Week 14 - Monday

COMP 4500

Last time

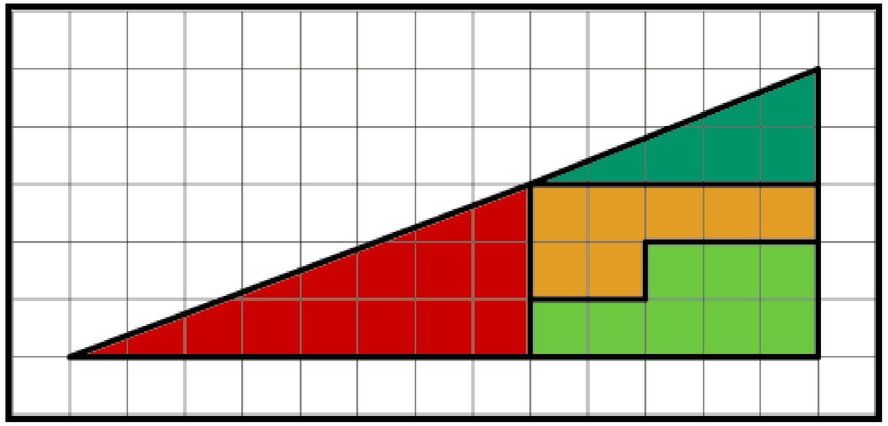
- What did we talk about last time?
- Approximation algorithm for knapsack

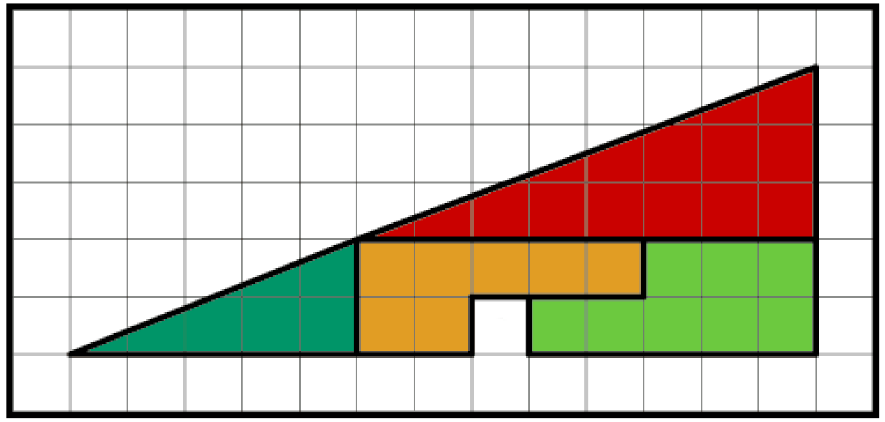
Questions?

Assignment 7

Logical warmup

- Consider the following shape to the right:
- Now, consider the next shape, made up of pieces of exactly the same size:
- We have created space out of nowhere!
- How is this possible?





Review

Final exam

- Final exam:
  - Wednesday, April 24, 2024
  - 8:00 – 10:00 a.m.
- It will mostly be short answer
- There will be diagrams
- There might be a matching problem
- There will likely be a (simple) proof
- It will be 50% longer than previous exams, but you will have 100% more time

Big Oh

- Order the following functions by rate of growth:
  - $(\sqrt{2})^{\log n}$
  - n^2
  - $n!$
  - $(\log n)!$
  - $\left(\frac{3}{2}\right)^n$
  - n^3
  - $(\log n)^2$
  - $\log(n!)$
  - 2^{2^n}
  - $n^{\frac{1}{\log n}}$
  - $\log(\log n)$
  - $n \cdot 2^n$
  - $n^{\log(\log n)}$
  - $\log n$
  - 1
  - $2^{\log n}$
  - $(\log n)^{\log n}$
  - $4^{\log n}$
  - $(n+1)!$
  - $\sqrt{\log n}$
  - n
  - 2^n
  - $n \log n$

Big Oh

- Determine the running time of various loops
- Example:

```
int counter = 0;
for(int i = 0; i*i < n; ++i)
   for(j = 1; j <= n; j *= 2)
        counter++;
```

Graphs

- Know basic definitions of graphs
  - Nodes
  - Edges
  - Directed vs. undirected
  - Adjacency matrix vs. adjacency lists
  - Trees
  - Connected
  - Strongly connected

Graph algorithms to know

- BFS
- DFS
- Determining bipartiteness
- Find connected component
- Find strongly connected component
- Topological sort

Example proof

- For all $\boldsymbol{n} \in \mathbb{Z}$, if $\boldsymbol{n}^2 + 7$ is odd, then $\boldsymbol{n}$ is even.
- **Hint:** Try a proof by contradiction.

Example proof

- Prove that, for all integers $n \geq 1$,

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$$

- **Hint:** Try a proof by induction.

Example proof

- Prove that a graph with two or more nodes where each node has a degree of ***n***/2 or higher must be connected.
- **Hint:** Try a proof by contradiction.

Interval Scheduling

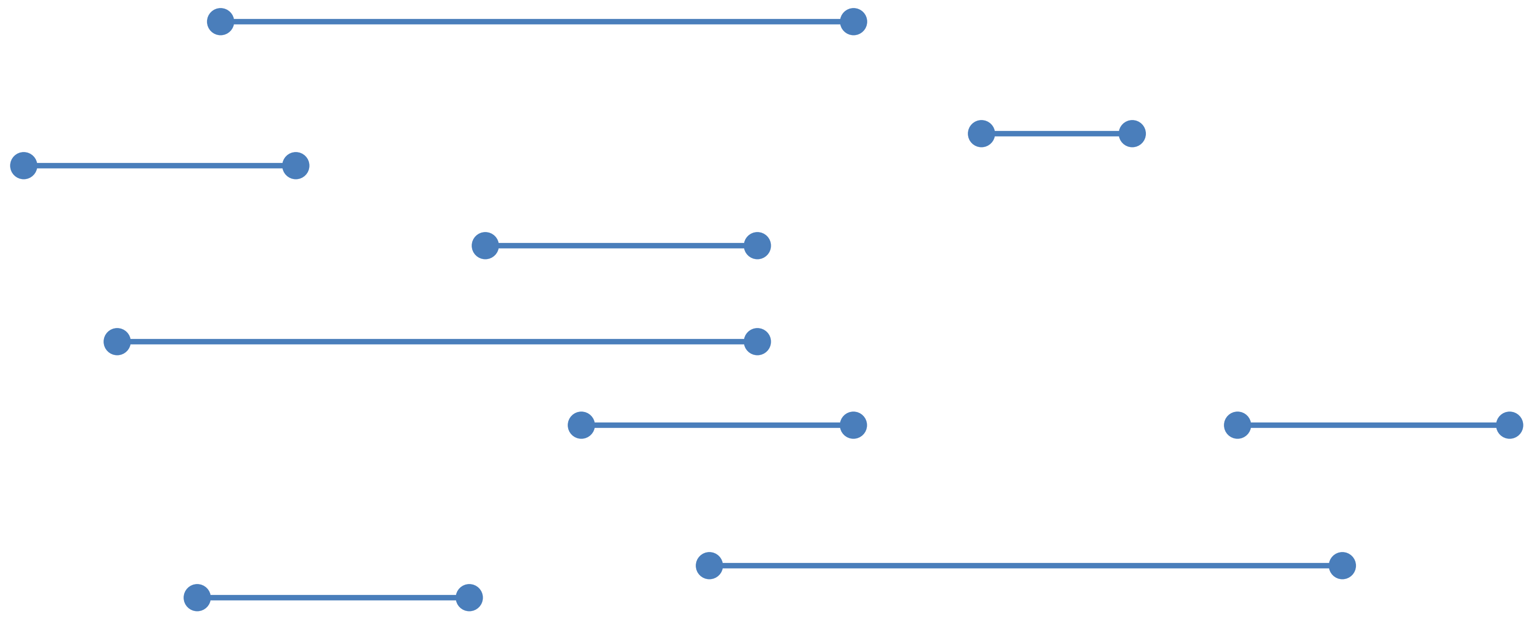
Interval scheduling

- In the interval scheduling problem, some resource (a phone, a motorcycle, a toilet) can only be used by one person at a time
- People make requests to use the resource for a specific time interval [*s*, *f*]
- The goal is to schedule as many uses as possible
- There's no preference based on who or when the resource is used

Interval scheduling algorithm

- Interval scheduling can be done with a greedy algorithm
- While there are still requests that are not in the compatible set
  - Find the request *r* that ends earliest
  - Add it to the compatible set
  - Remove all requests *q* that overlap with *r*
- Return the compatible set

Interval scheduling example



Running time

- First, we sort the ***n*** requests in order of finishing time
  - The best comparison-based sort takes O(***n*** log ***n***)
- We scan through the ***n*** sorted requests again and make an array ***S*** of length ***n*** such that ***S***[***i***] contains the starting value of ***i***, ***s***(***i***)
  - O(***n***) time
- Our algorithm selects the first interval in our list sorted on finishing time. We then move through array ***S*** until we find the first interval ***j*** such that ***s***(***j***) ≥ the finishing time selected. We add it. We continue the process until we have moved through the entire array ***S***.
  - O(***n***) time
- Total time: O(***n*** log ***n***)

Shortest Paths

Shortest path set up

- Directed graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with start node $\boldsymbol{s}$
- Assume that there is a path from $\boldsymbol{s}$ to every other node (although that's not critical)
- Every edge $\boldsymbol{e}$ has a length $\boldsymbol{l_e} \geq 0$
- For a path $\boldsymbol{P}$, length of $\boldsymbol{P}$ $\boldsymbol{l}(\boldsymbol{P})$ is the sum of the lengths of the edges on $\boldsymbol{P}$
- We want to find the shortest path from $\boldsymbol{s}$ to every other node in the graph
- An undirected graph is an easy tweak

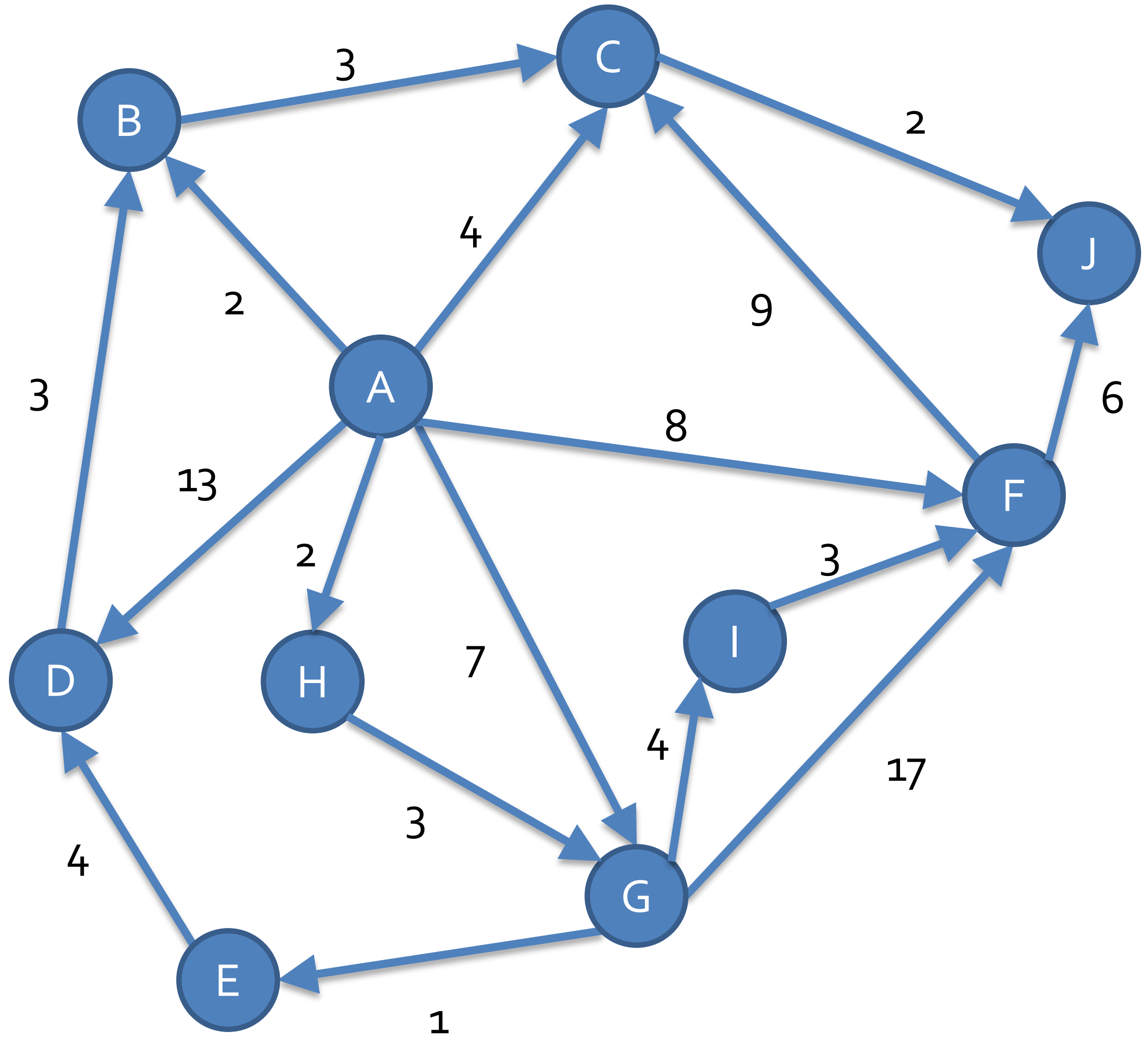
Designing the algorithm

- Let's first look at the **length** of the paths, not the actual paths
- We keep set ***S*** of vertices to which we have determined the true shortest-path distance
  - ***S*** is the explored part of the graph
- Then, we try to find the shortest new path by traveling from any node in the explored part ***S*** to any node ***v*** outside
- We update the distance to ***v*** and add ***v*** to ***S***
- Then, continue

Dijkstra's Algorithm

- Let $\boldsymbol{S}$ be the set of explored nodes
  - For each $\boldsymbol{u} \in \boldsymbol{S}$, we store a distance $\boldsymbol{d}(\boldsymbol{u})$
- Initially $\boldsymbol{S} = \{\boldsymbol{s}\}$ and $\boldsymbol{d}(\boldsymbol{s}) = 0$
- While $\boldsymbol{S} \neq \boldsymbol{V}$
  - Select a node $\boldsymbol{v} \notin \boldsymbol{S}$ with at least one edge from $\boldsymbol{S}$ for which $\boldsymbol{d'}(\boldsymbol{v}) = \min_{e=(u,v):u\in S} \boldsymbol{d}(\boldsymbol{u}) + \boldsymbol{l_e}$ is as small as possible
  - Add $\boldsymbol{v}$ to $\boldsymbol{S}$ and define $\boldsymbol{d}(\boldsymbol{v}) = \boldsymbol{d'}(\boldsymbol{v})$

Dijkstra's Algorithm Example



Reflections on Dijkstra's algorithm

- You can think of Breadth-First Search as a pulse expanding, layer by layer, through a graph from some starting node
- Dijkstra's algorithm is the same, except that the time it takes for the pulse to arrive is based not on the number of edges, but the lengths of the edges it has to pass through
- Because Dijkstra's algorithm expands from the starting point to whatever is closer, it grows like a blob
- There are algorithms that, under certain situations, can cleverly grow in the direction of the destination and will often take less time to find the path there

Minimum Spanning Trees

Minimum spanning tree

- We have a weighted, connected graph and we want to remove as many edges as possible such that:
  - The graph remains connected
  - The edges we keep have the smallest total weight
- This is the **minimum spanning tree** (MST) problem
- We can imagine pruning down a communication network so that it's still connected but only with the cheapest amount of wire total
- MST algorithms are also used as subroutines in other graph problems

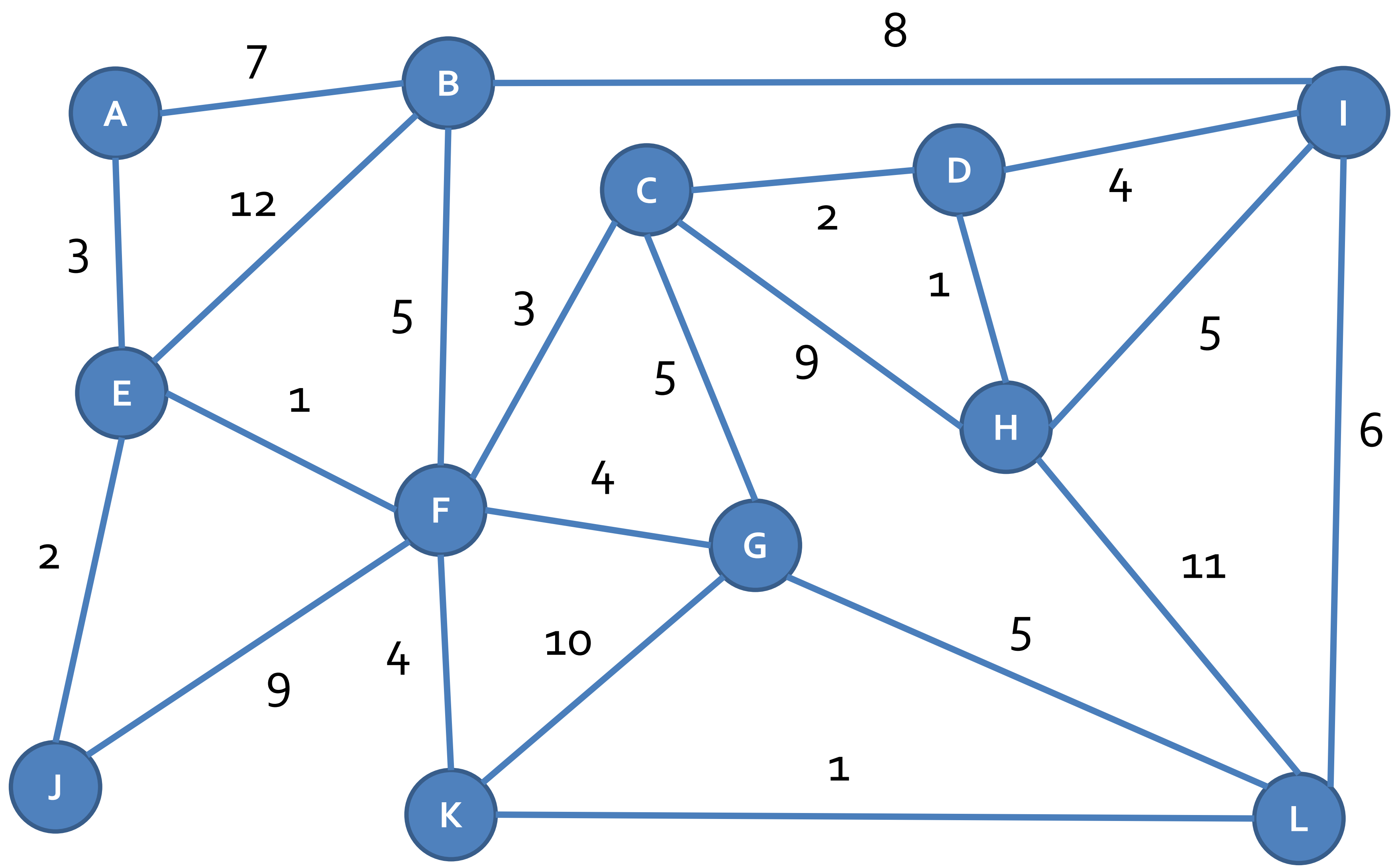
MST observations

- Assuming positive edge weights, the resulting graph is obviously a tree
  - If the graph wasn't connected, it wouldn't be a solution to our problem
  - If there was a cycle, we could remove an edge, make it cheaper, and still have connectivity

Approaches

- **Kruskal's algorithm:** Add edges to the MST in order of increasing cost unless it causes a cycle
- **Prim's algorithm:** Grow outward from a node, always adding the cheapest edge to a node that is not yet in the MST
- **Backwards Kruskal's algorithm:** Remove edges from the original graph in order of decreasing cost unless it disconnects the graph
- All three algorithms work!

MST example



Clustering

Clustering

- Imagine you have a set of objects
  - Photographs
  - Documents
  - Microorganisms
- You want to classify them into related groups
- Usually, you have some **distance function** that says how far away any two objects are
- You want to group together objects so that all the objects in a group are close

Notes about distance

- The distance function is usually defined between all points
  - If the points are in the plane or another Euclidean space, the distance could simply be the distance between them
  - A more flexible way to define distance is as weights on graph edges in a complete graph
- The distance between a point and itself is 0
- The distance between any two distinct points is greater than 0
- The distance between two points is symmetrical

Clustering by maximum spacing

- What if we want to divide our objects into k non-empty sets:
  - $C_1, C_2, \ldots, C_k$
- The **spacing** of this ***k***-clustering is the minimum distance between any pair of points in different clusters
- We want to find clusters with maximum spacing
  - There are other metrics to optimize your clusters on

Algorithm

- We don't want to group together objects that are far apart
- We sort all of the edges by weight and begin adding them back to our graph in order
- If an edge connects nodes that are already in the same cluster, we skip it
  - Thus, we don't make cycles
- We stop when we have ***k*** connected components

MST saves the day

- This algorithm is exactly Kruskal's algorithm
  - Add edges by increasing size, skipping ones that make a cycle
- We simply stop when we have ***k*** connected components instead of connecting everything
  - Alternatively, you can make the MST and delete the ***k*** – 1 most expensive edges

Huffman Codes

Prefix codes

- We want to make an encoding such that the encoding of one letter is not a prefix of the coding of another letter
  - Such an encoding is called a prefix code
- If you have a prefix code, you can scan bits from left to right and output a letter as soon as it matches
- Example prefix code:
  - $a \rightarrow 11$
  - $b \rightarrow 01$
  - $c \rightarrow 001$
  - $d \rightarrow 10$
  - $e \rightarrow 000$

Optimal prefix codes

- If each letter ***x*** has a frequency $\boldsymbol{f_x}$, with ***n*** letters total, $\boldsymbol{nf_x}$ gives the number of occurrences of ***x*** in a document
- Let ***code***(***x***) be the encoding of a letter ***x*** and ***S*** is the alphabet
- Total length of an encoding is:

$$\sum_{x \in S} nf_x |code(x)| = n \sum_{x \in S} f_x |code(x)|$$

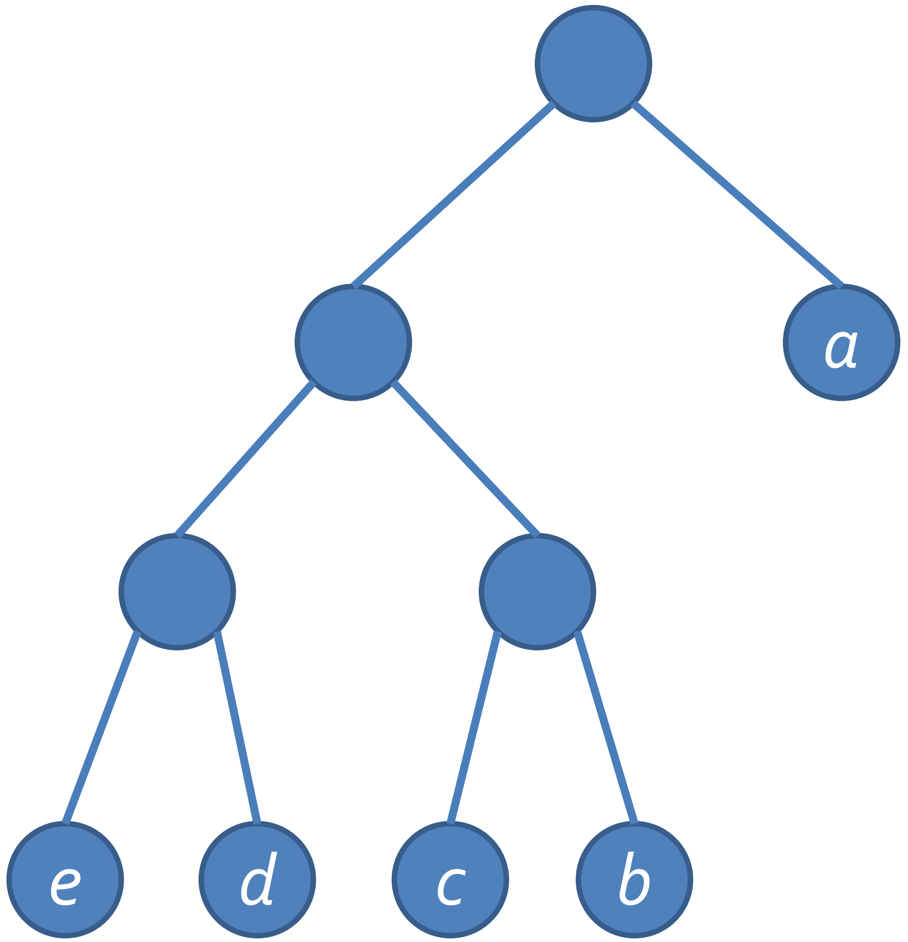
- An **optimal prefix code** minimizes average encoding length:

$$\sum_{x \in S} f_x |code(x)|$$

Algorithm design

- A key idea is that we can represent letters as leaves in a binary tree
  - Each left turn is a 0
  - Each right turn is a 1
- No letter will be the prefix of another
- Why?
- If a letter was the prefix of another, it would be on the path to the other letter, but every letter is a leaf

Prefix code tree example



$a \rightarrow 1$

$b \rightarrow 011$

$c \rightarrow 010$

$d \rightarrow 001$

$e \rightarrow 000$

Full binary trees

- Recall that a binary tree is a rooted tree in which each node has 0, 1, or 2 children
- A **full binary tree** is one in which every node that isn't a leaf has two children

How can we figure out the tree structure?

- We know that the binary tree will be full, but there are many full binary trees with ***n*** leaves
- Imagine that we had a full binary tree ***T**** that was an optimal prefix tree
- We know that the low frequency letters should appear at the deepest levels of the tree
- For letters ***y*** and ***z***, and corresponding nodes *node*(***y***) and *node*(***z***), if *depth*(*node*(***y***)) < *depth*(*node*(***z***)) then $f_y \geq f_z$.

We don't have the structure of T^*

- If we did, we could label it by putting the highest frequency letters in the highest levels of the tree and then going down, level by level
- Instead, we work backwards
- The lowest frequency letter must be at the deepest leaf in the tree, call it ***v***
- Since this is a full binary tree, ***v*** must have a sibling ***w***

Algorithm description

- Take the two lowest frequency letters ***y*** and ***z***.
- Since they are neighbors in a full tree, we can stick them together and treat them like a meta-letter ***yz*** with the sum of their frequencies.
- Recursively repeat until everything is merged together.

Algorithm

- If ***S*** has two letters then
  - Encode one with 0 and the other with 1
- Else
  - Let ***y*** and ***z*** be the two lowest-frequency letters
  - Form a new alphabet ***S'*** by deleting ***y*** and ***z*** and replacing them with a new letter ***w*** of frequency $f_y + f_z$
  - Recursively construct a prefix code for ***S'*** with tree ***T'***
  - Define a prefix code for ***S*** as follows:
    - Start with ***T'***
    - Take the leaf labeled ***w*** and add two children below it labeled ***y*** and ***z***

Huffman Codes

- Make a Huffman encoding for the following alphabet, given the frequencies of each letter:

a	0.10
b	0.18
c	0.23
d	0.16
e	0.05
f	0.02
g	0.26

Upcoming

Next time…

- Review second third of the course
  - Divide and conquer
  - Master theorem
  - Dynamic programming

Reminders

- Work on Assignment 7
  - **Due Friday by midnight**
- Review chapters 5 and 6
- Final exam:
  - Wednesday, April 24, 2024
  - 8:00 – 10:00 a.m.
- **Department celebration today!**
  - 11:30 a.m. – 1:30 p.m. on the Point patio
  - Free food!