UPE Tutoring:

CS 180 Midterm Review

Sign-in https://tinyurl.com/180mtF20

Slides link available upon sign-in

Topics

- **Introductory Problems**
  - Celebrity Problem
  - Two Egg Drop
  - Stable Matching
- **Asymptotic Time Complexity**
- **Greedy Algorithms**
  - Interval Scheduling
- **Graphs**
  - BFS/DFS
  - Topological Sort
  - Dijkstra's Algorithm

Introductory Problems

Celebrity Problem

- **Context:** There are N people. A famous person/celebrity doesn't know any of the other N-1 people, but all N-1 people know that person.
  - The only available method to find out how a person knows another:
    - Pick 2 people, A and B
    - Ask A if they know B: notated as A → B
    - The answer is Yes (A → B) or No (A !→ B)

- **Problem:** Find who is famous.

Celebrity Problem - Naive Solution

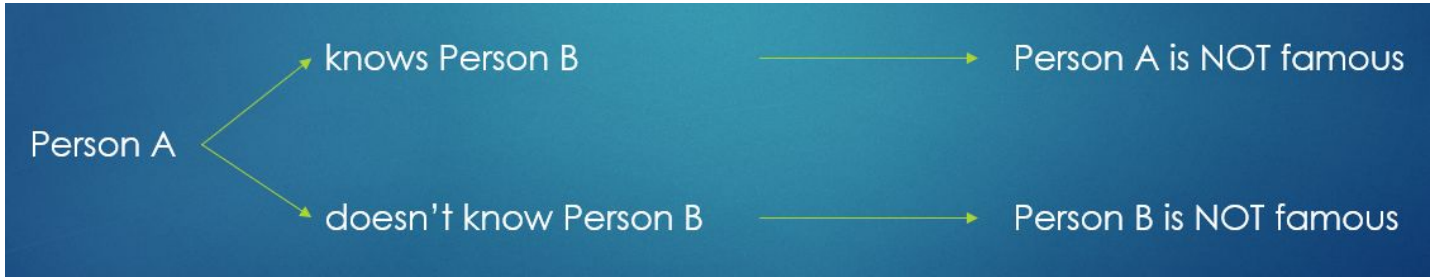
- **Algorithm:**
  - For each person (N people):
    - Check if this person doesn't know any others (N-1 asks)
    - Check if everyone else knows this person (N-1 asks)

- **Time Complexity**:
  - $2(N-1) * N = 2N^2 - 2N$ asks
  - $O(N^2)$

Can we find a more optimal algorithm?

Celebrity Problem - Key Observations

- Observation 1: *With each ask, we can eliminate either A or B.*



- Observation 2: *There is at most one famous person.*
  - If one person is famous then everyone else knows that person, but since famous people don't know anyone, all others are disqualified.

Celebrity Problem - Optimal Solution

- **Algorithm:**
  - While there are > 1 people left: (N-1 asks)
    - Pick two arbitrary people in the group
    - Ask A → B, which eliminates either A or B
  - With the last person:
    - Ask everyone else about them (N-1 asks)
    - Ask them about everyone else (N-1 asks)
- **Time Complexity:**
  - Elimination asks + Final Candidate Check → (N - 1) + 2(N - 1) = 3(N - 1)
  - O(N)
- Note that even without going in depth into a proof this already matches our intuitions; the algorithm will identify at most 1 famous person.

Two Egg Drop Problem - Background

- **Context:** We have two identical eggs and access to a 100-story building. Our goal is to find the highest floor you can drop an egg at without breaking it.
  - Properties of the eggs:
    - If an egg breaks when dropped from floor **n**, then it would have broken from any higher floor.
    - If an egg is dropped and doesn't break, then it can be dropped again. The max floor is not affected.
    - Once an egg breaks, it cannot be dropped again.

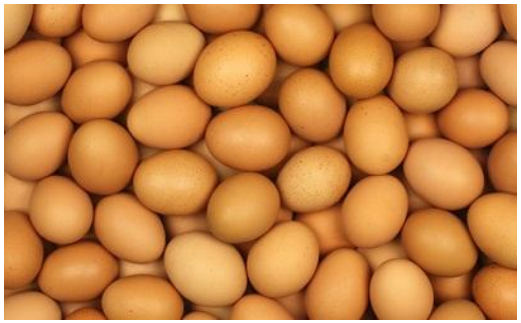
- **Problem:** What strategy should we use to find the highest floor such that we **minimize** the total number of drops? How many drops will we use?

Two Egg Drop Problem - Only One Egg?

- What if we only had 1 egg?
  - We must begin at the bottom floor and increment to higher floors by 1 until it breaks
    - Worst case: 100 drops, when the egg doesn't break until dropped from floor 100

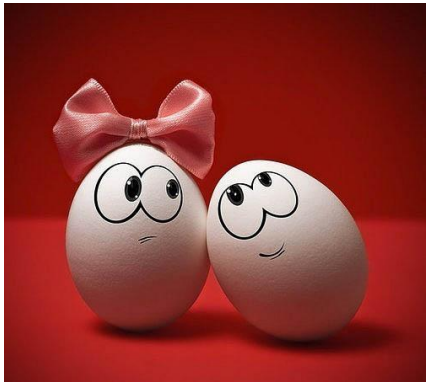
- Why can't we skip up by more than 1?
  - If we skip floors and the egg breaks, we have no way of knowing for sure which of the skipped floors was the highest one we could afford to drop the egg from.

Two Egg Drop Problem - All the Eggs??



- What if we could use as many eggs as we wanted?
  - The problem becomes a binary search problem.
- **Algorithm:**
  - Start at floor 50 and drop an egg. It either:
    - Breaks - We know that the solution is between floor 1 and 50
    - Lives - We know that the solution is between floor 51 - 100.
  - Whichever ends up being the case, we repeatedly split our search space in half, until we arrive at one floor, which must be our solution.
- **Time Complexity:** $\log_2(100) = 6.6 \rightarrow$ 7 drops for all cases

Two Egg Drop Problem - Back to Two Eggs



- By looking at the two extremes, we at least know that our answer for the two egg case must be between 7 and 100.
- We can't use a binary search, but we have more than 1 egg, so we can still do some ambitious floor skips...until our first egg breaks.
- What if we start at floor 10, and repeatedly skip 10 floors until our first egg breaks?
    - Worst case complexity: The floor at which eggs break is 99
        - First egg dropped 10 times
        - Second egg 9 times
        - Total: 19 times.

Two Egg Drop Problem - Minimization of Max Regret

- Finally, we want to minimize drops of the worst case scenario.
  - On the previous slide, our second egg's worst case is the same no matter how many times we drop the first egg.
  - Can we balance the # drops between the two eggs?
  - Yes; we can make the total # drops for all cases (highest floor eggs break at) a constant at n drops
  - How would you skip floors to achieve a constant n drops?

Two Egg Drop Problem - Solution

- **Algorithm:**
  - Drop once at floor n.
    - If it breaks, do n - 1 drops with the second egg. Total = n
  - Otherwise, we skip n - 1 floors. (1 drop already used at floor n.) Drop again.
    - If it breaks, do n - 2 drops with the second egg. Total = n
  - Repeat the above with skipping n - 2, n - 3, ...
  - Thus, we will always drop n times no matter how high the target floor is

- **Time Complexity:** The total floors we can test with n drops is the sum of all our skips
  - n + (n - 1) + (n - 2) + ... + 1 >= 100 ➔ n(n + 1) / 2 >= 100
  - Solving for the smallest n: 13.651 ➔ 14 drops

General Egg Drop Problem

- We can extend the Egg Drop Problem to k floors, t eggs, and a constant n drops
- Let's define $h_t(n)$ = total floors we can test with n drops and t eggs
  - We just found that when t = 2, $h_2(n) = n(n + 1) / 2$
  - When t = 1, we are forced to do linear search, so $h_1(n) = n$
  - When t = 0, it is obvious that $h_0(n) = 0$
- With t eggs:
  - Let's start by dropping one egg, exploring 1 floor:
    - If it breaks, we'll explore the lower floors; there are $h_{t-1}(n - 1)$ floors
    - If it doesn't break, we'll explore the upper floors: $h_t(n - 1)$ floors
  - Total floors explored: $h_t(n) = 1 + h_{t-1}(n - 1) + h_t(n - 1)$. A recurrence relation!
    - With base cases of t = 0, 1, 2
- Find n drops: Solve for n when $h_t(n) >= k$

Two Egg Drop Problem - Resources

- An even more in-depth look at the Two Egg Problem
- Walkthrough of the general egg drop problem
  - Note: The notation of the problem is slightly different
- How to use Dynamic Programming to solve the General Egg problem instead of using wasteful recursion
  - DP isn't covered yet for the midterm, but why not get a taste of what is to come?

Stable Matching - Background

- Let M = $\{m_1, ..., m_n\}$ be a set of n men, and W = $\{w_1, ..., w_n\}$ be a set of n women.
- Let M x W denote the set of all possible pairs of the form (m, w) where $m \in M$ and $w \in W$

- We define a **matching** S as a set of ordered pairs, each from within M x W, with the property that each member of M and each member of W appears in <u>at most one</u> pair in S.

- We define a **perfect matching** S' as a matching with the property that each member of M and each member of W appears in <u>exactly one</u> pair in S'.

- <u>English Please?</u>
  - The perfect matching S' is just a way of pairing off our men and women, such that everyone is married to someone, and no one is married to more than one person.

Stable Matching - Background

- Preference
  - Each man m ∈ M ranks all of the women in W. Man m **prefers** w to w' if m ranks w higher than w'. Each man's ranking is his **preference list**. Each woman also has a preference list.
- Stability
  - If there are two pairs (m, w) and (m', w') in a matching **P**, such that m prefers w' over w and w' prefers m over m', there's nothing stopping m and w' from abandoning their partners to form (m, w'). We call (m, w') an **instability with respect to P** because it doesn't belong to P despite the properties above.

- A matching P is **stable** if it is a 1) perfect matching and 2) contains no instability w.r.t. itself

Stable Matching - Background

- More English, Please?
- **Preference** is just a way of men and women comparing two members of the opposite sex and considering who they'd rather marry. A **preference list** is just a ranking of available candidates.

- An **instability** in a matching P just means that there is a man and woman in P that would rather be married to each other than who they are currently married to (sucks, right?)

- A matching P is **stable** if everybody is married to someone, there is no polygamy going on, and there isn't a man/woman pair that would rather be married to each other than with their current partners. Whew!

Stable Matching - Problem Statement

- Does there exist a stable matching for every set of preference lists?
- Given a set of preference lists, can we efficiently construct a stable matching if there is one?

Stable Matching - Short Answer

Yes, and yes.

Stable Matching - The Algorithm (Gale-Shapley)

```
Initially all m ∈ M and w ∈ W are free
While there is a man m who is free and hasn't proposed to
every woman
   Choose such a man m
   Let w be the highest-ranked woman in m's preference list
      to whom m has not yet proposed
   If w is free then
      (m, w) become engaged
   Else w is currently engaged to m'
      If w prefers m' to m then
         m remains free
      Else w prefers m to m'
         (m, w) become engaged
         m' becomes free
      Endif
   Endif
Endwhile
Return the set S of engaged pairs
```

Algorithm Design p. 6

Stable Matching - Proof

- After all that terminology setup, we get a relatively simple algorithm. How do we know that this produces a stable matching?

- Observation 1: *A woman w remains engaged as soon as she is proposed to. The sequence of her engagement partners gets better and better, in terms of her own preference list.*

- Observation 2: *The sequence of women that m proposes to gets worse and worse, in terms of his own preference list.*

Stable Matching - Proof

- Observation 3: *Complexity: The algorithm terminates after at most n^2 iterations of the while loop.*
  - Each iteration, a man will propose to a woman for the first and only time. If we let P(t) be the set of pairs (m, w) such that after iteration t m has proposed to w, we see that P(t + 1) > P(t). There exist n^2 matchings of men to women, so that must be the upper bound.
- Observation 4: *If m is free, there exists a woman to whom he has not proposed.*
  - **Proof by contradiction**: if m is free but has already proposed to every woman, then according to Observation 1 all n women have been engaged. Since it is a matching, there must be n engaged men, but there are n men total and m is not engaged.

Stable Matching - Proof

- Observation 5: *The set S returned at termination is a perfect matching*.
  - **Proof by contradiction**: Consider a case where the algorithm terminates with a free man m. At termination, m has proposed to every woman. However, according to Observation 4, there cannot be a free man that has proposed to every woman.

Stable Matching - Proof

- Observation 6: *For an execution of the Gale-Shapley algorithm that produces a set of pairs S, that set S is a stable matching.*
  - **Proof by Contradiction:** Assume that S contains an instability, that is there exists (m, w) and (m', w') in S such that m prefers w' and w' prefers m.
  - In the algorithm, m's last proposal was to w. If m didn't propose to w' before w, then w must be higher on m's preference list, a contradiction. If he did, then he was rejected by w' for some man m'' (which may or may not be m'). Either way, this is another contradiction.
  - We already know from Observation 5 that S is a perfect matching.
  - It follows that S must be a stable matching.

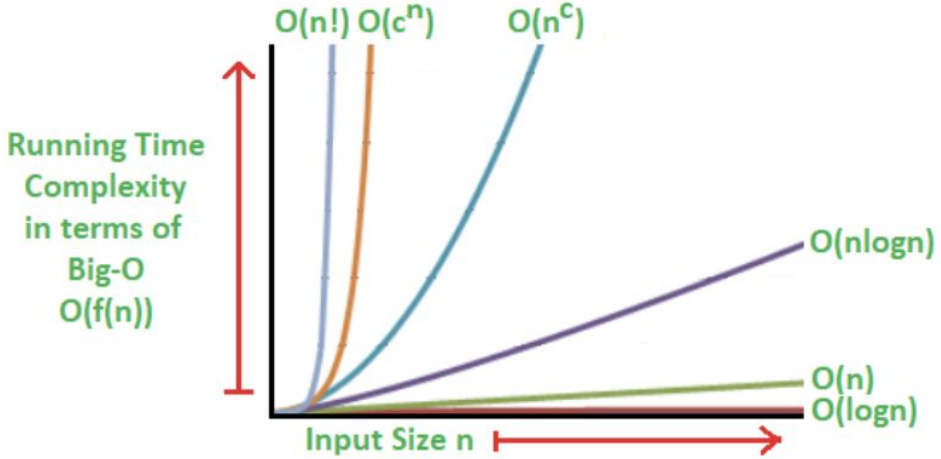
Time Complexity - Big O and Asymptotic Analysis

- Worst case time complexity analysis - in this class, usage is very similar to considering time complexity in CS 32.

- Mathematically:
  - Let T(n) be a function (e.g. worst case runtime of a certain algorithm on input size n)
  - Given another function f(n), we say that T(n) is O(f(n)) if, for sufficiently large n, the function T(n) is bounded above by a constant multiple of f(n)
  - T(n) = O(f(n)) if there exist constants c > 0 and n_0 >= 0 such that for all n >= n_0:
    - T(n) <= c * f(n)
  - Note that c cannot depend on n.
  - We say that T is asymptotically upper bounded by f.
  - Note that this means any O(n) function is also technically O(nlogn), and so on.

Time Complexity - Common Running Times

- O(1) - Constant
- O(n) - Linear
- O(nlogn) - Divide and Conquer
- O(logn) - Binary Search
- $O(n^2)$, $O(n^3)$, $O(n^k)$
- Non-polynomial



Greedy Algorithms and Interval Scheduling

Interval Scheduling - Problem

- **Context:** We have a set of requests S = {1, 2, ..., n} where the ith request corresponds to an interval of time starting at s(i) and ending at t(i).
  - A subset of requests is *compatible* if no two of them overlap in time.

- **Problem:** Given the definition of a compatible subset, design an algorithm to find as large a compatible subset of an input interval set S as possible (an optimal subset).

Interval Scheduling - Approach

- We try to find a simple rule to select a request i_1 to add to our result.
  - After we accept it, we reject all remaining requests that overlap, find the best i_2, etc. until we run out of requests.

- Picking a rule:
  - Always select a request that starts earliest ➔ lowest start time s(i)
  - Always select a request that finishes earliest ➔ lowest finish time f(i)
  - Always select a request that requires the smallest amount of time ➔ lowest f(i) - s(i)
  - Always select a request that has the least overlap with other requests

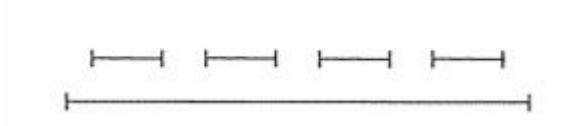
Interval Scheduling - Approach

- We try to find a simple rule to select a request i_1 to add to our result.
  - After we accept it, we reject all remaining requests that overlap, find the best i_2, etc. until we run out of requests.

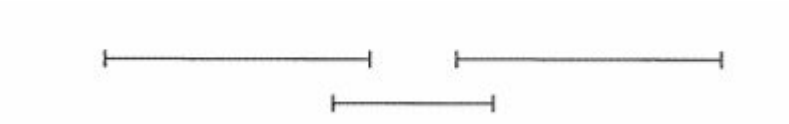
- Picking a rule:
  - Always select a request that starts earliest ➔ lowest start time s(i)
  - Always select a request that finishes earliest ➔ lowest finish time f(i)
  - Always select a request that requires the smallest amount of time ➔ lowest f(i) - s(i)
  - Always select a request that has the least overlap with other requests

Interval Scheduling - Why Others Fail

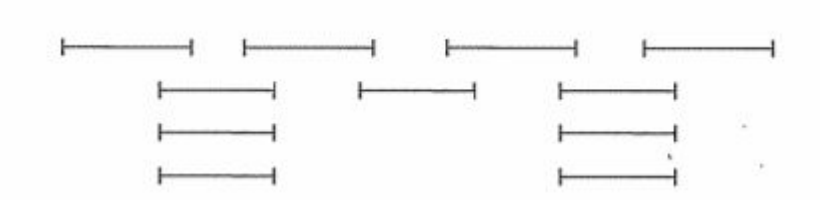
- Starts Earliest



- Smallest Time



- Least Overlap



Interval Scheduling - Formal Algorithm

Initially let R be the set of all requests, and let A be empty
While R is not yet empty
  Choose a request $i \in R$ that has the smallest finishing time
  Add request i to A
  Delete all requests from R that are not compatible with request i
EndWhile
Return the set A as the set of accepted requests

Algorithm Design p. 118

Interval Scheduling - Proof

- Let our greedy solution be A = $\{i_1, ..., i_k\}$ and some optimal solution O = $\{j_1, ..., j_m\}$. We want to show k = m (greedy answer makes an optimal answer).

- **Intuition:** We want the resource to become free again ASAP.
- **Induction:**
  - $f(i_1) <= f(j_1)$. For i_1 we just pick the earliest possible finish time, so this is true.
  - $f(i_r) <= f(j_r)$, for r <= k. We know that up to point r - 1, our greedy algorithm has "stayed ahead" (A's finish time earlier or same as O's finish time). Worst case scenario, they end at the same time, and the greedy algorithm picks j_r, which allows the induction to hold!
- **Time Complexity:**
  - O(NlogN) - refer to Algorithm Design p. 121 for runtime analysis

Greedy Algorithms Proof Approaches

- **“The greedy algorithm stays ahead”:**
  a. Write a greedy algorithm
  b. Show that its step by step process does better than any other variation
  c. It follows that the algorithm produces an optimal solution
- **Exchange Argument:**
  a. Write a greedy algorithm
  b. Consider any possible solution to the problem
  c. Transform it into the solution found by the greedy algorithm without hurting its quality
  d. It follows that the algorithm produces an optimal solution

Interval Scheduling - Why Be Greedy?

- We showed that the solution to simple interval scheduling is to greedily choose the intervals with the smallest finishing time.
- But do you know why?

Interval Scheduling - Optimal Substructure

- An **optimal substructure** is a fancy of way of saying “the optimal answer of this larger answer can be constructed from optimal answer of smaller subsets of this answer”.
- In this case, denote S_i as the set of activities after the interval i ends.
- We can ask the same question : “What’s the max cardinality of S_i’?”. The answer of this S_i’ can be used to construct our solution for S! *This is an optimal substructure*.
- Now, consider the set of intervals that overlap at the beginning, such that no interval can be added before it. If we select the interval of earliest finish time (called f_i), rather than another interval(called f_j), then we can get S_i to be from [f_i, end), and S_j to be from [f_j, end).

Interval Scheduling - Monotonicity

- We also need **monotonicity** in our argument to ensure that choosing S_i will be better than S_j.

- **Monotonicity** of a function means if x < y, then f(x) <= f(y) *always*(monotonically increasing), or f(x) >= f(y) *always*(monotonically decreasing). Generalizing this to sets as inputs, mapped to real number outputs, we have $x \subseteq y$, then f(x) <= f(y) for monotonically increasing.

- Our function max_nonoverlap() is monotonic. Why? Because the larger set S_i can always include the same intervals as S_j's answer!

When to Be Greedy

Now that you understand Interval Scheduling, what about other greedy problems? There's a boilerplate template that you can follow!

- Determine whether optimal substructure exists *(We can solve for a smaller subset of S).*
- Develop a recursive solution *(Take one of the front intervals, solve for the S_j).*
- Show that if we make a greedy choice, only 1 subproblem remains*(S_i is our subproblem)*, and that it's always safe to make the greedy choice *(monotonicity is your friend).*
- Use the greedy solution, and make it iterative for brownie points.

Graphs

Graphs - Basics

- Graphs are covered in the prerequisite class Math 61, so it assumed that students have some familiarity with them.

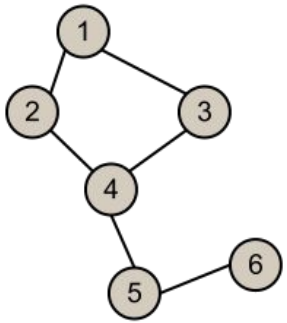
- A **graph (G)** is simply a collection of **V** nodes and **E** edges, each of which joins two vertices.
- An edge $e \in E$ looks like this: $e = \{u, v\}$ for some $u, v \in V$.

- A **directed graph** is similar to a graph, except it allows for representation of asymmetric relationships. In those situations, an edge e' = {u, v} specifically means that edge e' leaves node u and enters v.
- A **tree** is an undirected graph that is connected and has no cycles.

Graphs - Adjacency Matrix Representation

- An **adjacency matrix** is an nxn matrix where n represents the number of nodes in the graph. A 1 in coordinate {u, v} means there exists an edge leading from u to v.

Undirected Graph & Adjacency Matrix



Undirected Graph

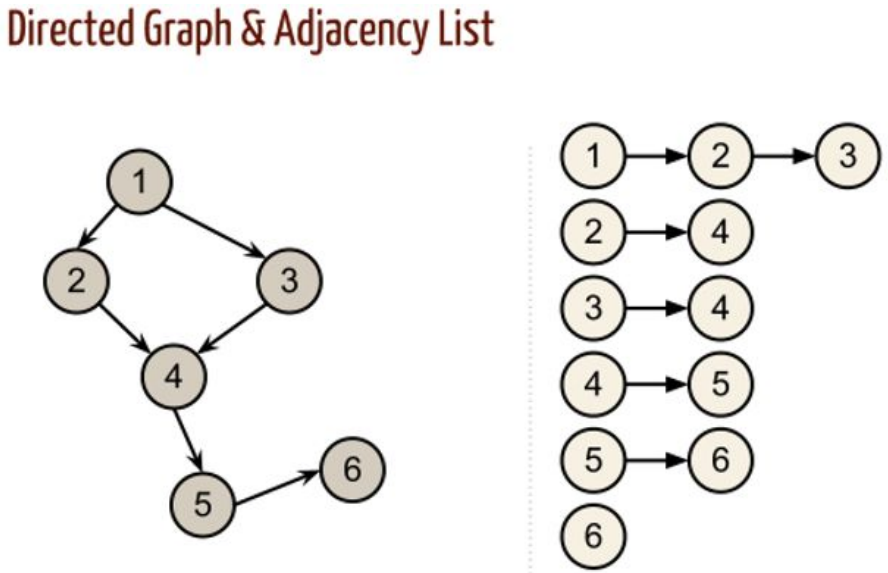
	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	0	1	0	0
3	1	0	0	1	0	0
4	0	1	1	0	1	0
5	0	0	0	1	0	1
6	0	0	0	0	1	0

Adjacency Matrix

Credit to Stoimen for graphics.

Graphs - Adjacency List Representation

- An **adjacency list** is a set of linked lists such that the first node represents a node and successive nodes are nodes the origin vertex leads to via edges.



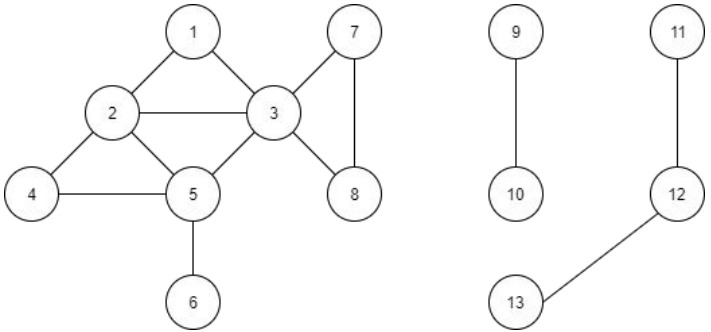
Breadth First Search

- We can use BFS to determine s-t connectivity.
- We explore outward from node s, adding future vertices one “layer” at a time.
- Think of it as “flooding” outwards from s, until we reach a layer that has no nodes.
- Define layers as follows:
  - Layer L_1 consists of nodes that are neighbors of s. (s is in Layer L_0.)
  - Assuming we have defined Layers L_1 to L_j, Layer L_{j+1} consists of nodes that do not belong to an earlier layer that have an edge to a node in Layer L_j.

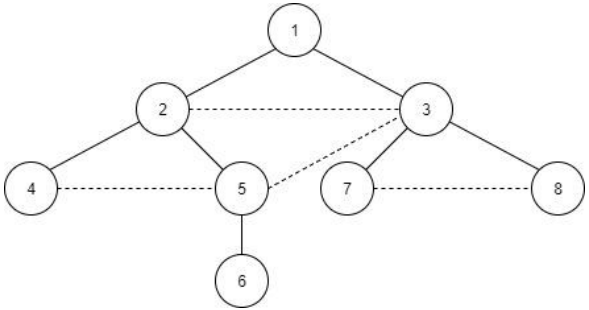
- A BFS using this method produces a tree T rooted at s with all nodes reachable from s. Each layer L_j consists of all nodes at distance j from s. A path from s to t exists iff t appears in one of these layers.

Breadth First Search - Breadth First Search Trees

- A **breadth first search tree** is a tree constructed by a breadth first search that adds edges with a previously unseen node to the tree.



Graph G, consisting of 13 vertices.



BFS Tree T derived from Graph G after 3 layers added, starting with node 1 as the root. Solid lines are edges in T; dotted lines are in G but not in T

Breadth First Search - BFS Tree Proof

- Notice that in the BFS tree, the nontree edges (edges not in the tree) are connecting either same layer nodes or nodes one layer apart. This is a property of all BFS trees.

- Observation 1: *Let T be a BFS tree, and let x and y be nodes in T belonging to Layers L_i and L_j respectively. Also let (x, y) be an edge of G. Then i and j differ by at most 1.*
  - **Proof by contradiction:** Assume that i < j - 1. Consider the point where edges incident to x are being examined. Since x is in Layer L_i, the only nodes discovered from x are in L_{i+1} and earlier. However, y is a neighbor but is not any of these nodes; a contradiction!
- BFS also finds the shortest paths from s to all other nodes in an unweighted graph
  - Proved in lecture

Connected Components

- The set of nodes discovered by the BFS is the **connected component (R)** of G containing s.
  - The set of nodes reachable from the starting node s

```
R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ R
  Add v to R
Endwhile
```

  - The proof is on Algorithm Design page 82-83.
- For a node t in R, we can recover the path from s to t by recording the edge (u, v) we used to add node v to R. Then, we trace the edges backwards from t to eventually reach s.

Depth First Search

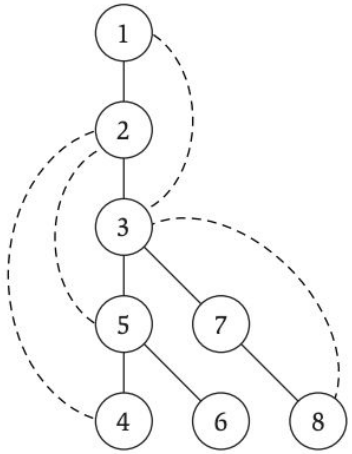
- Another natural way to solve the s-t connectivity problem
- Think about exploring a maze; you want to travel until you hit a dead end before you consider another path.

```
DFS(u):
  Mark u as "Explored" and add u to R
  For each edge (u, v) incident to u
    If v is not marked "Explored" then
      Recursively invoke DFS(v)
    Endif
  Endfor
```

Algorithm Design p. 84

- To solve the s-t connectivity problem, mark all nodes as not explored and call DFS(s).

Depth First Search - DFS Tree



DFS tree constructed from graph G, beginning with 1 as s. Dotted lines are edges in G not in T.

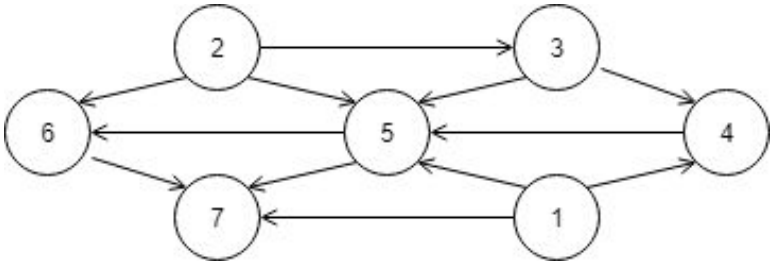
- A **depth first search tree** is a tree constructed by a depth first search that adds edges with a previously unseen node to the tree.
- In the algorithm, every time we call DFS(v) during the DFS(u) call, we add the edge (u, v) to the tree.

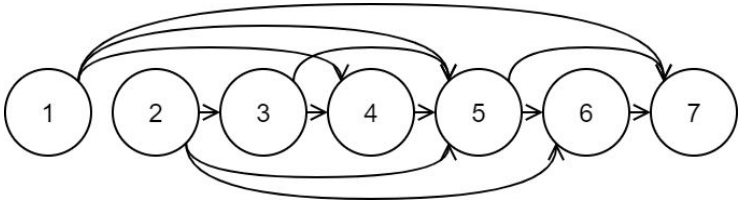
- Observation 1: *For a recursive call DFS(r), nodes marked as "Explored" between the invocation and end of the call are descendants of r in T.*
- Observation 2: *Let T be a DFS tree, and x and y be nodes in T, and (x, y) be an edge in G not in T. Then x or y is an ancestor of the other.*

Topological Sort - Directed Acyclic Graphs

- A **directed acyclic graph (left)** is a directed graph that contains no cycles.

- A **topological ordering (right)** of a graph G is an ordering of its nodes as $v_1, v_2, \ldots, v_n$ such that for every edge (v_i, v_j), we have $i < j$. All edges point "forward" in the ordering. If each node represents a task that must be completed and each edge a dependency, then a topological ordering is an order in which all tasks can be completed safely.





Topological Sort - Proof that Topo Order is DAG

- Observation 1: *If a graph G has a topological ordering, then G is a DAG.*
  - **Proof by contradiction:** Assume that G has a topological ordering $v_1, \ldots, v_n$ which also has a cycle C. Let v_i be the lowest indexed node on C, and let v_j be the node on C just before v_i, such that (v_j, v_i) is an edge.
  - By our choice of i, we know $j > i$, which contradicts our assumption that $v_1, \ldots, v_n$ is a topological ordering, since $j < i$ would need to be true for (v_j, v_i) to exist as an edge.

Topological Sort - Problem Statement

Does every DAG have a topological ordering? If so, how do we find one efficiently?

Topological Sort - Getting Started

- The first node in a topo ordering needs to have no incoming edges.

- Observation 2: *In every DAG G, there is a node v with no incoming edges.*
  - **Proof by contradiction:** Let G be a DAG where every node has at least one incoming edge. Pick any node v, and begin following edges backward from v: since v has at least one incoming edge we can always follow an edge backwards to some node u, and so on.
  - We can do this indefinitely, since every node has an incoming edge. After doing this n + 1 times, we have visited some node w twice because there are only n nodes. We can then let C denote the nodes visited between visits of w, which is a cycle, a contradiction.

Topological Sort - Proof of Converse

- Observation 3: *If a graph G is a DAG, then it has a topological ordering.*
  - **Proof by induction:** Claim by induction that every DAG has a topological ordering. This is true for base cases of DAGS with one or two nodes.

  - Inductive step: Now consider that it is true for DAGs with n nodes. Given a DAG G with n + 1 nodes, we can use Observation 2 to find a node v with no incoming edges, and place it first in our topological ordering since all of its edges point forward. G - {v} is a DAG, since deleting v can't create any cycles. G - {v} has n nodes, so we can apply induction to get its topological ordering. We append that to v. This is an ordering of G where all nodes point forward, so it is indeed a topological ordering. Whew!

Topological Sort - Algorithm

- Below is the algorithm to compute a topological ordering of graph G.

```
To compute a topological ordering of G:
Find a node v with no incoming edges and order it first
Delete v from G
Recursively compute a topological ordering of G−{v}
  and append this order after v
```

Algorithm Design p. 102

- It takes O(n) time to find a node v with no incoming edges and deleting it.
- The algorithm runs for n iterations, to the time complexity is $O(n^2)$, where n is the number of nodes.

- Can we do better?

Topological Sort - Optimal Algorithm

- We can do better by tracking two things:
  - For each node w, the number of incoming edges w has from active nodes
  - The set S of all active nodes in G with no incoming edges from other active nodes
- **Algorithm:**
  - Initialize the above with one pass through nodes and edges.
  - Each iteration, we delete a node v from S. Then we go through all nodes w to which v had an edge and decrement 1 from their number of active incoming edges. If, for a node w, it drops to 0, we add w to S.
    - This way, we track nodes eligible for deletion at all times, with constant work per edge.
- **Time Complexity:** O(m + n), where m is # of edges and n is # of nodes.

Shortest Paths in a (Positively Weighted) Graph

- **Context:** Given a graph G = (V, E)
  - Given a function that defines the length for an edge: w(e) where e $\in$ E, w(e) >= 0
  - Path definition: P = {e_1, e_2, ..., e_n}, where e_1 = (s, v_1), e2 = (v_1, v_2)... and e_n = (v_n,t). P$\subseteq$E.
  - Length of the path definition: sum of w(e_1) + w(e_2) ... + w(e_n)

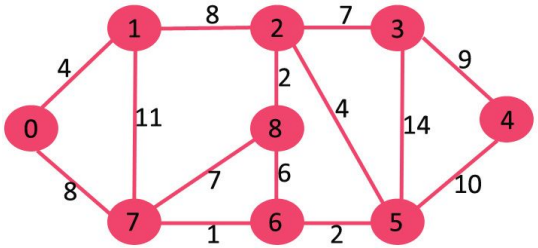
- **Problem:** For two nodes s, t $\in$ V, find the shortest path (and its length) between s and t.
  - Alternatively: Given a start node s, find the shortest paths from s to any other node

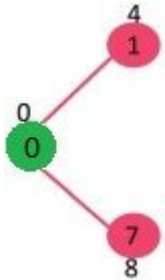
Dijkstra's Algorithm

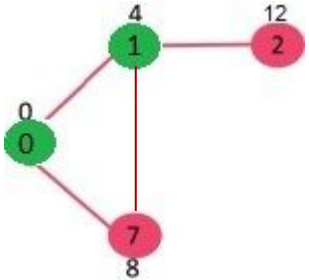
- Initialization:
  - Let S be a set of explored nodes. Initially S = {s}
    - Also, store the edge (x, y) used to add a node y to S
  - Let d(u) be a store of the best distance to node u, for each node u∈S. Initially d(s)=0
    - Initially, d(z) for any other node z is ∞
- While all nodes have not been explored (while S != V):
  - Consider all neighbor nodes (to any node u in S) that have not been explored yet
    - Find the neighbor node v such that it achieves the min(d(u) + w(e)) out of all the possible neighbor nodes. e is the edge (u, v)
    - Add v to S and store the edge (u, v)
    - Set d(v) = d(u) + w(e)
- Reconstruct the shortest paths by walking backwards on the edges stored

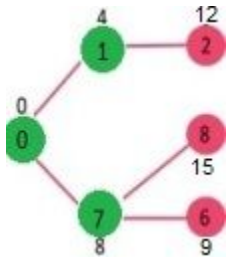
Dijkstra's Algorithm - Example

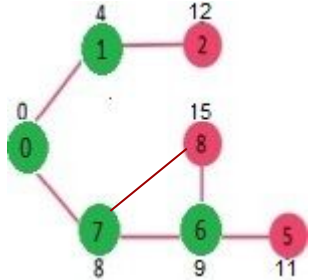
We start with the graph:



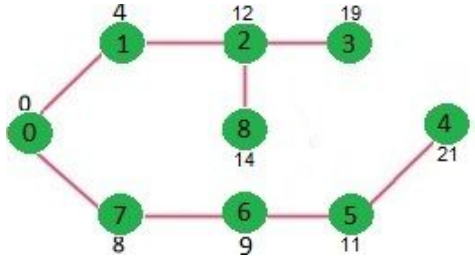








...



Dijkstra's Algorithm - Optimal Substructure & Monotonicity

- What is the optimal substructure?
  - The shortest path from s to any node v is in the shortest path from s to t!
  - If P* = s ➔ v ➔ t in the optimal answer, then s ➔ v must also be the shortest path from s to v.

- Where is the monotonicity?
  - The monotonicity lies in our assumption of positive length edges. We will never find a path shorter than the optimal by traversing farther in any suboptimal paths, because they will just get longer and longer.

Dijkstra's Algorithm - Formal Proof

- We want to prove this inductively:
- **Base case:** Trivial - a single node to itself is distance 0.
- **Inductive case:** Suppose for contradiction we have our first vertex u, such that the distance d_u in S is not the shortest distance. This means that:
  1. u is not s. s ➔ s is 0. Thus, we also know from this that S ≠ ∅.
  2. On the path P = s ➔ v ➔ u, we have that s ➔ v was already in the shortest paths set. At the beginning, v is equal to s, since S is not empty. We contradict, and state that s ➔ v ➔ u is longer than another path, namely s ➔ x ➔ y ➔ u. It can't be s ➔ x ➔ u because s ➔ x >= s ➔ v. Additionally, s ➔ x ➔ y >= s ➔ v ➔ u because we have chosen s ➔ v ➔ u as the shortest path of all paths considered. Monotonically, s ➔ x ➔ y ➔ u >= s ➔ x ➔ y >= s ➔ v ➔ u.

Midterm Tips

- Types of questions and advice for each:
  - (1 question) Explain an algorithm e.g. DFS, Topological sort
    - Remember the algorithm itself, correctness proofs, and time complexity proofs for all algorithms learned in class + in book chapters + HW
      - Not memorizing, more like internalizing
  - (1 question) Run an algorithm on data
    - Hope that you internalized algorithms enough
  - New, creative problems
    - How is it similar to problems you've seen e.g. complexity
    - Think of Best Conceivable Complexity/Runtime (BCR), ways to achieve it
    - Practice solving new problems
    - Pray for luck

Midterm Questions from Fall 2019

1. Describe the topological sort algorithm in a DAG. Prove its correctness. Analyze its complexity.
2. Run Merge sort on the following set of numbers. Show every step. Analyze the time complexity of merge sort on a set of n numbers (show every step).
    a. Y'all haven't learned Merge Sort yet, so replace it with any other algo
3. You have been commissioned to write a program for the next version of electronic voting software for UCLA. The input will be the number of candidates, d, and an array votes of size v holding the votes in the order they were cast where each vote is an integer from 1 to d. The goal is to determine if there is a candidate with a majority of the votes (more than half the votes). You can use only a constant number of extra storage (note that v and d are not constants). Prove the correctness of your algorithm and analyze its time complexity.

Midterm Questions cont.

4. Consider a sorted list of n integers and given integer L. We want to find two numbers in the list whose sum is equal to L. Design an efficient algorithm for solving this problem (note: an $O(n^2)$ algorithm would be trivial considering all possible pairs). Justify your answer and analyze its time complexity.

5. Suppose that you are given an algorithm as a blackbox. You cannot see how it is designed. The blackbox has the following properties: If you input any sequence of real numbers, and an integer k, the algorithm will answer YES or NO indicating whether there is a subset of the numbers whose sum is exactly k. Show how to use this blackbox to find the subset whose sum is k, if it exists. You should use the blackbox O(n) times (where n is the size of the input sequence).

Final Question that is Similar to Midterm Question 5

Consider an array a1,...,an of n integers, that is hidden from us. We have access to this array through a procedure knapsack(.,.). For a set $S \subseteq \{1,..,n\}$ and an integer k, knapsack(S,k) will output "yes" if there is a subset $T \subseteq S$ such that the numbers indexed in T add up to k, and it will output "no" otherwise.

a. Design an algorithm that calls knapsack only O(n) times and outputs a set $S \subseteq \{1,...,n\}$ such that the numbers indexed in S add up to k, if such a set exists. You can use ONLY the knapsack function (e.g. you cannot sort the numbers or do any other operations on them).

b. For example, suppose a1 = 2, a2 = 4, a3 = 3, a4 = 1, and k = 7. Then, knapsack({1,2,3,4}, 7) returns "yes" and knapsack({1,3,4}, 7) returns "no". In this case your algorithm can output either the sets {1,2,4} or {2,3}. Note that for example {1,2,4} are indices of the numbers, that is a1, a2, and a4.

Good luck!

Sign-in https://tinyurl.com/180mtF20
Slides https://tinyurl.com/180mtF20slides

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
  - Location: ACM/UPE Clubhouse (Boelter 2763)
  - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.