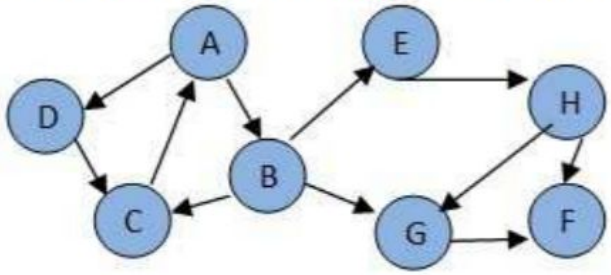
DFS , BFS

- Consider the following graph. If there is ever a decision between multiple neighbor nodes in the BFS or DFS algorithms, assume we always choose the letter closest to the beginning of the alphabet first.



In what order will the nodes be visited using a Breadth First Search? The answer is: ABDCEGHF
In what order will the nodes be visited using a Depth First Search? The answer is: ABCEHFGD

T F Let T be a complete binary tree with n nodes. Finding a path from the root of T to a given vertex $v \in T$ using breadth-first search takes $O(\lg n)$ time.

Solution: **False.** Breadth-first search requires $\Omega(n)$ time. Breadth-first search examines each node in the tree in breadth-first order. The vertex v could well be the last vertex explored. (Also, notice that T is not necessarily sorted.)

1.2. **(2 pt.)** Suppose we run DFS on a graph G. Suppose that v and w are vertices in G, and in our run of DFS we assign start and finish times to these vertices. Which of the following are possible? Circle all that apply.

(A) v.start $\leq$ w.start $\leq$ w.finish $\leq$ v.finish
(B) w.start $\leq$ w.finish $\leq$ v.start $\leq$ v.finish
(C) v.start $\leq$ w.start $\leq$ v.finish $\leq$ w.finish
(D) w.start $\leq$ v.start $\leq$ w.finish $\leq$ v.finish
(E) w.start $\leq$ v.finish $\leq$ v.start $\leq$ w.finish

1.4. **(2 pt.)** Let $G = (V, E)$ be a unweighted directed **acyclic** graph, and let $s \in V$. You want to design a dynamic programming algorithm to find the *longest path* in G that starts at s. You decide to fill in a table L, where $L[x]$ is the cost of the longest path from s to x. Which of the following is the correct recursive structure? Circle exactly one. **(Note: don't worry about how you would actually implement the DP algorithm, that's not part of this problem.)**

(A) $L[x] = \max\{L[u] + 1 : (u, x) \in E\}$
(B) $L[x] = \min\{L[u] - 1 : (u, x) \in E\}$
(C) $L[x] = L[u] + 1$, where u is the vertex that maximizes $L[u]$

2.4. **(4 pt.)** Given an undirected unweighted graph G with maximum degree[1] d and a vertex s, find all of the vertices v with distance at most 6 from s, in time $O(d^6)$.

Run BFS to depth 6.

2.10. **(4 pt.)** *(May be more difficult)* Let $G = (V, E)$ be a directed unweighted graph. Say that G is "kind-of-connected" if for every $u, v \in V$, either there is a path from u to v in G, or there is a path from v to u in G, or both. Decide if G is kind-of-connected in time $O(n + m)$.

Use the SCC algorithm to find the SCC DAG G' of G.

Topologically sort the SCC DAG; say the SCC's are $A_1, A_2, \ldots, A_r$

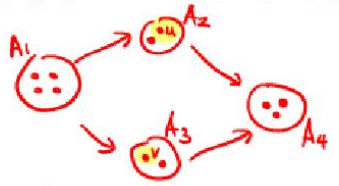
For $i = 1, \_, r-1$:

    if there is no edge from A_i to A_{i+1} in the SCC DAG:

        return FALSE

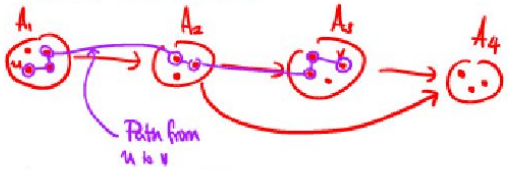
return TRUE

That is, if the SCC DAG looks like



then there's no path from u to v, so we should return FALSE

But if it looks like

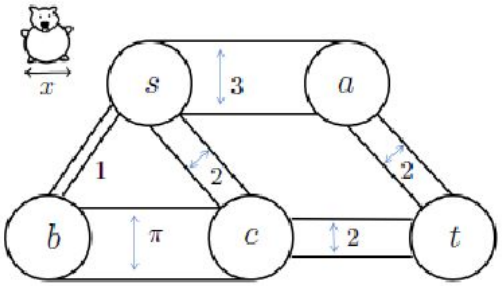


then there's a path from any u to any v.

3 Algorithm Design (33 pts)

3.1. **(11 pt.)** Let $G = (V, E)$ be an undirected, weighted graph, with n vertices and m edges, which represents a network of tunnels belonging to a family of groundhogs.[2] The positive weight $w(u, v)$ on an edge $\{u, v\}$ represents the *width* of the tunnel between u and v. A groundhog of width x can only pass through tunnels of width $\geq x$.

For example, in the graph below, a groundhog with width 2 could get from s to t (either by $s \to a \to t$ or by $s \to c \to t$). However, a groundhog of width 3 could not get from s to t.



3.1.1. **(5 pt.)** Given G, vertices $s, t \in V$, and a desired groundhog width $x > 0$, design an algorithm which finds a path from s to t that a groundhog of width x could fit through, or else reports that no such path exists. Your algorithm should run in time $O(n + m)$. You may use any algorithm we have seen in class as a subroutine.

[We are expecting: Pseudocode, a high-level description of your algorithm, and a short justification of the running time.]

```
def findFatPath(G, s, t, x):
    Let G' be a copy of G w/ all the tunnels of width < x removed.
        \\This takes time O(n+m) if the graph is represented with adjacency lists.
    Run BFS on G'   \\Time O(n+m)
    If BFS finds an s-t path:
        return that path
    Else:
        return "No such path"
```

Description: Remove all the thin edges from G, then run BFS.

Running Time: Copying the graph and running BFS both take time O(m+n).

[Continued on next page]

3.1.2. **(6 pt.) Using your answer to 3.1.1. as a subroutine**, design an algorithm which will find the path from s to t which accomodates the largest groundhog possible. Your algorithm should run in time $O((n+m)\log(m))$. You may use any algorithm we have seen in class as a subroutine.

Note: The tunnel widths are arbitrary real numbers: you cannot assume that they are bounded integers.

[We are expecting: Pseudocode, and a high-level description of your algorithm. Don't worry about floors and ceilings, we will not take off points for small errors like that.]

High-level idea: use binary search and 3.1.1. to find the largest x.
Note that the maximum width x must be equal to one of the edge weights.

```
def findFattestPath(G, s, t):
    Sort the edges E by increasing width.   # Time O(m log(m))
    l=0, h=n-1      # searching the interval {l, l+1, ..., h}
    while h > l:
        mid = ⌈(h+l)/2⌉
        P = findFatPath(s, t, G, width(E[mid]))
        If P == "no such path":  # then mid was too wide! Search below it.
            h = mid - 1
        else:                    # then mid worked, but there might be a wider solution.
            l = mid
    # after the while loop is done, h = l
    Return findFatPath(s, t, G, width(E[l]))
```

The running time is $O(\log(m)(n+m))$ b/c the while loop runs $O(\log(m))$ times, and each time findFatPath takes time $O(n+m)$.

2. (10 pts) Give an $O(n+m)$ time algorithm to decide whether an undirected graph G contains a cycle of odd length, and, if so, outputs one. Then, argue that your algorithm (i) terminates, (ii) is correct, and (iii) has the claimed running time.

   (Hint: What algorithm from class determines whether an undirected graph contains a cycle of odd length? Modify the algorithm to output the cycle.)

Find-Odd-Cycle(G)

```
1  ▷ S is a Stack
2  Run BFS on G to label all vertices with their distance d(v) from some vertex s
   and their parent in the BFS tree
3  for every edge (v, w) of G
4    do if d(v) = d(w)
5          then while v ≠ w
6                  do Output v
7                     Push(S, w)
8                     v ← v's parent
9                     w ← w's parent
10                 Output v
11                 while S is not empty
12                   do w ←Pop(S)
13                      Output w
14                 exit
15 Report that G contains no odd cycles
```

(i) The algorithm terminates because it runs BFS and then looks at each edge at most once in line 3 and then each vertex at most once when outputting the cycle.

(ii) As we proved in class, if there is an odd cycle then there are two vertices v and w that are adjacent and at the same level of the BFS tree. Each vertex has one parent in the BFS tree, so v and w have a least common ancestor l, and the set of vertices $v \to l \to w$ form an odd cycle. If there is no odd cycle then the algorithm reaches the end because **Report-Bipartiteness**(G) does and then reports that G contains no odd cycles.

(iii) BFS takes $O(n+m)$ time. The algorithm considers every edge at most once in line 3 and every vertex at most once in lines 5—10. Thus, the algorithm runs in linear time.

(b) T F The depth of a breadth-first search tree on an undirected graph $G = (V, E)$ from an arbitrary vertex $v \in V$ is the diameter of the graph G. (The ***diameter*** d of a graph is the smallest d such that every pair of vertices s and t have $\delta(s,t) \leq d$.)

Solution: False. An arbitrary vertex could lay closer to the 'center' of the graph, hence the BFS depth will be underestimating the diameter. For example, in graph $G = (V, E) = (\{a, v, b\}, \{(a, v), (v, b)\})$, a BFS from v will have depth 1 but the graph has diameter 2.

(c) **T** **F** Every directed acyclic graph has exactly one topological ordering.

Solution: False. Some priority constraints may be unspecified, and multiple orderings may be possible for a given DAG. For example a graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (a, c)\})$ has valid topological orderings $[a, b, c]$ or $[a, c, b]$. As another example, $G = (V, E) = (\{a, b\}, \{\})$ has valid topological orderings $[a, b]$ or $[b, c]$.

(g) **T** **F** If a depth-first search on a directed graph $G = (V, E)$ produces exactly one back edge, then it is possible to choose an edge $e \in E$ such that the graph $G' = (V, E - \{e\})$ is acyclic.

Solution: True. Removing the back edge will result in a graph with no back edges, and thus a graph with no cycles (as every graph with at least one cycle has at least one back edge). Notice that a graph can have two cycles but a single back edge, thus removing *some* edge that disrupts that cycle is insufficient, you have to remove specifically the back edge. For example, in graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (b, c), (a, c), (c, a)\})$, there are two cycles $[a, b, c, a]$ and $[a, c, a]$, but only one back edge (c, a). Removing edge (b, c) disrupts one of the cycles that gave rise to the back edge ($[a, b, c, a]$), but another cycle remains, $[a, c, a]$.

(h) **T** **F** If a directed graph G is cyclic but can be made acyclic by removing one edge, then a depth-first search in G will encounter exactly one back edge.

Solution: False. You can have multiple back edges, yet it can be possible to remove one edge that destroys all cycles. For example, in graph $G = (V, E) = (\{a, b, c\}, \{(a, b), (b, c), (b, a), (c, a)\})$, there are two cycles ($[a, b, a]$ and $[a, b, c, a]$) and a DFS from a in G returns two back edges ((b, a) and (c, a)), but a single removal of edge (a, b) can disrupt both cycles, making the resulting graph acyclic.

(b) What is the running time of depth-first search, as a function of $|V|$ and $|E|$, if the input graph is represented by an adjacency matrix instead of an adjacency list?

Solution: DFS visits each vertex once and as it visits each vertex, we need to find all of its neighbors to figure out where to search next. Finding all its neighbors in an adjacency matrix requires $O(V)$ time, so overall the running time will be $O(V^2)$.

2 points were docked for answers that didn't give the tightest runtime bound, for example $O(V^2 + E)$. While technically correct, it was a key point to realize that DFS using an adjacency matrix doesn't depend on the number of edges in the graph.

Dynamic Programming

T F If a dynamic-programming problem satisfies the optimal-substructure property, then a locally optimal solution is globally optimal.

Solution: **False.** The property which implies that locally optimal solutions are globally optimal is the *greedy-choice property*. Consider as a counterexample the edit distance problem. This problem has optimal substructure, as was shown on the problem set, however a locally optimal solution—making the best edit next—does not result in a globally optimal solution.

2.7. **(4 pt.)** On a hypothetical final exam, there are n problems. Problem i is worth p_i points, and it takes t_i minutes to complete, where t_i is a positive integer. You have T minutes total to take the exam. The numbers (p_i, t_i) are released well before the exam starts. Given these numbers, in time $O(nT)$, find a set of problems that you can solve in T minutes in order to maximize the points that you receive during the exam. (Assume that there is no partial credit on this hypothetical exam, although there may be partial credit given on the real-life exam).

This is exactly the 0/1 knapsack problem.
Use the DP solution we saw in class.

[TRUE/FALSE] Any Dynamic Programming algorithm with n subproblems will run in O(n) time.

"False. The subproblems may take longer than constant time to compute, as was the case with with longest increasing sub sequence"

[TRUE/FALSE] If a dynamic programming solution is set up correctly, i.e. the recurrence equation is correct and each unique sub-problem is solved only once (memoization), then the resulting algorithm will always find the optimal solution in polynomial time.

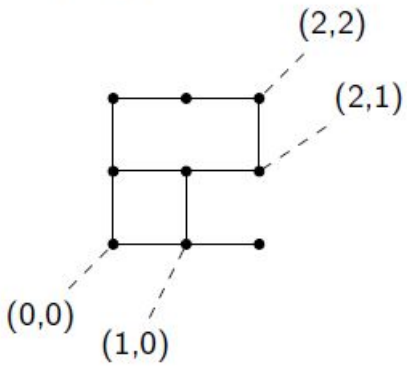
false, Example: TSP be el dynamic programming O(2^n * n^2)

- Suppose that roads in a city are laid out in an $n \times n$ grid, but some of the roads are obstructed.
- For example, for $n = 3$, the city may look like this:

Define M[i,j] = #paths from (0,0) to (i,j).

M[i,j] = 𝟙{(i-1,j)→(i,j)} · M[i-1,j] + 𝟙{(i,j-1)→(i,j)} · M[i,j-1]

Alg:
- Initialize M[i,j] = 0 ∀ i,j ∈ {0,...,n-1}
- M[0,0] ← 1
- for i = 0,...,n-1:
  - for j = 0,...,n-1:
    - if i > 0 and there is a road (i-1,j)→(i,j): M[i,j] += M[i-1,j]
    - if j > 0 and there is a road (i,j-1)→(i,j): M[i,j] += M[i,j-1]
- Return M[n-1,n-1]



where we have only drawn the roads that are not blocked. You want to count the number of ways to get from $(0, 0)$ to $(n - 1, n - 1)$, using paths that only go up and to the right. In the example above, the number of paths is 3.

- Design a DP algorithm to solve this problem.

Shortest Path

MIT 2015

(f) **T** **F** [4 points] Recall the $O(n^3 \lg n)$ matrix-multiplication algorithm to compute shortest paths, where we replaced the matrix-multiplication operator pair $(*, +)$ with $(+, \min)$. If we instead replace the operator pair with $(+, *)$, then we compute the product of the weights of all paths between each pair of vertices.

Solution: False. If the graph has a cycle, there are infinitely many paths between some pairs of vertices, so the product ought to be $\pm\infty$, yet the matrix-multiplication algorithm will compute finite values if the original matrix has all finite values (e.g., a clique).

(i) T F [4 points] In the recursion of the Floyd–Warshall algorithm:

$$d_{uv}^{(k)} = \min\{d_{uv}^{(k-1)}, d_{uk}^{(k-1)} + d_{kv}^{(k-1)}\},$$

$d_{uv}^{(k)}$ represents the length of the shortest path from vertex u to vertex v that contains at most k edges.

Solution: False. $d_{uv}^{(k)}$ is the length of the shortest path from vertex u to vertex v that only uses vertex $\{1, 2, \cdots k\}$ as intermediate nodes.

(d) Consider the following algorithm, which is intended to compute the shortest distance among a collection of points in the plane:

```
1  Sort all points by their y-coordinate.
2  for each point in the sorted list:
3      Compute the distance to the next 7 points in the list.
4  return the smallest distance found.
```

Give an example where this algorithm will return a distance that is not in fact the overall shortest distance.

Solution: To construct a counterexample, it is sufficient to construct an example where the shortest distance is between two points that have at least 7 points between them in the ordering of y-coordinates. The example we use is as follows:

$$(0, -1) \quad (0, 5) \quad (0, 10) \quad (0, 15) \quad (0, 20) \quad (0, 25) \quad (0, 30) \quad (0, 35) \quad (0, 1)$$

In any ordering of these points by y-coordinate, the seven points with y-coordinate 0 will lie between $(0, -1)$ and $(0, 1)$. Hence, the algorithm will never compute the Euclidean distance between $(0, -1)$ and $(0, 1)$, and so it cannot find the true shortest distance.

Problem 3. Estate Showing. [30 points] (3 parts)

Trip Trillionaire is planning to give potential buyers private showings of his estate, which can be modeled as a weighted, directed graph G containing locations V connected by one-way roads E. To save time, he decides to do k of these showings at the same time, but because they were supposed to be private, he doesn't want any of his clients to see each other as they are being driven through the estate.

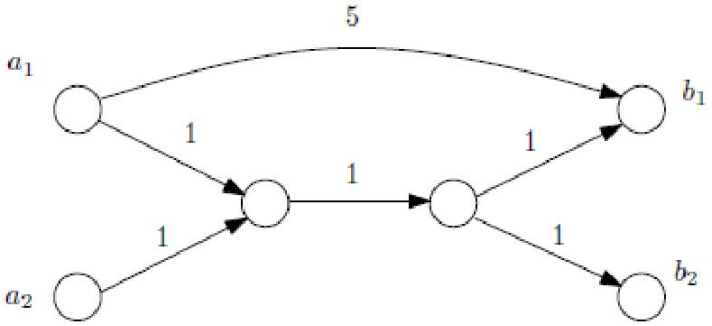
Trip has k grand entrances to his estate, $A = \{a_1, a_2, \ldots, a_k\} \subset V$. He wants to take each of his buyers on a path through G from their starting location a_i to some ending location in $B = \{b_1, b_2, \ldots, b_k\} \subset V$, where there are spectacular views of his private mountain range.

Because of your prowess with algorithms, he hires you to help him plan his buyers' visits. His goal is to find a path for each buyer i from the entrance they take, a_i, to any ending location b_j such that no two paths share any edges, and no two buyers end in the same location b_j.

(a) Trip tells you his idea: find all-pairs shortest paths on G, and then try to select k of those shortest paths $a_i \rightsquigarrow b_j$ such that all k paths start and end at different vertices and no two paths share any edges.

Give a graph where there exists a set of paths satisfying the requirements, but where Trip's strategy won't find it.

Solution: Consider this graph:



The all-pairs shortest paths algorithm would find all shortest paths from a_i to b_j ($a_1 \rightsquigarrow b_1$, $a_1 \rightsquigarrow b_2$, $a_2 \rightsquigarrow b_1$, and $a_2 \rightsquigarrow b_2$), which all go through the same edge. Trip's algorithm, which considers only those paths, would find no solution. There are two completely disjoint paths using the shortest path $a_2 \rightsquigarrow b_2$ and the direct edge (a_1, b_1), but Trip's algorithm would not find this combination because one of them is not a shortest path.

T F Let $G = (V, E)$ be a directed graph with negative-weight edges, but no negative-weight cycles. Then, one can compute all shortest paths from a source $s \in V$ to all $v \in V$ faster than Bellman-Ford using the technique of reweighting.

Solution: **False.** The technique of reweighting preserves the shortest path by assigning a value $h(v)$ to each vertex $v \in V$, and using this to calculate new weights for the edges: $\tilde{w}(u, v) = w(u, v) + h(u) - h(v)$. However, to determine values for $h(v)$ such that the edge weights are all non-negative, we use Bellman-Ford to solve the resulting system of difference constraints. Since the technique of reweighting relies on Bellman-Ford, it cannot run faster than Bellman-Ford.

2.3. (4 **pt.**) Given a weighted directed graph G with non-negative edge weights, and given vertices s and t, deterministically find a shortest path from s to t in time $O((m+n)\log(n))$.

Use Dijkstra's Algorithm

Question 7: Strenuous Trail (30 points)

You are trying to find the most strenuous hiking trail between two points: the path with the most elevation gain or loss per mile. We model this problem as follows: You are given a connected undirected graph $G = (V, E)$ with two designated nodes $a, b \in V$, where each node v is associated with a height $h(v) \in \mathbb{R}$ and each edge e is associated with a positive length $w(e)$. In addition, $h(a) = h(b) = 0$.

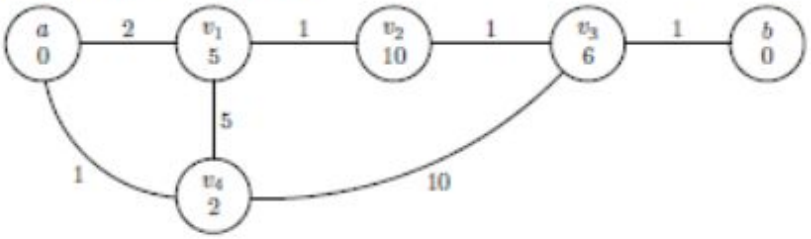
We say that a path from a to b is an *up-and-down* path if there is some node v in the path such that all the nodes from a to v have increasing heights, and all the nodes from v to b have decreasing heights. (Figuratively, think about climbing a hill where v is the top of the hill.)

Formally, let $p = (v_1, v_2, \ldots, v_{k-1}, v_k)$ be a path. The path p is up-and-down if there is some vertex v_i $(1 < i < k)$, such that for all $1 \leq j < i$, $h(v_j) < h(v_{j+1})$ and for all $i \leq j < k$, $h(v_j) > h(v_{j+1})$. The path may not be a simple path and can visit vertices multiple times (technically this is called a walk). We introduce the following definitions:

- The length of p is the sum of the lengths of the edges of the path: $\text{length}(p) = \sum_{i=1}^{k-1} w((v_i, v_{i+1}))$.
- The elevation of p is defined as the maximum height of a node in the path: $\text{elevation}(p) = \max_{1 \leq i \leq k} h(v_i)$.
- The difficulty of p is $\text{difficulty}(p) = \frac{\text{elevation}(p)}{\text{length}(p)}$.

Design an $O(|E| + |V| \log |V|)$ time algorithm that finds an up-and-down path from a to b that has the maximum difficulty or reports that there is no up-and-down path from a to b. Briefly argue correctness and analyze running time.

Example: Consider the following graph.



The number inside each node is its height (for example, $h(v_1) = 5$), and the number next to each edge is its length (for example, $w((s, v_1)) = 2$).

The output of the algorithm is the most difficult up-and-down path from a to b, which is (a, v_1, v_2, v_3, b). The elevation of the path is 10, its length is 5, and its difficulty is 2. The path (a, v_4, v_3, b) is another up-and-down path from a to b that has difficulty 1/2. The path (a, v_1, v_4, v_3, b) is not an up-and-down path.

Hint: If you are looking for an a-b path of maximum difficulty where the vertex with the highest height is v, how could you find the portion of the path from a to v and the portion of the path from v to b?

Question 7

The main observation is that the most difficult up-and-down path should consist of a shortest "ascending" path from a to the node, say v, of the max height on p and then a shortest "descending" path from v to b. Consider the following algorithm:

Algorithm 1: FINDPATH$(G(V,E), w, h)$

1. Construct a directed graph $G' = (V, E')$ where for each $e \in E$ where $e = (u, v)$, there is a directed edge $e' = (u, v)$ in E' only if $h(u) < h(v)$.
2. Run Dijkstra's algorithm to find single-source shortest paths from a on G'. Let d_1 and π_1 be the shortest path distances and predecessors.
3. Run Dijkstra's algorithm to find single-source shortest paths from b on G'. Let d_2 and π_2 be the shortest path distances and predecessors.
4. If $d_1(v) = \infty$ or $d_2(v) = \infty$ for all $v \in V \setminus \{a, b\}$
5. Return "No up-and-down path from a to b"
6. Else
7. Let $v' = \arg\max_{v \in V \setminus \{a,b\}} \frac{h(v)}{d_1(v)+d_2(v)}$
8. Construct the shortest paths p_1 and p_2 to v' using π_1 and π_2, respectively.
9. Return $p_1 \cup \text{reverse}(p_2)$

Correctness: Assume an up-and-down path from a to b exists and let p^* be the most difficult up-and-down path among such up-and-down paths. We show that the algorithm FINDPATH returns an up-and-down path with difficulty at least difficulty(p^*). By the definition of p^*, it would follow that the returned path has difficulty exactly equal to difficulty(p^*).

We note that p^* consists of a shortest "ascending" path p^+ from a to the node, say v^*, of the max height on p^* and a shortest "descending" path p^- from v^* to b, where ascending (descending) paths are on edges in the direction of increasing (decreasing) node heights. Otherwise, we can improve the difficulty of p^* by replacing p^+ or p^- with a shorter path.

By the construction of G' in Step 1 and Dijkstra's algorithm in Steps 2 and 3, we compute shortest ascending paths from a to v^* and from b to v^*. Reversing the direction of the shortest ascending path from b to v^*, we get a shortest descending path from v^* to b. It follows that FINDPATH finds an up-and-down path through v^* with the same difficulty as p^*. Then, FINDPATH returns an up-and-down path with difficulty at least difficulty(p^*) in Steps 7 - 9.

T F Dijkstra's algorithm is always correct even in graphs with negative edge weights.

T F Prim's algorithm is always correct even in graphs with negative edge weights.

T F Using a suitable data structure, Dijkstra's algorithm can be implemented in $O(m \log n)$ time in graphs with m edges and n vertices.

T F Using a suitable data structure, Prim's algorithm can be implemented in $O(m \log n)$ time in graphs with m edges and n vertices.

(d) **T** **F** Given a graph $G = (V, E)$ with positive edge weights, the Bellman-Ford algorithm and Dijkstra's algorithm can produce different shortest-path trees despite always producing the same shortest-path weights.

Solution: True. Both algorithms are guaranteed to produce the same shortest-path weight, but if there are multiple shortest paths, Dijkstra's will choose the shortest path according to the greedy strategy, and Bellman-Ford will choose the shortest path depending on the order of relaxations, and the two shortest path trees may be different.

(e) **T** **F** Dijkstra's algorithm may not terminate if the graph contains negative-weight edges.

Solution: False. It always terminates after $|E|$ relaxations and $|V|+|E|$ priority queue operations, but may produce incorrect results.

(f) **T** **F** Consider a weighted directed graph $G = (V, E, w)$ and let X be a shortest s-t path for $s, t \in V$. If we double the weight of every edge in the graph, setting $w'(e) = 2w(e)$ for each $e \in E$, then X will still be a shortest s-t path in (V, E, w').

Solution: True. Any linear transformation of all weights maintains all relative path lengths, and thus shortest paths will continue to be shortest paths, and more generally all paths will have the same relative ordering. One simple way of thinking about this is unit conversions between kilometers and miles.

Greedy Algorithm

Problem 5. Piano Recital [15 points] (3 parts)

Prof. Chopin has a piano recital coming up, and in preparation, he wants to learn as many pieces as possible. There are m possible pieces he could learn. Each piece i takes p_i hours to learn.

Prof. Chopin has a total of T hours that he can study by himself (before getting bored). In addition, he has n piano teachers. Each teacher j will spend up to t_j hours teaching. The teachers are very strict, so they will teach Prof. Chopin only a single piece, and only if no other teacher is teaching him that piece.

Thus, to learn piece i, Prof. Chopin can either (1) learn it by himself by spending p_i of his T self-learning budget; or (2) he can choose a unique teacher j (not chosen for any other piece), learn together for $\min\{p_i, t_j\}$ hours, and if any hours remain ($p_i > t_j$), learn the rest using $p_i - t_j$ hours of his T self-learning budget. (Learning part of a piece is useless.)

(a) [6 points] Assume that Prof. Chopin decides to learn exactly k pieces. Prove that he needs to consider only the k lowest p_is and the k highest t_js.

Solution: Assume there exists a selection of teachers and pieces for learning k pieces. Let the set of lowest k pieces be P_k. If there is a piece in our selection that is $\notin P_k$, then we must have a piece in P_k not in the final selection. If we swap the one with the higher cost ($\notin P_k$) with the one with lower cost ($\in P_k$), the new selection thus made will still be valid, because if the higher time cost was fulfilled in the previous selection, the lower time cost in the new selection will still be fulfilled. In this way, we can swap pieces until all of them are $\in P_k$.

Similarly, we can swap the teachers for those of higher value until they are the ones with the k highest times.

(b) [5 points] Assuming part (a), give an efficient greedy algorithm to determine whether Prof. Chopin can learn exactly k pieces. Argue its correctness.

Solution: Let us sort all the teachers and pieces in increasing order beforehand. Call the sorted lists P and T. We see that if a solution exists, there is also one in which P_1 is paired with T_{n-k+1}, P_2 is paired with T_{n-k+2} and so on.

So for each $1 \leq i \leq k$, the greedy algorithm checks if $P_i \leq T_{n-k+i}$. If it is, then we don't need to use the shared time for this piece. If it is not, we need to use $T_{n-k+i} - P_i$ of the shared time. We can add up these values. In the end, if the total shared time we need is $> T$, we return false. Otherwise, we return true.

This takes $O(k)$ time, apart from the initial sorting.

(c) [4 points] Using part (b) as a black box, give an efficient algorithm that finds the maximum number of pieces Prof. Chopin can learn. Analyze its running time.

Solution: Notice that if k_{max} is the maximum value of pieces we can learn, we can also learn k pieces for any $k \leq k_{max}$. This suggests that we binary search over the value of k.

We try $O(\log n)$ values during the binary search, and checking each value takes $O(n)$ time. This takes $O(n \log n)$ time. The sorting also took $O(n \log n)$ time, so the algorithm takes $O(n \log n)$ time overall.

Problem 8. Load Balancing [15 points] (2 parts)

Suppose you need to complete n jobs, and the time it takes to complete job i is t_i. You are given m identical machines $M_1, M_2, \ldots, M_m$ to run the jobs on. Each machine can run only one job at a time, and each job must be completely run on a single machine. If you assign a set $J_j \subseteq \{1, 2, \ldots, n\}$ of jobs to machine M_j, then it will need $T_j = \sum_{i \in J_j} t_i$ time. Your goal is to partition the n jobs among the m machines to minimize $\max_i T_i$.

(a) [5 points] Describe a greedy approximation algorithm for this problem.

Solution: Let J_j to be the set of jobs that M_j will run, and T_j to be the total time it machine M_j is busy (i.e., $T_j = \sum_{i \in J_j} t_i$). Initially, $J_j = \emptyset$, and $T_j = 0$ for all j.

For $i = 1, \ldots, n$, assign job i to machine M_j such that $T_j = \min_{1 \le k \le m}(T_k)$. That is, $J_j = J_j \cup i$ and $T_j = T_j + t_i$. Output J_j's.

This runs in $O(n \lg m)$ time by keeping a min-heap of the machines based on the current total runtime of each machine.

Solution: Alternate solution: Sort jobs in non-increasing order. Without loss of generality, let the jobs in order be $t_1, \ldots, t_n$. Let $J_j = \{t_{k:k \equiv j \mod m}\}$. Variations of this algorithm also works, with different sorting orders and assignments. This takes $O(n \lg n)$ time to sort the jobs.

(b) Suppose we apply Huffman coding to an alphabet of size 4, and the resulting tree is a perfectly balanced binary tree (one root with two children, each of which has two children of its own). Find the maximum frequency of any letter.

Solution: The maximum frequency of any letter is $2/5$. To see why, we must prove that this is both feasible, and as large as possible.

1. Suppose that we have four characters a, b, c, d with frequencies $f_a = 2/5$ and $f_b = f_c = f_d = 1/5$. Without loss of generality, suppose that Huffman's algorithm merges c and d. Then $f_{cd} = f_c + f_d = 2/5$. This introduces a tie[1], which we may break by assuming that Huffman's algorithm chooses to merge b with a. This yields a perfectly balanced tree.
2. To see that any frequency greater than $2/5$ is impossible, let $f_a \ge f_b \ge f_c \ge f_d$ be the frequencies in decreasing order. We define these frequencies so that $f_a + f_b + f_c + f_d = 1$. For the sake of contradiction, suppose that $f_a > 2/5$. This means that $f_b + f_c + f_d < 3/5$. Because $f_b > f_c, f_d$, we can conclude that $f_c + f_d < 2/5$. But this means that when c and d are merged, their combined frequency will be strictly less than f_a, so we will not get a perfectly balanced tree.

Max value > 2* one of the small values

X> 2 $\frac{1-X}{3}$

Problem 4. Credit Card Optimization [30 points] (3 parts)

Zelda Zillionaire is planning on making a sequence of purchases with costs $x_1, \ldots, x_n$. Ideally, she would like to make all of these purchases on one of her many credit cards. Each credit card has credit limit ℓ. Zelda wants to minimize the number of credit cards that she needs to use to make these purchases, without exceeding the credit limit on any card. More formally, she wants to know the smallest k such that there exists $\pi : \{1, \ldots, n\} \to \{1, \ldots, k\}$ assigning each purchase i to a credit card j, where π must satisfy the following constraint:

$$\forall j \in \{1, \ldots, k\} : \sum_{\substack{i \in \{1,\ldots,n\} \\ \text{s.t. } \pi(i)=j}} x_i \leq \ell.$$

Zelda is thinking of using the following algorithm to distribute her purchases:

```
1  Create an empty list of credit cards L.
2  for each purchase x_i:
3      for each credit card j ≤ |L|:
4          if L[j] + x_i ≤ ℓ:
5              Purchase x_i with credit card j.
6              Set the balance for card j to L[j] = L[j] + x_i.
7              Skip to the next purchase.
8      if no existing credit card has enough credit left:
9          Purchase x_i with a new credit card.
10         Append a new credit card to L.
11         Set the balance of the new credit card to x_i.
12 return k = |L|
```

(a) Give an example where Zelda's algorithm will not use the optimal number of credit cards.

Solution:

Let the credit limit l be 10. Consider the following sequence of purchases as input to Zelda's algorithm: $2, 1, 9, 8$ The optimal solution would only use 2 credit cards by grouping purchases $\{2, 8\}$ and $\{9, 1\}$. Zelda's algorithm would split the charges into 3 cards: $\{2, 1\}$, $\{9\}$, $\{8\}$, which is suboptimal.

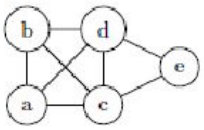
This problem part was worth 5 points. Almost everyone got this problem right. Some students didn't read carefully the pseudocode and inverted the order of the two loops.

Soln: sort descending

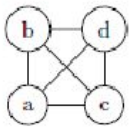
3.2. **(11 pt.)** Let $G = (V, E)$ be an undirected, unweighted graph. For a subset $S \subseteq V$, define the **subgraph induced by** S to be the graph $G' = (S, E')$, where $E' \subseteq E$, and an edge $\{u, v\} \in E$ is included in E' if and only if $u \in S$ and $v \in S$.

For any $k < n$, say that a graph G is k-well-connected if every vertex has degree at least k. (That is, if there are least k edges coming out of each vertex).

For example, in the graph G below, the subgraph G' induced by $S = \{a, b, c, d\}$ is shown on the right. G' is 3-well-connected, since every vertex in G' has degree at least 3. However, G is not 3-well-connected since vertex E has degree 2.



$G = (V, E)$



$G' = (S, E')$, for $S = \{a, b, c, d\}$

Observation: If G' is a k-well-connected subgraph induced by S, and $v \in V$ has degree $< k$, then $v \notin S$. This is because v would have degree $< k$ in the induced subgraph G' as well, and so G' couldn't be k-well-connected if it included v.

3.2.1. **(9 pt.)** Guided by the **observation** above, design a greedy algorithm to find a maximal set $S \subseteq V$ so that the subgraph $G' = (S, E')$ induced by S is k-well-connected.

In the example above, if $k = 3$, your algorithm should return $\{a, b, c, d\}$, and if $k = 4$ your algorithm should return the empty set.

You may assume that your representation of a graph supports the following operations:

- `degree(v)`: return the degree of a vertex in time $O(1)$
- `remove(v)`: remove a vertex and all edges connected to that vertex from the graph, in time $O(\text{degree}(v))$.

Your algorithm should run in time $O(n^2)$.

[We are expecting: Pseudocode and English description of what your algorithm is doing, as well as a justification of the running time. You do not need to prove that your algorithm is correct.]

Idea: greedily remove all the low-degree vertices.

```
def findWellConnectedSubgraph(G, k):
    while(TRUE):
        If there is a vertex with degree(v) < k:   \\ takes time O(n) to run degree(v) for all v.
            remove(v)   \\ O(deg(v)) ≤ O(k)
        Else:
            break
    Return G.
```

[More space on next page]

Running time:

The while loop runs at most n times.

Each iteration uses $O(n + k) = O(n)$ time.

(n: to find a vertex with degree $< k$; k: to delete that vertex)

So the total running time is $O(n^2)$

3.2.2. **(2 pt.)** You do not need to formally prove why your algorithm is correct, but give an informal but convincing justification. (A few sentences should be enough).
[We are expecting: a few sentences about why your algorithm is correct.]

This algorithm is correct because at each step we are removing a vertex that could not possibly be in a set S so that $G|_S$ is k-well-connected. That is, our greedy choices do not rule out success.

3.3. **(11 pt.)** Suppose that A is an $n \times n$ array that contains only zeros and ones. Your goal is to find the largest square in A that is all ones. That is, you want to find i, j and ℓ so that ℓ is as large as possible and so that the sub-array $A[i-\ell+1 : i+1][j-\ell+1 : j+1]$ is entirely ones. (Notice that this requires $\ell \leq \min(i+1, j+1)$.

For example, if A were the matrix

	$j=0$	$j=1$	$j=2$	$j=3$	$j=4$
$i=4$	1	0	0	0	0
$i=3$	0	1	1	0	1
$i=2$	0	1	1	1	1
$i=1$	1	1	1	1	1
$i=0$	0	1	1	1	1

then you should return $i = 2, j = 4, \ell = 3$, corresponding the the box drawn above. (Here, the lower-left corner of the matrix is indexed as $(0, 0)$).

In the questions on the next two pages, you will design an algorithm to perform this task.

3.3.1. **(5 pt.)** Define a function $F(x, y)$ to be the side length of the largest all-one square whose upper-right corner is (x, y).

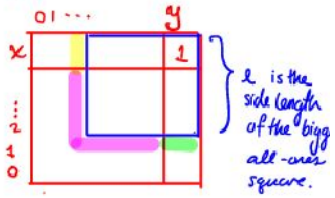
In the example above, $F(2, 4) = 3$, while $F(0, 0) = 0$.

Give an equation[3] that expresses $F(x, y)$ in terms of $F(x-1, y), F(x, y-1), F(x-1, y-1)$ and $A[x, y]$, and explain why your equation is correct.

[We are expecting: An equation and an informal but convincing argument that it is correct.]

First, if $A[x,y] = 0$, then $F(x,y) = 0$.

If $A[x,y] = 1$, then let $\ell = F(x,y)$ ⟶

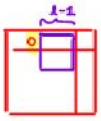


ℓ is the side length of the big all-ones square.

There are 3 cases (non-exclusive):

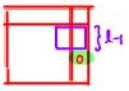
1. There is a zero here.

   In that case, we have $\ell - 1 = F(x, y-1)$



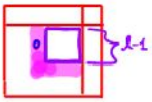
2. There is a zero here

   In that case, we have $\ell - 1 = F(x-1, y)$



3. There is a zero somewhere here

   In that case, we have $\ell - 1 = F(x-1, y-1)$



Notice that we always have

$\ell - 1 \leq F(x-1, y-1)$,
$\ell - 1 \leq F(x, y-1)$,
$\ell - 1 \leq F(x-1, y)$,

and these 3 cases say that at least one of these is an equality.

Between all of these,

$$F(x,y) = A[x,y] \cdot \left(\min\left\{ F(x,y-1), F(x-1,y), F(x-1,y-1) \right\} + 1 \right)$$

[Another part on next page]

3.3.2. **(6 pt.)** Give an algorithm to return the largest all-one square. Your algorithm should take as input an $n \times n$ array A, and should return i, j, ℓ as described above. Your algorithm should run in time $O(n^2)$.

[We are expecting: Pseudocode along with an English description of the idea of your algorithm, as well as a short justification of the running time.]

Idea: We use the formula from the previous part to fill in an $n \times n$ table.

```
def findBigSquare(A):
    F = n×n array full of 0's
    for i = 0, ..., n-1:
        F[i, 0] = A[i, 0]
        F[0, i] = A[0, i]
    for x = 1, ..., n-1:
        for y = 1, ..., n-1:
            F[x, y] = A[x, y] · (min{F[x-1, y], F[x, y-1], F[x-1, y-1]} + 1)   (*)
    x*, y* = argmax_{x,y} F[x, y]
    Return x*, y*, F[x*, y*]
```

initialize these entries.

Running time: The running time is dominated by the two for loops, and (*) can be done in $O(1)$ time, so the running time is $O(n^2)$.

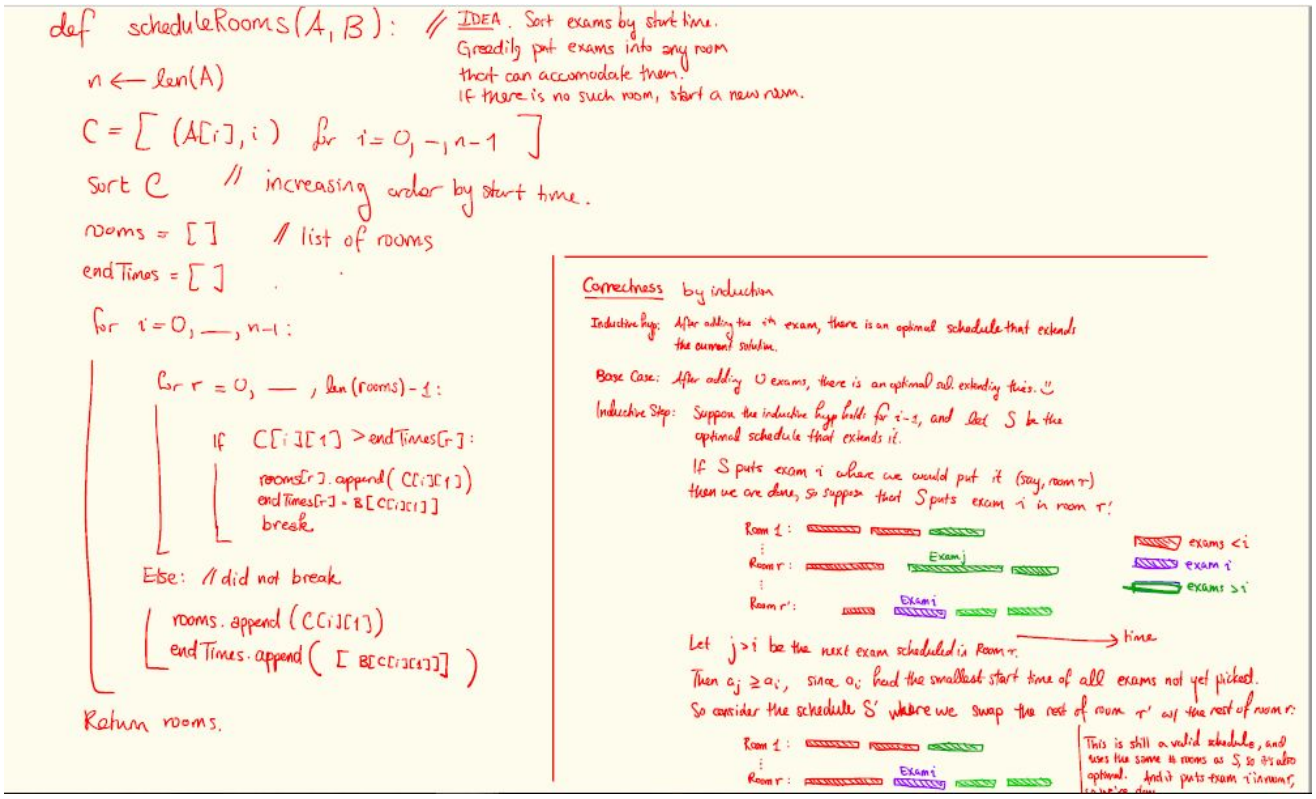
There are n final exams on Dec. 13 at Stanford; exam i is scheduled to begin at time a_i and end at time b_i. Two exams which overlap cannot be administered in the same classroom; two exams i and j are defined to be *overlapping* if $[a_i, b_i] \cap [a_j, b_j] \neq \emptyset$ (including if $b_i = a_j$, so one starts exactly at the time that the other ends). Design an algorithm which solves the following problem.

- **Input:** Arrays A and B of length n so that $A[i] = a_i$ and $B[i] = b_i$.
- **Output:** The smallest number of classrooms necessary to schedule all of the exams, and an optimal assignment of exams to classrooms.
- **Running time:** $O(n \log(n) + nk)$, where k is the minimum number of classrooms needed.
- **For example:** Suppose there are three exams, with start and finish times as given below:

i	1	2	3
a_i	12pm	4pm	2pm
b_i	3pm	6pm	5pm

Solution on next page

Then the exams can be scheduled in two rooms; Exam 1 and Exam 2 can be scheduled in Room 1 and Exam 3 can be scheduled in Room 2.



NP Hard

Problem 7. Startups are Hard [20 points] (3 parts)

For your new startup company, *Uber for Algorithms*, you are trying to assign projects to employees. You have a set P of n projects and a set E of m employees. Each employee e can only work on one project, and each project $p \in P$ has a subset $E_p \subseteq E$ of employees that must be assigned to p to complete p. The decision problem we want to solve is whether we can assign the employees to projects such that we can complete (at least) k projects.

(a) [5 points] Give a straightforward algorithm that checks whether any subset of k projects can be completed to solve the decisional problem. Analyze its time complexity in terms of m, n, and k.

Solution: For each $\binom{n}{k}$ subsets of k projects, check whether any employee is required by more than one project. This can be done simply by going each of the k projects p, marking the employees in E_p as needed, and if any employee is marked twice, then this subset fails. Output "yes" if any subset of k project can be completed, and "no" otherwise.

The time complexity is $\binom{n}{k} \cdot m$ because there are $\binom{n}{k}$ subsets of size k and we pay $O(m)$ time per subset (because all but one employee will be marked only once). Asymptotically, this is $(n/k)^k m$.

(c) [10 points] Show that the problem is NP-hard via a reduction from 3D matching.
Recall the 3D matching problem: You are given three sets X, Y, Z, each of size m; a set $T \subseteq X \times Y \times Z$ of triples; and an integer k. The goal is to determine whether there is a subset $S \subseteq T$ of (at least) k disjoint triples.

Solution: Each $(x, y, z) \in T$ becomes a project that requires employees $E_{(x,y,z)} = \{e_x, e_y, e_z\}$. Thus $n = |T|$, $E = X \cup Y \cup Z$, and $m = |X| + |Y| + |Z|$. We set k to be the same in both problems. The size of the matching is equal to the number of projects that can be completed because both problems model disjointness: if k projects can be completed, a subset S of size k can be found, and vice versa. The reduction takes polynomial time.

(a) **T** **F** If a problem has an algorithm that is correct for 2/3 fraction of the inputs, then it also has an algorithm that is correct for 99.9% of the inputs.

Solution: False. If an algorithm is correct for only 2/3 of the *inputs*, then it is not necessarily possible to amplify the results to make it correct for those inputs. (Consider, for example, a problem that is easy on 2/3 of the inputs but hard/undecidable for the remaining 1/3 of the inputs.) An algorithm must be correct for 2/3 of the possible sequences of random choices for amplification to apply.

(g) **T** **F** There is an NP-hard problem with a known polynomial time randomized algorithm that returns the correct answer with probability 2/3 on all inputs.

Solution: False. We currently don't know whether there are NP-hard problems that can be solved by polynomial time randomized algorithms, and conjecture that these do not exist.

(h) **T** **F** Every special case of an NP-hard problem is also NP-hard.

Solution: False. Consider the MST problem, which is a special case of the Steiner Tree problem.

Longest path for DAG can be obtained by negating the weights and solving using MST . However, the longest path for a graph (not directed) is NP hard.

(c) Show that minimizing the number of credit cards used is NP-hard.

Solution:

We prove that minimizing the number of credit cards is NP-hard by reducing SET-PARTITION to the Zelda's problem in polynomial time.

First, we recast Zelda's optimization problem as a decision problem. Let the decision problem be formulated as follows: For a given target number of cards k, we are able to come up with an assignment of purchases $x_1 \dots x_n$ to k credit cards, where the cumulative balance on each credit card can not exceed the limit l.

Recall that we defined SET-PARTITION as follows:

Given set $S = \{s_1, \dots s_n\}$ we can partition them into 2 disjoint subsets A and B such that $\max\{\sum_{i \in A} s_i, \sum_{j \in B} s_j\}$ is minimized.

The decision version of the SET-PARTITION problem can be reformulated as the question whether the numbers can be partitioned into two sets A and $B = S - A$ such that $\sum_{i \in A} s_i = \sum_{j \in B} s_j$.

Given an input set $S = \{s_1, \dots s_n\}$ to the SET-PARTITION problem, we can use the elements in the set as the purchases in Zelda's problem, set the number of cards to be 2 and the credit card limit $l = \frac{1}{2} \sum_{i \in S} s_i$. Clearly, if the answer to the Zelda's decision problem is in the affirmative, then each credit card will have balance of exactly $\frac{l}{2}$, and we can take the disjoint subsets of purchases on each card to be the two sets A and B that would satisfy SET-PARTITION. The reduction can be executed in linear time.

We also accepted the reduction from SUBSET-SUM to Zelda.

We gave 5 points for correctly identifying an NP-hard problem from which one could reduce to Zelda's problem to prove that the problem is NP-hard (the direction of the reduction needed to be correct to receive all the points). 10 points was given for a correct reduction and analysis.

GEEKS FOR GEEKS:

1- Assuming P != NP, which of the following is true ?
(A) NP-complete = NP
(B) NP-complete P =
(C) NP-hard = NP
(D) P = NP-complete
Answer: (B)
Explanation: The answer is B (no NP-Complete problem can be solved in polynomial time). Because, if one NP-Complete problem can be solved in polynomial time, then all NP problems can solved in polynomial time. If that is the case, then NP and P set become same which contradicts the given condition.

2- Let S be an NP-complete problem and Q and R be two other problems not known to be in NP. Q is polynomial time reducible to S and S is polynomial-time reducible to R. Which one of the following statements is true? (GATE CS 2006)
(A) R is NP-complete
(B) R is NP-hard
(C) Q is NP-complete
(D) Q is NP-hard
Answer: (B)
Explanation: (A) Incorrect because R is not in NP. A NP Complete problem has to be in both NP and NP-hard.
(B) Correct because a NP Complete problem S is polynomial time educable to R.
(C) Incorrect because Q is not in NP.
(D) Incorrect because there is no NP-complete problem that is polynomial time Turing-reducible to Q.

3- Let X be a problem that belongs to the class NP. Then which one of the following is TRUE?
(A) There is no polynomial time algorithm for X.
(B) If X can be solved deterministically in polynomial time, then P = NP.
(C) If X is NP-hard, then it is NP-complete.
(D) X may be undecidable.

Answer: (C)

Explanation: (A) is incorrect because set NP includes both P(Polynomial time solvable) and NP-Complete .

(B) is incorrect because X may belong to P (same reason as (A))

(C) is correct because NP-Complete set is intersection of NP and NP-Hard sets.

(D) is incorrect because all NP problems are decidable in finite set of operations.

4- 2-SAT is P while 3-SAT is NP Complete.
5- Which of the following statements are TRUE?
(1) The problem of determining whether there exists a cycle in an undirected graph is in P.
(2) The problem of determining whether there exists a cycle in an undirected graph is in NP.
(3) If a problem A is NP-Complete, there exists a non-deterministic polynomial time algorithm to solve A.
(A) 1, 2 and 3
(B) 1and 3
(C) 2 and 3

(D) 1 and 2
Answer: (A)
Explanation: 1 is true because cycle detection can be done in polynomial time using DFS. **2** is true because P is a subset of NP. **3** is true because NP complete is also a subset of NP and NP means **N**on-deterministic **P**olynomial time solution exists.

QUIZ MIT

1. True, False, or Unknown: P $\neq$ NP.

Unknown; this is generally believed to be true, but it might not be.

2. True, False, or Unknown: The Hamiltonian path problem for undirected graphs is in P (i.e., UHAMPATH=$\{\langle G, s, t\rangle \mid G$ is an *undirected* graph with a Hamiltonian path from s to $t\}$).

Unknown; UHAMPATH is NP-complete. If P=NP, then UHAMPATH is in P.
If P$\neq$NP, then UHAMPATH is not in P.

3. True, False, or Unknown: NP $\cap$ coNP = P.

Unknown; we know that P is contained within NP $\cap$ coNP, but these sets might not be equal.

4. True, False, or Unknown: If SAT $\in$ P, then coNP $\neq$ P.

False; SAT $\in$ P implies that $\overline{SAT} \in$ P. (Recall that P is closed under complement.)
Since $\overline{SAT}$ is coNP-complete, this would mean that coNP=P.

Problem 3: **(10 points)** Suppose that L_1, L_2, and L_3 are nontrivial languages over $\Sigma = \{0, 1\}$. Prove that if:

1. $L_1 \leq_P L_2 \cap L_3$,
2. $L_2 \in$ NP, and
3. $L_3 \in$ P,

then $L_1 \in$ NP. You may invoke theorems proved in class and in the book, but if you do this, cite them explicitly.

Since $L_3 \in$ P, we know that it is also in NP.
Since NP is closed under intersection, we also know that $L_2 \cap L_3$ is in NP.
Finally, since L_1 is reducible to a language in NP in polynomial time, then we have a polynomial-time NTM that decides L_1 – it can go through the reduction to use the polynomial-time NTM for $L_2 \cap L_3$.

Problem 4: **(20 points)** For any k, the language k-COLOR is defined to be the set of (undirected) graphs whose vertices can be colored with at most k distinct colors, in such a way that no two adjacent vertices are colored the same color. In class, we learned that 2-COLOR $\in$ P and 3-COLOR is NP-complete.

1. Prove that 4-COLOR is NP-complete.

Part 1. 4-COLOR is in NP. The coloring is the certificate (i.e., a list of nodes and colors).
The following is a verifier V for 4-COLOR.
V="On input $\langle G, c \rangle$,

1. Check that c includes ≤ 4 colors.
2. Color each node of G as specified by c.
3. For each node, check that it has a unique color from each of its neighbors.
4. If all checks pass, *accept*; otherwise, *reject*."

Part 2. 4-COLOR is NP-hard. We give a polynomial-time reduction from 3-COLOR to 4-COLOR. The reduction maps a graph G into a new graph G' such that $G \in$ 3-COLOR if and only if G' 4-COLOR. We do so by setting G' to G, and then adding a new node y and connecting y to each node in G'.

If G is 3-colorable, then G' can be 4-colored exactly as G with y being the only node colored with the additional color. Similarly, if G' is 4-colorable, then we know that node y must be the only node of its color – this is because it is connected to every other node in G'. Thus, we know that G must be 3-colorable.

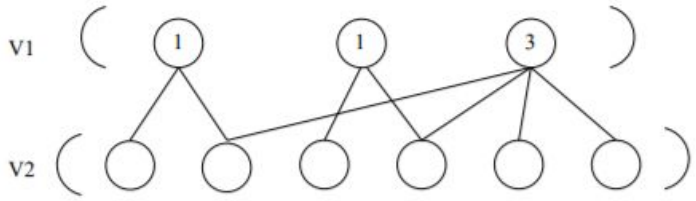
This reduction takes linear time to add a single node and G edges.

Since 4-COLOR is in NP and NP-hard, we know it is NP-complete.

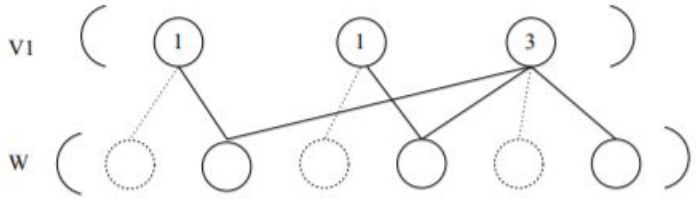
2. Assuming $P \neq NP$, for which values of $k \in \mathbb{N}$ is k-COLOR NP-complete? Briefly explain.

$k \geq 3$. We know that 1-COLOR and 2-COLOR are in P, and that 3-COLOR and 4-COLOR are NP-complete. In fact, we can reduce n-COLOR to $(n+1)$-COLOR, for $n \geq 3$, by the same way as above.

Problem 5: **(20 points)** Let $G = (V_1, V_2, E)$ be a "bipartite" undirected graph, that is, a graph whose nodes are divided into two sets, V_1 and V_2, such that every edge in E connects a node in V_1 to a node in V_2. The nodes in V_1 are all labeled with non-negative integers. For example, G might look like the following:



Some of the nodes in V_2 can be removed to leave a subset $W \subseteq V_2$. A subset W is called *consistent* with G if every node in V_1 which has been assigned a number m is connected to exactly m nodes in W. The problem is to determine whether a consistent W exists. For example, one consistent W for the graph above would be:



We formulate this problem as the language:

$MATCH = \{\langle G, N\rangle |\ G$ is a bipartite graph (V_1, V_2, E), N is an assignment of nonnegative integers to V_1, and there exists $W \subseteq V_2$ that is consistent with G and $N\}$.

(a) Show that $MATCH$ is in NP, using the certificate and verifier method.

The subset $W \subseteq V_1$ is the certificate.

The following is a verifier V for $MATCH$.

V="On input $\langle\langle G, N\rangle, c\rangle$,

1. Test whether c is a set of nodes in V_2.
2. For each node $v \in V_1$, test that it is connected to exactly $N(v)$ nodes in W.
3. If all tests pass, accept; otherwise, reject."

(b) Show that $MATCH$ is NP-hard, using a reduction from 3SAT.

We will show 3SAT $\leq_P MATCH$. The reduction maps a Boolean formula ϕ to a graph G and a numbering scheme N. Our variable gadget will be as follows: for each variable x_i, add one node to V_1 with a label of 1 and add two nodes to V_2 that correspond to x_i and $\overline{x_i}$, then add two edges connecting the 1 node with both the nodes in V_2. Setting a variable x_i to TRUE corresponds to selecting the node x_i to be in W.

Our clause gadget will be as follows: for each clause c_j, add one node to V_1 with a label of 3 and add two "dummy" nodes to V_2, then add edges connecting the 3 node to the two dummy nodes and to the three nodes in V_2 that correspond to the literals in c_j.

We want to argue that this reduction is correct; that is, $\langle\phi\rangle \in$ 3SAT if and only if $\langle G, N\rangle \in MATCH$. First, if $\langle\phi\rangle \in$ 3SAT, then we can always find a consistent W as follows. Place into W each node corresponding to the truth of a satisfying assignment for ϕ (i.e., if x_1 =TRUE, add the node x_i; else add $\overline{x_i}$); this makes W consistent with all the '1' nodes in V_1. Next, for each '3' node, check to see how many neighbors it has already in W (this cannot be more than 3, since each clause can have at most 3 true literals). If a '3' node has less than 3 neighbors in W, supplement this amount to 3 by adding the necessary dummy nodes. This makes W consistent with all the '3' nodes, and thus with all of V_1.

Secondly, we argue that if $\langle G, N\rangle \in MATCH$, then ϕ must be satisfiable. Our variable gadget construction restricts only one of the two $x_i, \overline{x_i}$ nodes to be in W (i.e., TRUE); and our clause gadget requires that at least one node corresponding to a literal in c_j is in W for each clause c_j (i.e., all clauses are TRUE).

So far, we have just argued that there exists a mapping from 3SAT to $MATCH$. Now we need to argue that this mapping can be done in polynomial-time. If ϕ is a Boolean formula with m variables and ℓ clauses. Then, our variable gadget adds $3m$ nodes and $2m$ edges to G, while our clause gadget adds 3ℓ nodes and 5ℓ edges to G. (Note: that the size of N is linear in m.) Thus, the size of G and N, constructed by trivial operations, is $O(m) + O(\ell)$, which is polynomial in $|\phi|$.

Problem 6: **(20 points)** Suppose that we are given a polynomial time algorithm M (formally, a basic deterministic TM) that decides membership in the language VERTEX-COVER = $\{\langle G, k\rangle |\ G$ is an undirected graph and G has a vertex cover of size $\leq k\}$.

1. Describe a polynomial time algorithm that, given the representation of an undirected graph G, finds the size of the smallest vertex cover of G. Your algorithm may use M as a "subroutine". Explain why your algorithm takes polynomial time.

For $i = 1, 2, \ldots, |G|$, run M on $\langle G, i\rangle$. Output the smallest value of i for which M accepts.

Let $p(|G|)$ be M's running time on $\langle G, k\rangle$ (note that $k \leq |G|$). Then our algorithm takes $|G| * p(|G|)$ time, which is also polynomial.

2. Describe a polynomial time algorithm that, given the representation of an undirected graph $G = (V, E)$, finds a smallest-size vertex cover of G (that is, a subset $V' \subseteq V$ such that for each edge $\{u, v\} \in E$ at least one of u and v belongs to V').

Use the algorithm from part (1) to determine the minimum size k.

Start with $H = G$. Then repeat until $k = 0$: Select any edge and for each of the two endpoints, see if removing that vertex and all its incident edges leaves a smaller graph G' that has a $k - 1$ cover. (You can test for this by again running M on $\langle G', k - 1\rangle$.) One of the two tests must succeed. When you find out which, put that vertex into the final cover, and set H to be the remaining graph with that vertex and its incident edges removed. Set $k = k - 1$.

One endpoint of each edge *must* be in the smallest vertex cover. Thus, if $v \in V$ is in the smallest vertex cover of size k (note, there may be many smallest vertex covers, but this algorithm will find one of them), then it must take $k - 1$ nodes to cover all the edges not touched by v. This means that M will accept $\langle G', k - 1\rangle$ where G' is the graph H with v and all its incident edges removed. Once v is found to be in the smallest vertex cover, we are only concerned with the edges of H that v does not touch, so we removed v and its incident edges from the graph, decrement k, and repeat.

If $v \in V$ is not in a smallest vertex cover of size k, then, by definition, this means that all edges of H which are not incident to v cannot be covered with $k - 1$ remaining nodes. Thus, the M will not accept.

Since the algorithm in part 1 and M run in polynomial-time, then this algorithm does so as well, with a multiplicative increase of $O(|G|)$ in the running time.

Minimum Spanning Tree

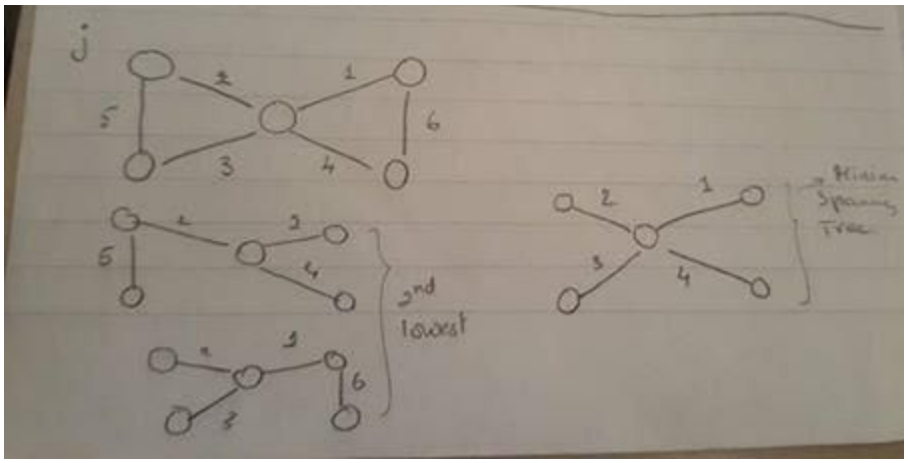
MIT 2015

(g) **T** **F** [4 points] Negating all the edge weights in a weighted undirected graph G and then finding the minimum spanning tree gives us the *maximum*-weight spanning tree of the original graph G.

Solution: True.

(h) **T** **F** [4 points] In a graph with unique edge weights, the spanning tree of second-lowest weight is unique.

Solution: False, can construct counter-example.



Problem 9. Minimum spanning trees [12 points]

Let $G = (V, E)$ be a connected, undirected graph with edge-weight function $w : E \to \mathbb{R}$, and assume all edge weights are distinct. Consider a cycle $\langle v_1, v_2, \ldots, v_k, v_{k+1} \rangle$ in G, where $v_{k+1} = v_1$, and let (v_i, v_{i+1}) be the edge in the cycle with the largest edge weight. Prove that (v_i, v_{i+1}) does *not* belong to the minimum spanning tree T of G.

Solution: Proof by contradiction. Assume for the sake of contradiction that (v_i, v_{i+1}) does belong to the minimum spanning tree T. Removing (v_i, v_{i+1}) from T divides T into two connected components P and Q, where some nodes of the given cycle are in P and some are in Q. For any cycle, at least two edges must cross this cut, and therefore there is some other edge (v_j, v_{j+1}) on the cycle, such that adding this edge connects P and Q again and creates another spanning tree T'. Since the weight of (v_j, v_{j+1}) is less than (v_i, v_{i+1}), the weight of T' is less than T and T cannot be a minimum spanning tree. Contradiction.

4 Algorithm Analysis (15 pts)

4.1. (10 pt.)

Consider the following algorithm for finding a minimum spanning tree in a connected, weighted, undirected graph $G = (V, E)$.

```
def newMST(G):
    while there is a cycle in G:
        let C be any cycle in G
        remove the largest-weight edge from C
    return G
```

That is, while the algorithm can find a cycle in G, it deletes the edge with the largest weight in that cycle. When it can no longer find a cycle, then it returns whatever is left.

4.1.1. **(5 pt.)** Use the following lemma to write a proof by induction that `newMST` is correct: that is, that it always returns an MST of G. **You do not have to prove the lemma (yet).**

> **Lemma:** Let C be any cycle in G, and let $\{u, v\}$ be the edge in that cycle with the largest weight. Then there exists an MST of G that does *not* include edge $\{u, v\}$.

[We are expecting: Your inductive hypothesis, base case, and conclusion, and a description of how the lemma can establish the inductive step.]

Inductive hypothesis: After removing the t^* edge and getting G_t, there is an MST T of G so that $T \subseteq G_t$.

Base case: $G_0 = G$, so the inductive hyp. holds by def. for $t = 0$.

Inductive Step: Suppose the inductive hyp. holds for $t-1$, and let $T \subseteq G_{t-1}$ be an MST of G. Then T is an MST of G_{t-1} as well, since $T \subseteq G_{t-1}$ is a spanning tree of G_{t-1}, and it must be minimal or we'd have a smaller spanning tree for G. By the Lemma, there exists an MST T' of G_{t-1} that does not include the removed edge $\{u, v\}$. Then cost(T') = cost(T) (since both are MSTs of G_{t-1}), hence T' is an MST of G as well. Then T' is an MST of G_t, so that $T' \subseteq G_t$, and this establishes the inductive hypothesis for t.

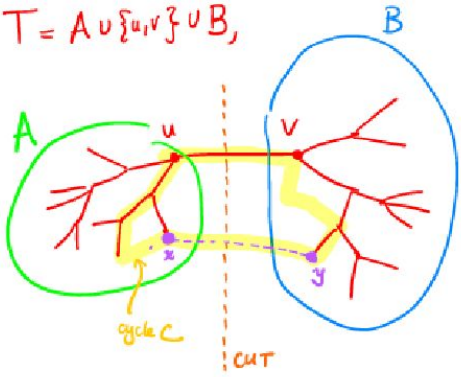
GRADING NOTE:
You needed to use the inductive assumption in your inductive step to get full credit.
(For example, "by the lemma, there exists an MST $\subseteq G_t$" is not correct, because the lemma is about an MST of G_{t-1}, not of G. The inductive hyp. says that it's also an MST of G.)

Conclusion: When the while loop terminates, G_t is a tree, and the inductive hyp. implies that there is an MST T of G s.t. $T \subseteq G_t$. But since both T, G_t are trees (and in particular have $n-1$ edges) this implies that $T = G_t$.

[More space and more problems on next page]

4.1.2. **(5 pt.)** Prove the lemma. (**Hint:** this problem is similar to one from your homework.)

Let C be a cycle in G_1, and let $\{u,v\}$ be the heaviest edge in C.
Let T be any MST of G.
If T does not contain $\{u,v\}$, we are done.
If T does contain $\{u,v\}$, then T can be written as $T = A \cup \{u,v\} \cup B$, where A, B are disjoint trees.



Consider the cut given by $\{A, B\}$.
Since C is a cycle and $\{u,v\}$ crosses the cut, there must be some other edge $\{x,y\} \in C$ that crosses the cut.
$\{x,y\} \notin T$, since T has no cycles.
Consider T' formed from T by swapping $\{u,v\}$ and $\{x,y\}$.
Then T' is still a tree (we did not create any cycles)
T' still spans (we didn't change the set of vertices we touched)
$cost(T') \leq cost(T)$ (since $cost(x,y) \leq cost(u,v)$).
So T' is an MST of G_1 that does not contain $\{u,v\}$.

Question 4: Minimum Spanning Trees (20 points)

Suppose we have weighted graphs $G_1 = (V, E)$ and $G_2 = (V, E)$ over the same set of nodes and edges, but with different weight functions on the edges: $w_1(e)$ is the weight of edge e in G_1 and $w_2(e)$ is the weight of edge e in G_2. Let $G_3 = (V, E)$ be another graph on the same set of nodes and edges, with weight $w_3(e) = w_1(e) + w_2(e)$ for every $e \in E$. In other words, the weight of an edge in G_3 is the sum of the weights of the corresponding edges in G_1 and G_2.

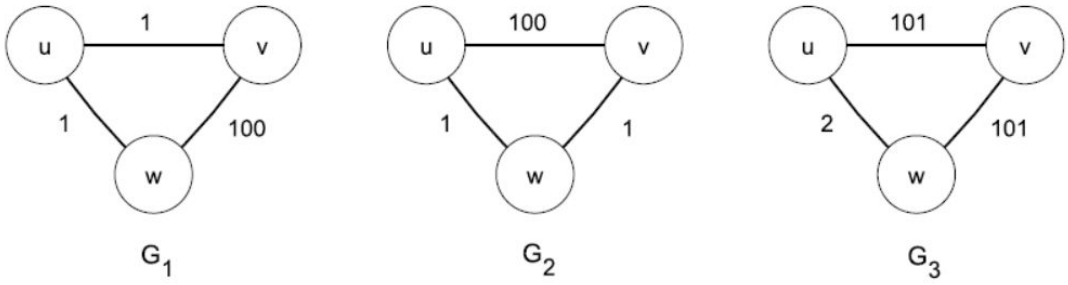
Let T_1, T_2, and T_3 be the minimum spanning trees of G_1, G_2, and G_3, respectively. We define the cost of T_i to be $cost(T_i) = \sum_{e \in E(T_i)} w_i(e)$ where $E(T_i)$ is the set of edges in T_i.

(a) (10 points) Prove that $cost(T_1) + cost(T_2) \leq cost(T_3)$.

(a) $cost(T_3) = \sum_{e \in E(T_3)} w_3(e) = \sum_{e \in E(T_3)} (w_1(e) + w_2(e)) = \sum_{e \in E(T_3)} w_1(e) + \sum_{e \in E(T_3)} w_2(e) = cost(S_1) + cost(S_2)$ for some spanning trees S_1 and S_2 of G_1 and G_2, respectively. Because $cost(T_1) \leq cost(S_1)$ and $cost(T_2) \leq cost(S_2)$, $cost(T_1) + cost(T_2) \leq cost(S_1) + cost(S_2)$. So, $cost(T_1) + cost(T_2) \leq cost(T_3)$.

(b) (10 points) Show an example where the inequality of part (a) is strict. In particular, draw an example of a connected graph on 3 nodes and two weight functions w_1 and w_2, and show that $cost(T_1) + cost(T_2) < cost(T_3)$.

(b) Consider the following examples:



T_1 includes the edges $(u, v), (u, w)$ and costs 2. T_2 includes the edges $(u, w), (w, v)$ and costs 2. T_3 includes the edges $(u, v), (u, w)$ and costs 103. Hence, $cost(T_1) + cost(T_2) < cost(T_3)$.

– The first k edges chosen by Kruskal's algorithm have the following property: there is no cheaper acyclic subgraph of G with k edges. (Assume edge costs are distinct.)

– The first k edges chosen by Prim's algorithm (starting from some "root node" r) have the following property: there is no cheaper connected subgraph of G containing the node r along with k other nodes. (Assume edge costs are distinct.)

Complete the following two questions.

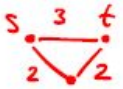
(a) For any pair of distinct vertices $s, t \in V$, there is a unique path from s to t in T.

True **False**

Otherwise there would be a cycle!

(b) For any pair of distinct vertices $s, t \in V$, the cost of a path between s and t in T is minimal among all paths from s to t in G.

True **False**



General Algorithms

Problem 5. Well-Separated Clusters [30 points] (3 parts)

Let $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ be a set of n points in the plane. You have been told that there is some k such that the points can be divided into k clusters with the following properties:

1. Each cluster has radius r.
2. The center of each cluster has distance at least $5r$ from the center of any other cluster.
3. Each cluster contains at least ϵn elements.

Your goal is to use this information to construct an algorithm for computing the number of clusters.

(a) Give an algorithm that takes as input r and ϵ, and outputs the exact value of k.

Solution:

Executive Summary. We find all points sufficiently close to a given point, remove all such points and increment the number of clusters, and repeat until we run out of points. This procedure is exact and takes $O(nk)$ time.

Algorithm Initialize the number of clusters $k \equiv 0$, and the set $S' \equiv S$. Choose a point $p \in S'$, and compute the distance from p to every other point in S'. Remove p and all points within distance $2r$ from S', and increment k. Repeat this procedure until S' is empty; at that point, return k.

Correctness We first note that if the point p chosen in an iteration lies in a cluster with center c, then all points within this cluster will be removed in the same iteration. Indeed, if q is any other point in this cluster, by the triangle inequality,

$$d(p, q) \leq d(p, c) + d(c, q) \leq r + r = 2r.$$

Furthermore, no points from other clusters will be removed in the same iteration. Indeed, if q' is in another cluster with center c', then by the triangle inequality

$$\begin{aligned} d(p, q') &= (d(c, p) + d(p, q') + d(q', c')) - d(c, p) - d(q', c') \\ &\geq d(c, c') - d(c, p) - d(q', c') \\ &\geq 5r - r - r = 3r. \end{aligned}$$

Runtime Analysis Each iteration computes at most n distances, and removes exactly 1 cluster. Hence there are k iterations total, so the overall runtime is $O(nk)$. Since each cluster contains at least ϵn points, there are at most $1/\epsilon$ clusters, so the runtime can also be phrased as $O(n/\epsilon)$.

(b) Given a particular cluster C_i, if you sample t points uniformly at random (with replacement), give an upper bound in terms of ϵ for the probability that none of these points are in C_i.

Solution:

Solution The number of points in C_i is at least ϵn, so the probability that a point sampled u.a.r. lies within C_i is at least $\epsilon n/n = \epsilon$. Therefore the probability that a point sampled u.a.r. *doesn't* lie within C_i is at most $1 - \epsilon$. Since the points are chosen independently, the probability that t points sampled u.a.r. don't lie within C_i is at most $\boxed{(1-\epsilon)^t}$.

Grading comments This problem was relatively straightforward; the full 8 marks were given for the correct answer. An alternative (slightly weaker) bound using Chernoff was also given full credit as long as it was computed correctly. When the answer was incorrect, part marks were allotted for similarity to the solution and evidence of work.

(c) Give a sublinear-time algorithm that takes as input r and ϵ, and outputs a number of clusters k that is correct with probability at least $2/3$.

Solution:

Executive Summary Use the algorithm from part (a) on a subset of the points chosen u.a.r. The size of the subset is chosen so that the algorithm is correct with the desired probability.

Algorithm Initialize S' to be a subset of S of size t (to be determined below), where the points are chosen uniformly at random. Run the same algorithm as in part (a) on this subset.

Correctness From part (b), the probability that t points sampled u.a.r. don't lie in a given cluster is $(1-\epsilon)^t$. Hence, by the union bound, the probability that t points sampled u.a.r. don't lie in *any* cluster is at most $k(1-\epsilon)^t$; since $k \leq 1/\epsilon$, this can be upper bounded by $\frac{1}{\epsilon}(1-\epsilon)^t$. The algorithm is incorrect iff the t sampled points miss any cluster, so we want this probability to be at most one third:

$$\begin{aligned}
\frac{1}{\epsilon}(1-\epsilon)^t &\leq \frac{1}{3} \\
(1-\epsilon)^t &\leq \frac{\epsilon}{3} \\
t\log(1-\epsilon) &\leq \log\frac{\epsilon}{3} \\
t &\geq \frac{\log\frac{\epsilon}{3}}{\log(1-\epsilon)}
\end{aligned}$$

(the sign is reversed in the last inequality since $\log(1-\epsilon)$ is negative). The rest of justification of correctness follows from part (a).

Runtime analysis The runtime of this algorithm is $O(tk)$ or $O(t/\epsilon)$; the analysis is similar to that for part (a). Since neither t nor ϵ depend on n, this algorithm is sublinear.

Problem 6. Looking for a Bacterium [30 points] (2 parts)

Imagine you want to find a bacterium in a one dimensional region divided into n 1-μm regior We want to find in which 1-μm region the bacterium lives. A microscope of resolution r lets y query regions 1 to r, $r+1$ to $2r$, etc. and tells you whether the bacteria is inside that region. Ea ime we operate a microscope to query one region, we need to pay (n/r) dollars for electrici (microscopes with more precise resolution take more electricity). We also need to pay an n-doll fixed cost for each type of microscope we decide to use (independent of r).

(a) Suppose that you decide to purchase $\ell = \lg n$ microscopes, with resolutions $r_1, \ldots, r_\ell$ such that $r_i = 2^{i-1}$. Give an algorithm for using these microscopes to find the bacterium, and analyze the combined cost of the queries and microscopes. **Solution:**

Executive Summary We will use all the microscope to perform a binary search. Starting with the lowest resolution microscope, we use successively more powerful microscopes and eventually using the microscope of resolution 1 to locate exactly where the bacterium is.

Algorithm We will assume n is a power of 2, if not, we can imagine expanding to a larger region that is a power of 2. This will not affect the asymptotic cost. We first use a microscope of resolution $n/2$ to query the region $(1, n/2)$. Regardless of whether the answer is yes or no, we can eliminate a region of size $n/2$. We then do the same with a microscope of resolution $n/4$, and so on.

Correctness This is essentially the same as a binary search, we use each type of microscope exactly once, and eventually we will pin point the location of the bacteria to a region of size 1.

Cost analysis We are only interested in the total electricity cost for this problem. Some students provided runtime analysis, but their correctness were not graded. We use each microscope exactly once, starting from the the one with the lowest resolution, it costs us 2, 4, 8, ... n to use each microscope. The last cost of n comes from using the microscope of resolution 1. This geometric series sums to a total of $2n - 2$, thus the electricity cost is $O(n)$. The total cost of purchasing the microscope is $n \lg n$, for a combined cost of $O(n \lg n)$.

(b) Give a set of microscopes and an algorithm to find the bacterium with total cost $O(n \lg \lg n)$.

Solution:

Executive Summary We borrow our idea from the vEB data structure. We will purchase microscopes of resolutions $n^{1/2}$,$n^{1/4}$,$n^{1/8}$, ..., $n^{1/2^i} = 2$. Clearly, $i = \lg \lg n$. So we spend $n \lg \lg n$ for buying microscopes. We do a linear search at each level, and we will show that the total cost of electricity is also $n \lg \lg n$.

Algorithm We perform a search for the bacterium as follows. First we use the microscope with resolution $\sqrt{n}$ to locate which region of size $\sqrt{n}$ the bacterium is in. This will take at most $\sqrt{n}$ applications of this microscope. We can then narrow our search in the region of size $\sqrt{n}$ and use microscopes of resolution $n^{1/4}$ to continue this process, eventually we can narrow down the location of the bacterium to a region of size 2. From there, it takes one simple application of a microscope of resolution 1 to find the bacterium.

Correctness This collection of microscopes allows us to search every region exhaustsively, thus we can always find the bacterium.

Cost analysis The total purchase of $\lg \lg n$ microscopes will cost us $n \lg \lg n$ dollars. At stage 1, we use the microscope of resolution $\sqrt{n}$ a total of $\sqrt{n}$ times. Each use of this microscope costs $\frac{n}{\sqrt{n}}$ dollars, for a total of n dollars. In fact, at every subsequent step, we always apply the microscope of resolution $\sqrt{r}$ a total of $\sqrt{r}$ times to completely scan a region of size r. The total cost of this is always n. Thus, the total electricity cost associated with using each type of microscope is at most n. Therefore, the total electricity cost os also $n \lg \lg n$. Our combined total cost is thus $O(n \lg \lg n)$.

Problem 2. Algorithms and running times [9 points]

Match each algorithm below with the tightest asymptotic upper bound for its worst-case rur time by inserting one of the letters A, B, . . ., I into the corresponding box. For sorting algorit n is the number of input elements. For matrix algorithms, the input matrix has size $n \times n$. graph algorithms, the number of vertices is n, and the number of edges is $\Theta(n)$.

You need not justify your answers. Some running times may be used multiple times or not a Because points will be deducted for wrong answers, do not guess unless you are reasonably s

☐	Insertion sort	A: $O(\lg n)$
☐	Heapsort	B: $O(n)$
☐	BUILD-HEAP	C: $O(n \lg n)$
☐	Strassen's	D: $O(n^2)$
☐	Bellman-Ford	E: $O(n^2 \lg n)$
☐	Depth-first search	F: $O(n^{2.5})$
☐	Floyd-Warshall	G: $O(n^{\lg 7})$
☐	Johnson's	H: $O(n^3)$
☐	Prim's	I: $O(n^3 \lg n)$

Solution: From top to bottom: D, C, B, G, D, B, H, E, C.

Problem 6. Wiggly arrays [10 points]

An array $A[1..2n+1]$ is ***wiggly*** if $A[1] \leq A[2] \geq A[3] \leq A[4] \geq \ldots \leq A[2n] \geq A[2n+1]$. Given an unsorted array $B[1..2n+1]$ of real numbers, describe an efficient algorithm that outputs a permutation $A[1..2n+1]$ of B such that A is a wiggly array.

Solution: There are several ways to solve this problem in $\Theta(n)$ time. You can find the median k of B using the deterministic $\Theta(n)$ select algorithm and partition B around the median k into two equal sized sets B_{low} and B_{high}. Assign $A[1] \leftarrow k$. Then for each $i > 1$, if i is even, assign an element from B_{high} to $A[i]$, otherwise assign an element from B_{low} to $A[i]$. Since all elements in B_{high} are greater than or equal to all elements in B_{low} and the median is less than or equal to all elements in B_{high}, the array is wiggly. The overall running time is $\Theta(n)$.

10 points for correct $\Theta(n)$ solution and correct analysis. 8-9 points for correct solution but missing a minor step in the analysis.

5 points for correct $\Theta(n \lg n)$ solution and correct analysis. 2-4 points for correct solution but incomplete analysis.

1.5.5. The time it takes to find an ordering $v_1, \ldots, v_n$ of the vertices in a directed acyclic graph $G = (V, E)$, so that for every directed edge $(v_i, v_j) \in E$, $i < j$:

$\Theta(n+m)$

1.5.6. The number of edges in a minimum spanning tree in a connected undirected graph:

$\Theta(n)$ (actually exactly $n-1$)

1.5.7. The worst-case running time of the Bellman-Ford algorithm: $\Theta(n \cdot m)$

1.5.8. The time it takes to determine if an unweighted undirected graph is bipartite: $\Theta(n+m)$

Question 6: Clustering (30 points)

In this problem, we consider a problem about clustering n points on the number line into k clusters.
Input: n distinct points $x_1, \ldots, x_n \in \mathbb{R}$ in sorted order, a parameter $k \leq n$
Task: Divide the n points into k disjoint non-empty clusters $S_1, \ldots, S_k$ such that $\bigcup_{i=1}^{k} S_i = \{x_1, \ldots, x_n\}$ and all points in S_i are to the left of all points in S_{i+1} for $1 \leq i < k$, i.e., for any $y \in S_i, z \in S_{i+1}$, $y < z$.

Objective: Minimize $\sum_{i=1}^{k} \text{cost}(S_i)$

where $\text{cost}(S_i) = (\max(S_i) - \min(S_i))^2$.

Note that $\min(S_i)$ is the minimum element of S_i, i.e., the leftmost point of S_i. Similarly, $\max(S_i)$ is the maximum element of S_i, i.e., the rightmost point of S_i.

For example, if $S_i = \{x_j\}$, then $\text{cost}(S_i) = 0$. If $S_i = \{x_j, x_{j+1}, \ldots, x_{j+t}\}$, $x_j < x_{j+1} < \ldots x_{j+t}$, then $\text{cost}(S_i) = (x_{j+t} - x_j)^2$.

Design an $O(n^2 k)$ time algorithm to find the optimal clustering using dynamic programming. Briefly argue correctness and analyze running time.

Example: Consider clustering 4 points $\{1, 5, 8, 10\}$ into 2 clusters. There are three options:

1. $S_1 = \{1\}, S_2 = \{5, 8, 10\}$, with total cost $0^2 + 5^2 = 25$.
2. $S_1 = \{1, 5\}, S_2 = \{8, 10\}$, with total cost $4^2 + 2^2 = 20$.
3. $S_1 = \{1, 5, 8\}, S_2 = \{10\}$, with total cost $7^2 + 0^2 = 49$.

The output of the algorithm is the optimal solution $S_1 = \{1, 5\}, S_2 = \{8, 10\}$.

For $1 \leq i \leq n$, $1 \leq j \leq \min(k, i)$, let $C(i, j)$ denote the minimum cost for clustering $x_1, \ldots x_i$ into exactly j clusters. In the optimum clustering of $x_1, \ldots x_i$ into j clusters, suppose that the minimum point in the cluster that includes x_i is x_a. We define $s(i, j)$ to be this index a.

For convenience, we define $C(0, 0) = 0$.

We compute the values of $C(i, j)$ for $i \geq 1$, $1 \leq j \leq \min(k, i)$ as follows:

$$C(i, j) = \min_{j \leq a \leq i} \{c(a - 1, j - 1) + (x_i - x_a)^2\}$$

$$s(i, j) = \arg\min_{j \leq a \leq i} \{c(a - 1, j - 1) + (x_i - x_a)^2\}$$

We apply this for i increasing from 1 to n. For each value of i, we compute all entries $C(i, j)$ before proceeding to $i + 1$.

The final value reported is $C(n, k)$, i.e., the cost of clustering all n points into k clusters. The optimum clustering can be reported by backtracking, using the values $s(i, j)$ as described by this pseudocode:

```
OutputClustering(n,k) {
    If k = 0 return;
    OutputClustering(s(n,k)-1,k-1);
    Output "S_k = {", s(n,k), s(n,k)+1, ..., n, "}";
}
```

Running Time: Each entry of the table takes $O(n)$ time to compute. There are $O(nk)$ entries, so overall running time is $O(n^2k)$.

Correctness: We claim that this dynamic program correctly computes the costs $C(i,j)$. We need to show that the recurrence relation $C(i,j)$ is computing the minimum cost of clustering points $x_1,\ldots,x_i$ into $j \leq i$ clusters. If in the optimum clustering, the last cluster j contains the points $x_a,\ldots,x_i$, the cost of the optimal solution is the cost of the optimum clustering of $x_1,\ldots,x_{a-1}$ into $j-1$ clusters, and the cost of the last cluster $(x_i - x_a)^2$. The recurrence considers all possible values of a and picks the one that results in the lowest total cost (every value of a that the recurrence considers corresponds to the cost of a clustering, so the minimum cannot be lower than the optimum clustering). In addition, we compute the values in increasing order of i,j in a way that ensures that we never access the cost $C(a-1,j-1)$ of a subproblem before it has been computed.

Question 5: Matching Points on a Line (30 points)

Input: Two arrays of n points on the number line: red points $r_1, r_2, \ldots, r_n \in \mathbb{R}$ and blue points $b_1, b_2, \ldots, b_n \in \mathbb{R}$. You may assume that all red points are distinct and all blue points are distinct.
Task: Pair up red and blue points, i.e., find a bijection $\pi : \{1,\ldots n\} \to \{1,\ldots n\}$ so that r_i is matched with $b_{\pi(i)}$. Note that each red point is matched with a unique blue point and vice versa.
Objective: Minimize $\sum_{i=1}^{n} |r_i - b_{\pi(i)}|$

Design an $O(n \log n)$ time algorithm that finds the matching (bijection) that minimizes the sum of distances between the matched points. Prove the correctness of the algorithm and analyze its running time. *Suggestion*: Experiment with examples for small values of n. Drawing figures is helpful for understanding the problem.

Example: Consider the input where the red points are $r_1 = 8, r_2 = 1$ and the blue points are $b_1 = 3, b_2 = 9$. There are two possible matchings:

1. Matching r_1 to b_1 and r_2 to b_2. The cost is $|8-3| + |1-9| = 13$.
2. Matching r_1 to b_2 and r_2 to b_1. The cost is $|8-9| + |1-3| = 3$.

The algorithm will return the second matching.

Algorithm: Sort the red and blue points separately.
Let's renumber the points so that $r_1 < r_2 < \ldots r_n$ and $b_1 < b_2 < \ldots b_n$. Now match r_i to b_i.

Running Time: Sorting the two sets of points takes $O(n \log n)$ time. Producing the matching takes $O(n)$ time (actually $O(1)$ time given the sorted orders, since we don't need to do any additional work to produce the matching).

Correctness: We will assume that points are renumbered in sorted order. We will prove that the greedy matching that matches r_i to b_i minimizes the cost of the matching.

Lemma 1: There is an optimal matching that matches r_1 to b_1.
Proof: Without loss of generality, assume that $r_1 \leq b_1$ (if not, swap the red and blue points). Consider an optimal matching M. If r_1 is matched to b_1 in M then we are done. Suppose this is not the case. We will show that we can modify M to obtain a different optimal matching that matches r_1 to b_1. Suppose M matches r_1 to b_j and r_i to b_1. Consider the alternate matching M' where r_1 is matched to b_1, r_i is matched to b_j. All points other than r_1, b_1, r_i, b_j are matched identically in M and M'. We will prove that the cost of M' is at most the cost of M. Hence M' must also be an optimal matching, and r_1 is matched to b_1 as required.

To show that the cost of M' is at most the cost of M, we need to show that
$|r_1 - b_1| + |r_i - b_j| \leq |r_1 - b_j| + |r_i - b_1|$ (*)
We prove this by considering various cases for the configuration of these points on the line. We know that $r_1 < r_i$ and $b_1 < b_j$. Also, by assumption, $r_1 \leq b_1$ Hence $r_1 \leq b_1 < b_j$. There are 3 cases for the possible location of r_i:
Case 1: $r_1 < r_i \leq b_1 < b_j$
Suppose that $r_i - r_1 = x > 0$, $b_1 - r_i = y \geq 0$ and $b_j - b_1 = z > 0$.
Then $|r_1 - b_1| = x + y$, $|r_i - b_j| = y + z$, $|r_1 - b_j| = x + y + z$, $|r_i - b_1| = y$. Hence the inequality (*) holds.

Case 2: $r_1 \leq b_1 < r_i \leq b_j$
Suppose that $b_1 - r_1 = x \geq 0$, $r_i - b_1 = y > 0$ and $b_j - r_i = z \geq 0$.
Then $|r_1 - b_1| = x$, $|r_i - b_j| = z$, $|r_1 - b_j| = x + y + z$, $|r_i - b_1| = y$. Hence the inequality (*) holds.
Case 3: $r_1 \leq b_1 < b_j < r_i$
Suppose that $b_1 - r_1 = x \geq 0$, $b_j - b_1 = y > 0$ and $r_i - b_j = z > 0$.
Then $|r_1 - b_1| = x$, $|r_i - b_j| = z$, $|r_1 - b_j| = x + y$, $|r_i - b_1| = y + z$. Hence the inequality (*) holds.
This concludes the proof of Lemma 1.

Lemma 2: There is an optimal matching that matches r_i to b_i for all $i = 1, \ldots n$.
Proof: We will prove by induction that there is an optimal matching that matches r_i to b_i for all $1 \leq i \leq j$.

Base Case: Follows from Lemma 1.
Inductive Step: By the inductive hypothesis there is an optimal matching M that matches r_i to b_i for all $1 \leq i \leq j$. Note that the cost of the matching is $\sum_{i=1}^{j} |r_i - b_i|$ plus the cost of the matched pairs on the remaining points. If we remove the points $r_1, \ldots r_j$ and $b_1, \ldots b_j$, then M restricted to the remaining points $R_j = \{r_{j+1}, \ldots r_n\}$ and $B_j = \{b_{j+1}, \ldots b_n\}$ does indeed give a valid red-blue matching on R_j and B_j. This R_j-B_j matching must be an optimal matching.

Let M' refer to the partial matching that matches r_i to b_i for all $1 \leq i \leq j$. Any R_j-B_j matching M_j can be combined with M' to produce a valid matching on the entire set of points. The cost this matching is the sum of the costs of M' and M_j. Hence any optimal R_j-B_j matching can be combined with M' to produce an optimal matching on the entire set of points.

By Lemma 1, there is an optimal R_j-B_j matching M_j that matches r_{j+1} to b_{j+1}. Combining this with M' gives an optimal matching on the entire set of points that matches r_i to b_i for all $1 \leq i \leq j+1$.

Note: This proof can shortened in the following way. Let M be an optimal matching such that i is the lowest index such that r_i is not matched to b_i. Then, r_i is matched to some b_j where $j > i$, and b_i is matched to r_k where $k > i$. Using similar arguments to those applied in the proof of Lemma 1, we can show that if we match r_i to b_i and r_k to b_j (and all other points are matched as in M), we still get an optimal matching. We can continue applying this until we get that the matching where r_i is matched b_i for all i is optimal.

2.9. **(5 pt.)** *(May be more difficult)* Given an array A of length n which contains distinct (but arbitrarily large) integers, and an integer $k \leq n$, find the k elements of A that are closest to the median of A, in time $O(n)$.

Here, "closest" means in absolute value: you should return the elements $A[i]$ that give the k smallest values of $|A[i] - m|$, where m is the median of A. You may assume that there are no ties.

1. Use SELECT to find the median m of A in time $O(n)$
2. In time $O(n)$, iterate through the array A to produce the array
$$B = [\ |A[0]-m|,\ |A[1]-m|,\ \ldots,\ |A[n-1]-m|\ ]$$
3. Use SELECT to find j so that $B[j]$ is the k^{th} smallest (time $O(n)$)
4. Iterate through B to find the set of indices i so that $B[i] \leq B[j]$ (time $O(n)$)
5. Return $A[i]$ for each i you found in 4. (time $O(n)$).