10/5: The Famous Problem

- Precision required in the design and understanding of the algorithm
- Contrary examples should use the lowest number possible
- Algorithm comes from Al Khwarizmi
    - Sequence of steps that works for an arbitrary value n
- The Famous Problem
    - Model of computation states that we can only ask a person if they know a single other person in one question
    - Pick any 2, ask one if they know the second person → assume perfect truths
    - Ask all other n - 1 people if they know that person
        - All yes answers → possible that the person is famous
    - Ask that person if they know everyone else
        - All no answers → person is famous
    - We have to ask ~2(n - 1) questions per person → algorithms require resources
        - Time, power, etc.
        - Resource here is the number of questions
        - This algorithm takes 2n(n - 1) questions → works for any arbitrary n
            - $2n^2 - 2n$ → for large n's, the n^2 term dominates → quadratic algorithm
    - Arbitrary vs. random
        - Random is harder → try to avoid
        - Arbitrary is easier → still has some selection criteria, but criteria doesn't matter for the success of the algorithm
    - 1 knows 2 → 2 is possibly famous, but 1 is definitely not famous
        - Problem size reduced
    - 1 doesn't know 2 → 1 is possibly famous, 2 is not famous
        - Problem size reduced
    - 1 question reduces problem size from n to n - 1 → elimination of people that aren't famous → problem reduction
        - Ask questions until left with 1 person → there are n - 1 questions asked
            - Last person is the only possible famous person → not guaranteed
            - Must go back and make sure if they know anyone and if everyone knows them
            - 3(n - 1) questions asked: ~n questions
- For now, we care about worst-case analysis rather than average-case
- Lower-bound analysis - helps us discover what the minimum complexity is

10/7: Perfect Matching

- We need a general model of computation
    - Must be universal and simple
    - Serial model of computation: Von-Neumann model
        - Contains an ALU that does simple computation
        - Contains a limited, small number of load-called registers
        - Has I/O

        - Has a large memory
            - Each unit takes about 1 unit of time
- Memory caching is simply a minor detail when it comes to algorithm design
    - High level algorithm design is most important, registers/caches/etc. are an afterthought
- Ex) Find the summation of n numbers, S
    - Read in the first number through the ALU, then insert into RAM
    - Repeat above with second number
    - Repeat up until number a_n
    - Reading all numbers to ALU takes n units of time
    - Reading all numbers from ALU to RAM takes another n units of time
        - All together, the operation takes $2n$ units of time
        - Unit of time varies with technology → use unit of time as a standard unit
    - Reading back into ALU takes another n units of time
    - Summing all the values takes n - 1 (about n) units of time
        - Intermediate values can be stored in the limited storage of the ALU → no time units necessary to move them
    - Output the resultant value
        - Total: about $O(4n)$
- Ex) We have a ladder with steps 1 → n and a person with an egg. What's the lowest level that the egg can be dropped from for it to break?
    - Only number of drops is counted in our model of computation
    - Jumping to a step as a guess won't work → if the egg breaks, the problem is unsolvable
        - As a result, the first step must be dropping it from step 1
        - Same logic tells us we cannot jump from step 2 to step 3
            - This logic forces an algorithm to take shape → we must go one step at a time until the egg breaks
    - Now, we buy another egg, can we do better than the previous algorithm?
        - Partition ladder into steps of $\sqrt{n}$ size
            - Since steps are integers, we technically floor/ceiling $\sqrt{n}$
            - This size is a balance of number of partitions vs. size of partitions
        - Drop one egg at each partition boundary to isolate a search range
            - This egg is dropped at most $\sqrt{n}$ times
            - The other egg will also be dropped at most $\sqrt{n}$ times
                - Worst case: $2\sqrt{n}$ drops to solve the problem
        - Partitions don't have to be equal size → we can get better with unequal sizes
- Ex) We have a list of men and a list of women, both of size n
    - Each person has an ordered list of the opposite gender that they want to marry
        - This list must contain all n members of the opposite gender exactly once'
    - We want to match each person to another, the output is a set of n edges → each person can marry exactly one other person

- An unstable match is a match between 2 pairs where both participants have a preferred option available
    - There must be 0 unstable matches in the final result
- Start by looking at M_i, who has a preference list of W
    - M_i proposes to each W in order of preference
        - Each M will ask at most *n* people to marry them
        - If he finds one, they are a temporary match
            - M_i is happy, his next choice is less preferred, W_j is not necessarily happy, she may find a more preferred M
            - W_j searches through preferences, if she find a more preferred match, M_i's temporary match is broken
                - M_i has to continue going through preferences → matches get worse over time
    - If there exists a man without a match, he will propose to the highest rank remaining in his list
        - The order that these men are iterated through doesn't affect the success of the algorithm → it's arbitrary

10/9: Discussion 1

- Von Neumann architecture
    - CPU and memory share a bus → connection between CPU and memory costs us a unit of time
    - Input and output are separate → connection to CPU also costs another unit of time
    - Universalizes computational architecture
    - Pushing n numbers into an array will cost n units of time
- Celebrity Problem
    - In a party of N people, only up to one person is known to everyone. Such a person may or may not be present in the party. If present, they don't know anyone in the party. The only questions we can ask are does "A know B?". Find the celebrity in the minimum number of questions
        - Good solution: rule out all but 1 people by asking n - 1 questions, rules out 1 person per question
            - Finish by asking the remaining person if they know anyone else and if everyone else knows him (another 2(n-1) questions)
            - O(N) questions
- Stable Matching
    - Set of 2 men, {m, m'}, and a set of 2 women, {w, w'} → find stable matches
    - Gale-Shapley algorithm
        - All M and W are free
        - While there is a man m who is free and hasn't proposed to every woman
            - Choose 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 w'
                    - If w prefers m' to m then
                        - m remains free
                    - else
                        - w prefers m to m', (m, w) become engaged while m' becomes free
            - Return the set S of engaged pairs
            - Algorithm is $O(N^2)$
        - Men-Optimal Proof
            - Claim: Gale-Shapley with the man proposing results in a men-optimal result
            - Proof by Contradiction:
                - Men propose in order → at least one man was rejected by a valid partner
                - Let m and w be the first such reject in S
                - This happens because w chose some m' > m
                - Let S' be a stable matching with m, w paired
                - Let w' be partner of m' in S'
                - m' was not rejected by valid woman in S before m was rejected by w
                    - m' prefers w to w'
                - Know w prefers m' over m, her partner in S'
                    - m' and w form a blocking pair in S' ><
            - Be careful not to change the basic truths of the world
- Induction
    - Start with P(n)
        - Base case: Show that the statement holds for n = 0, n = 1
        - Assumption: Assume that P(k) holds
        - Inductive Step: Show that id P(k) holds, then also P(k + 1) holds
- Interview Questions
    - Palindrome Permutation
        - Given a string, write a function to check if it is a permutation of a palindrome. A palindrome is a word or phrase that is the same forwards and backwards. A permutation is a rearrangement of letters. The palindrome doesn't need to be limited to just dictionary words.
            - Read letters of string into hashmap, skipping spaces
            - Iterate through hashmap, counting how many letters have an odd number of occurrences
            - If the number of odd letters is > 1, return false, else return true
    - Smallest Difference

        - Given 2 arrays of integers, compute the pair of values (one value in each array) with the smallest difference. Return this difference
            - Sort both arrays
            - Iterate through both arrays with 2 separate pointers, comparing the values in each array
            - Advance the pointer with the lower value
            - Track the minimum difference found

10/11: Order Notation

- GS Stable Matching Proof
    - Current pairs are (m, w) and (m', w') → unstable matching exists
    - Did m propose to w'?
        - If not, then we know by the algorithm that since m has proposed to w, then w must be more preferred than w' by m ><
        - If yes, then since m is not paired with w' in the final result, then w' must've later been asked by some m" that is > m
            - Since w' ended up with m', that means that m' must be > m"
            - By transitivity, this means m' > m ><
- Order Notation
    - T(n) = O(f(n))
        - There exists some c > 0 and n_0 >= 0 such that for n >= n_0, T_n <= cf(n)
            - c is a constant, and in practice, a small number
        - We only care about large cases n > n_0
        - $n^2 = O(n^3)$
        - $n^2 + 40 = O(n^3)$
        - n^3 != $O(n^2)$
        - $n^2 = O(n^2 - 5n - 1000)$
        - $n^2 = O(n^2)$
    - If algorithm runs in $5n + 20 + n^{1/2}$, then it's O(n) as a simplification
    - If algorithm runs in $4n^2 + 20 - 10n^{1/2}$, then it cannot be O(n)
    - log n, …, $n^{1/2}$, …, n, $n^{3/2}$, n^2, n^3, …, 2^n, …, n!, …
    - At the algorithm design level, it's a waste of time to worry about improving constants, just worry about the order
    - T(n) = Ω(f(n))
        - There exists some c > 0 and n_0 >= 0 such that for n >= n_0, T_n >= cf(n)
            - $n^3 = Ω(n^2)$, $n^2 = O(n^3)$
- Induction
    - Base case + inductive step

10/13: Greedy Paradigm

- Scheduling Problem
    - Given a set of *n* tasks with a start time S_i and end time E_i, how do we come up with a subset of tasks that don't overlap?
        - With no constraints: you can pick a single task and be done

        - Now, we need to maximize the size of the subset
    - Greedy Paradigm - we pick something to be in the solution without any analysis and then we stick to that solution
        - First Try:
            - Choice between a long interval and a short interval → pick the shorter interval → statistically eliminates less intervals for consideration
            - Delete all overlaps with selected interval
            - Repeat
                - Fails some cases
        - Second Try:
            - Move from left to right and push first interval into solution
            - Delete all overlaps with selected interval
            - Repeat
                - Fails some cases
        - Third Try:
            - Move from left to right, picking the next interval that ends first
            - Delete all overlaps with selected interval
            - Repeat
                - Optimal solution
                - Proof:
                    - Create an imaginary optimal solution
                    - Imagine the first *i* intervals of both solutions are the same
                    - After interval *i*, the solutions must have picked different intervals *a* and *b*
                        - Based on the methodology of the algorithm, $t_a < t_b$, where *t* is the endpoint of an interval
                        - Therefore *a* can be substituted into *b*
                        - This can be continued for the rest of the intervals in the imaginary solution
                        - The only way for the imaginary solution to be longer is for there to be intervals remaining after substitution
                            - These intervals would've been picked up by our algorithm ><

- Graphs
    - A graph consists of a set of vertices/nodes and a set of edges/links
        - Edges may or may not have direction
        - $G = (V, E)$, $V = \{ v_1, v_2, \dots , v_n \}$, $E = \{ (v_1, v_2), (v_2, v_3), \dots , (v_m, v_n) \}$
    - Breadth-First Search (BFS)
        - Need a starting point
        - Find neighbors of starting point and search each

        - Once there are no more neighbors, we go to a neighbor of the second level of searches
        - Repeat until all nodes have been checked
        - Can look at as a FIFO queue

10/16: Discussion 2

- Interval scheduling runtime → O(n log n) time due to sorting of intervals
- Binary Search Tree
    - A rooted binary tree where each node has a value, a left subtree and a right subtree. Any node in a nodes left subtree must contain a value less than the current node's value, and any in the right subtree must contain a value greater than the current node's value

10/19: Graph Searches

- When we talk about a path, we mean a simple path → no repeated vertices
- Cycle - a path where the first and last vertex are the same
- Search Algorithms
    - BFS - Visit all neighbors of a vertex one by one, we do not visit any other vertices until all the neighbors of the current vertex are visited
        - Order of visiting these neighbors is arbitrary
        - Highlight the edges that took you to a given vertex
            - Eliminates cycle → turns graph into a tree (BFS tree)
        - Each vertex is assigned a label dependent on its level
            - Within a level, there are no edges → would create a cycle
            - Number of edges between 2 vertices is the distance between those vertices
                - Distance is the length of the shortest path
            - The level number containing the vertex in the BFS tree is the distance from the root to the vertex
                - BFS trees are useful tools for finding distances
                - Proof:
                    - Take a BFS tree rooted at s with vertices x_1, x_2, … , x_i at levels 1, 2, … , i, respectively
                    - By definition x_i is at level i
                    - Assume $j = \text{dist}(s, x_i) < i$
                    - This means there is a path through vertices y_1, y_2, … , x_i which is of length j
                        - No paths of vertices other than x_i
                    - If such a path exists, y_1 is a neighbor to s,
                    - If you can get to x_i in j steps, then x_i would be at level j or less in the BFS tree
                    - This contradicts with the assumption that j is less than i
        - Time complexity: $O(e + n)$

- DFS - Each vertex will search as deep as possible before checking neighbors
    - Constructs a DFS tree
        - Levels have no significance in the DFS tree
        - May contain back-edges → original graph contains cycles
- A graph is undirected and connected with *e* edges and *n* vertices
    - Minimum number of edges? $e = n - 1$
    - Same applies for graphs → $e >= n - 1$
    - Maximum number of edges? $n(n - 1) / 2$
    - How to represent a graph?
        - Adjacency matrix → *n* cols and *n* rows → n^2 size, constant time access
            - Wherever there's an edge between *i* and *j*, there is a 1 at entry (*i*, *j*)
            - Main diagonal depends on the problem
            - Undirected graphs have symmetric adjacency matrices across the main diagonal
        - Linked list → *n* entries → 2*e* size, may have to traverse many elements for access
            - Each head represents a vertex, and the remainder of the linked list contains every vertex that is connected to the vertex
                - Order doesn't matter
        - Preferred representation is dependent on problem/topology of graph

10/21: Sorting

- Given a set of tasks and a set of precedence relations
    - Output is a directed graph
    - Goal to sort by precedence
        - Topological sorting problem
        - Cannot have cycles → no solutions → graph must be acyclic
- Consider a vertex
    - The number of edges coming into it is called the in-degree
        - If the in-degree of the vertex is 0, that vertex is the source
            - In a DAG, there must be at least one source
        - If the outdegree is 0, the vertex is a sink
    - While there is a source // picked arbitrarily
        - The number of edges coming out from it is called the out-degree
        - Find a source
        - Output the source
        - Delete the source and all of its outgoing edges
            - Impossible to create a cycle, since all elements of a cycle have an in-degree >0
    - Given a {b, c, d}

        - Once a is in the output, it can simply be deleted, since any of the following's conditions are already satisfied
        - Doesn't matter which is output first, as long as the output is a source
    - If a is a source, it doesn't depend on anything, if not, it is already accounted for in the first *i* steps
        - Assume the first *i* are in the right place, we can prove a is in the right place
- The in-degree and out-degree of a vertex is at most n - 1 in a graph with n vertices
    - Algorithm to find both is $O(n(n-1) + (n-1)) = O(n^2)$ → go to each vertex and check in and out-degrees
        - Go to each vertex and check if they have an edge to v_i → outdegree
        - Go from v_i and see if there's an edge connecting to each other vertex → indegree
        - Overcounting? → vertex-centric analysis
    - Take an edge *e*, modify the indegree/outdegree of adjacent nodes by incrementing them by 1
        - For each edge in O(e) time, you know the indegrees and outdegrees of each vertex
            - O(e) is better or the same as $O(n^2)$
        - Done through linked list data structure
        - What about a graph with no edges?
            - Must adjust time complexity to O(n + e) → same logic for most cases
        - Go through resulting list, put each node with an indegree of 0 in a source list → O(n) time for setup
            - Proceed through base algorithm
            - How many times do you decrement the indegrees of the vertices
                - Charge the time to the edge → this decrement happens once per edge → the total time it takes is O(e)
            - Algorithm runs in O(e + n)
- 2 coloring problem
    - Color a graph with 2 colors such that no 2 adjacent vertices have the same colors
        - Not possible with C_3, C_5, C_7, etc.
    - Use a BFS → assume no odd cycle
        - Start with a vertex and put it in level 1
            - Put all of its neighbors into level 2
            - Repeat
        - There are no edges within a level where there are no odd cycles
            - Proof:
                - Assume there is an edge within a level, a is connected to b
                - a and b must have been discovered by vertices from the previous level
                - Go until the paths of a and b converge

            - Between every level there would be 2 edges → 2m + 1 edges → odd number, contradiction
        - With this knowledge, we can color each level an alternating color → 2 coloring has been completed
    - Algorithm is BFS with a final assignment of colors at the end, same runtime as BFS

10/26: Single Source Shortest Path

- Midterm: Ch. 1, 2.1, 2.2, 2.3, Ch. 3, 4.4
- Need to differentiate between shortest path of weighted and unweighted graph
- Assume no negative edge weights, all weights are positive integers, connected graph
    - BFS doesn't work → path length is determined by weights, not number of edges
    - Find the lowest weight from source to child node
        - The shortest path from the source to that child node is through that edge
            - Can make this claim since there are no negative weights → all other weights are greater
    - Whenever you finalize a vertex's shortest weight, you update all its connected vertices with new shortest paths
        - Change all neighbors of x → at most n - 1
            - In the entire algorithm, the total number of neighbors is e
            - Total update time is O(e)
    - When you choose a vertex, you must find the minimum value that is still needed to check
        - Use a heap to track the minimums → the minimum is stored at the root
        - Finding the minimum takes O(1) time
        - Heap update (heapify) takes O(log n) time
        - Overall in algorithm → O(e log n) time

10/28: Connectivity

- Any time there is a path between 2 vertices, those vertices are connected
    - Undirected:
        - If a and b are connected, and a and d are connected, by transitivity, b and d are connected
            - There may be other paths, but at least the path through a exists
            - All vertices that are connected form a component of the graph
                - Can find through BFS → start at a, everything you discover from a is part of the component of a → move to unvisited vertex
    - Directed:
        - Connectivity's definition is a little muddled
        - a → b: are a and b connected? No
            - Connected components are undefined
        - Strongly connected components: a set of vertices such that you can go from any vertex to any other vertex in that subset

- a and n are in the same connected component k_x and a and m are in the same connected component k_y, and k_x is not k_y → not possible
    - Proof:
        - m can be reached from a and a can be reached from m
        - n can be reached from a and a can be reached from n
        - From m, there is by definition a path to a
        - From a, there is by definition a path to n
        - This means n is reachable from m
        - Similar logic backwards
        - m can be reached from n and n can be reached from m → they are in the same connected component
    - If k_x and k_y contain a vertex in common, they are the same connected component
- Every vertex belongs to 1 and exactly 1 strongly connected component

- Consider a connected, undirected, weighted graph G
    - A subgraph is any subset of vertices and edges in G
        - If you start with G and begin deleting vertices and edges, you end up with a subgraph
    - A tree that connects all vertices is called a spanning tree
        - A tree by definition cannot have a cycle
        - Spanning trees are not unique → graph with a cycle could remove any part of the cycle
            - The number of spanning trees is exponential → takes exponential time → we usually talk about 1 tree
    - Find the minimum spanning tree → minimum total weight among spanning trees
        - Not necessarily unique
        - Algorithm 1 (vertex-centric):
            - G(V, E) → partition into 2 sets of vertices V_1 and V_2
                - Both must contain at least 1 vertex
                - Every vertex must be in exactly one of these sets
                - There are always at least one edge from V_1 to V_2 → graph is connected
                - Assume the weight of all edges are unique
                - Look at all the edges between V_1 and V_2 → make e_{min} the minimum weight edge of these crossing edges
                - Claim - there exists a minimum spanning tree with e_{min} in it - MST Theorem
                    - Proof:

- Assume we have a minimum spanning tree without e_{min}
- Take this tree and put e_{min} in it
    - This by definition creates a cycle
    - Remove the e_x that necessarily goes between V_1 and V_2 within the created cycle
    - Since $e_{min} < e_x$, we've found a smaller tree than the minimum spanning tree
    - By contradiction, there exists a minimum spanning tree with e_{min} in it

- Number of edges between $V_1 = \{v_1\}$ and $V_2 = \{v_2, \ldots, v_n\}$ is $\deg(v_1)$
    - Take some minimum edge leading to v_k from V_2 and move v_k to V_1 → Prim's MST Algorithm
        - $V_1 = \{v_1, v_k\}$ and $V_2 = \{v_2, \ldots, v_{k-1}, v_{k+1}, \ldots, v_n\}$
        - Continue doing this until every node ends up in V_1
        - Main diff. between Prim's and Dijkstra's → Prim's - min. distance to any node in V_1, Dijkstra's - min. distance to source
        - Time complexity → you have to update *e* edges, maintain a heap for V_2 → total O(e log n)

11/4: Minimum Spanning Tree

- Edge-centric MST algorithm - Kruskal's MST Algorithm
    - *m* edges, assume they are in a sorted list → m log m time
    - Take e_1 and place it in the MST → e_1 = (a, b), place a in its own partition, it is the minimum edge between the 2 partitions
    - Next, take e_2 and take one of its vertices and put it in a partition, take the other vertex and put it in its own partition → ensure e_1 is within a single partition
    - As soon as you create a cycle, ignore that edge
    - Algorithm:
        - Sort the edges
        - Consider e_i
        - If e_i creates a cycle with the previous MST edges, ignore it
        - Else, add e_i to MST
- Given a set of numbers: { (1, 5), (3), (7, 8, 9), … } → all unique elements
    - Find → only 2 options, yes or no
    - Union → name a group with the name of one of its elements → (5, 3) → union the group with 5 and the group with 3
        - Once unioned, original sets cannot be returned to
    - Union-Find problem → do an arbitrary amount of unions and finds efficiently
        - Use the union-find data structure → use a rooted tree structure

        - Union - O(1), Find - O(n) → long paths bad
        - Use shallow paths
            - Find - O(1), Union - O(n)
        - Assume trees are balanced → h <= log n_i
            - Find - O(log n)
            - Union
                - Height of new tree is less than or equal to height of the larger of the 2 original trees
                - O(1)
- In Kruskal, how do we find the cycles?
    - Union-Find
    - Originally e log e time
    - For each edge, we do a union and a find → O(log n) time → e log n total
    - Combine → e log e
- Kruskal - e log e
- Prim - n^2 or e log e

11/9: Clustering and Sorting

- Grouping of elements that have some similar characteristic
- Represent elements as nodes, similarities as weighted edges → small weight = more similar
- Come up with a clustering that maximizes the minimum pairwise distance between 2 clusters → objective function
    - Given a weighted graph G and number K, find K clusters abiding by the objective function
- Clustering algorithm is very similar to Kruskal's → stop when k clusters are visible
    - Algorithm generates a bunch of clusters, C_1, …, C_k
    - d* is minimum of distances i, j in Kruskal-based clustering
    - Proof:
        - Claim clustering algorithm is sub-optimal
        - "Correct" clusters are C_1', …, C_k'
        - At least one C_r that is split between 2 or more clusters C_d' and C_e', otherwise the solutions are identical
        - These 2 clusters are connected to each other somehow, through an edge that is contained in C_r
        - d* is going to be unprocessed in Kruskal's → prime clusters are closer to each other than ours
- Sorting as tool into divide and conquer → merge sort
    - Split problem into 2 groups of roughly equal size → likely involves recursive partitioning
    - Split into smallest possible parts
    - Merge lists until returned to full size
    - Count based on how many comparisons it took to place an arbitrary element *z* in final list *F* → gets there because of 1 comparison → runtime is order t + s
    - Merge takes O(n) time
    - Recursive sort takes O(n log n) time
    - Merge sort takes O(n log n)
    - T(n) = 2T(n / 2) + Cn
        - T(n) = 2[2T(n / 4) + Cn / 2] + Cn
        - T(n) = 2^2T(n / 2^2) + 2Cn
        - T(n) = 2^3T(n / 2^3) + 3Cn

        - $T(n) = 2^iT(n / 2^i) + iCn$
            - $n / 2^i = 1 \rightarrow i = \log n$
        - $T(n) = 2^{\log n}T(n / 2^{\log n}) + \log n\ Cn$
        - $T(n) = n\ T(1) + Cn \log n$
        - $T(n) = O(n \log n)$

11/16: Inversion Count and Closest Pair

- Inversion Count
    - Given a permutation of a set of numbers
    - 3 1 2 4
        - 3, 1 and 3, 2 creates an inversion → wrong order
    - 0 inversions in a sorted list
    - n (n - 1) / 2 inversions in a reversed list
    - Assume left and right subarrays have their inversions counted
        - Only have to account for numbers on the left inverted with numbers on the right
    - Also assume left and right are sorted
    - Follow same algorithm as merge sort to merge
        - If the right subarray goes before the left, then there was an inversion
        - Increment by how many elements remain in the left subarray
    - O(n log n) time
- Closest Pair Problem
    - Sort by x-coordinate and y-coordinate
    - Assume left and right are sorted, each have information about the minimum distance between vertices inside each subproblem

11/18: Dynamic Programming

- Partitioning into subproblems
    - Unlinke D&C, these subproblems likely overlap
- Take interval scheduling problem from Greedy section
    - Instead of maximizing number of intervals, we want to give each interval a weight and get a subset that maximizes total weights
    - In a left to right scan, can we say that an interval *i* with weight w_i is in the solution?
        - We don't know
        - There are 2 possibilities:
            - *i* is in the solution
            - *i* is not in the solution
            - Solution → up until f_i
        - Assume that for any subregion smaller than the currently analyzed region, we know the optimal solution
- If *i* is not in the solution, the optimal solution that ends at f_i is the optimal solution that ends at f_i - 1
- If *i* is in the solution, then the optimal solution that ends at f_i is the optimal solution that ends at s_i - 1 + w_i

11/23: Dynamic Programming Again

- Line of best fit given some criteria for minimization
- Minimize error + some constant times the number of line segments
    - Decide how many line segments we need for this to occur
    - Given set of points and parameter $c \rightarrow n$ + 1 size of problem

- Given 2 points *i* and j, where i is to the right of j
    - Assume the optimal solution is known for the region up to j
    - The last line segment will cover i and some number of points back
    - Exhaustively consider i, i - 1, …, j where 1 <= j <= i
    - There are i possibilities for how many points the line segment covers