an independent set is a party[3] that doesn't cause an argument or a fight. In general, finding the maximum independent set in a graph is a difficult problem. However, suppose our input graph *is a tree*. Give an efficient algorithm to compute the maximum independent set in a tree. Justify the correctness of your algorithm.

[2]In general, there are some (I think) neat algorithms for matching. If this quarter were 13 weeks instead of 10, or if we had four hours a week of lecture instead of three, the fourth topic would probably be network flow (chapters 10 and 11 of the Erickson textbook and chapter 16 of the G/T textbook), which can be applied to matching problems. One general algorithm for matching has a running time of $\mathcal{O}(m\sqrt{n})$, which is not an expression you see very often.

[3]It's always a party in CompSci 161

8 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).