Question 1 - True/False (20 pts)

(In-person exams have first 10 problems, Online exams have a randomized set)

1- **[TRUE/FALSE]** A graph G has a unique MST if and only if all of G's edges have different weights.

   a. Consider the graph A-B-C, with edge weights AB = 2, BC = 2

2- **[TRUE/FALSE]** The runtime complexity of merge sort can be improved asymptotically by recursively splitting an array into three parts (rather than into two parts)

   a. nlogn (base 3) and nlogn (base 2) are not asymptotically better than one another

3- **[TRUE/FALSE]** In the interval scheduling problem, if all intervals are of equal size, a greedy algorithm based on the earliest start time will always select the maximum number of compatible intervals.

   a. You can prove it by induction.

4- **[TRUE/FALSE]** Amortized analysis is used to determine the average runtime complexity of an algorithm.

   a. Average runtime complexity of an operation

5- **[TRUE/FALSE]** Function $f(n) = 5n^2 4^n + 6n^4 3^n$ is $O(n^2 4^n)$.

   a. 4^n dominates first term in f(n) and 3^n dominates 2nd term. Therefore, True.

6- **[TRUE/FALSE]** If an operation takes O(1) time in the worst case, it is possible for it to take O(log n) amortized time.

   a. Amortized cost of a set of operations is the average cost of those operations over a worst case sequence of those operations. So for some of the operations the amortized cost could be higher than the worst case cost and for some operations the amortized cost could be lower than the worst case cost of the operation and yet for some operations the amortized cost could be the same as the worst case cost.

7- **[TRUE/FALSE]** If you consider every man and woman to be a node and the matchings between them to be represented by edges, the Gale-Shapley algorithm returns a connected graph

   a. No m-m nodes are connected and no w-w nodes are connected.

8- [TRUE/FALSE] A Fibonacci Heap extract-min operation has a worst case run time of O(log n).

   a. Theta(n)

9- [TRUE/FALSE] Consider a binary heap A whose elements have positive or negative key values. If we randomly square the key values for some elements, the heap property can be restored in A in linear time.

   a. Heapify takes O(n). You just call it on your heap again.

10- [TRUE/FALSE] The order in which nodes in the set V-S are added to the set S can be the same for Prim's and Dijkstra's algorithms when running these two algorithms from the same starting point on a given graph.

   a. Consider a tree.

11- [TRUE/FALSE] In the stable matching problem discussed in class, suppose Luther prefers Ema to others, and Ema prefers Luther to others. Then Ema only has one valid partner.

   a. If they were paired with anyone else, it would cause an instability.

12- [TRUE/FALSE] Algorithm A has a worst case running time of Θ(n^3) and algorithm B has a worst case running time of Θ(n^2 logn). Then Algorithm A can never run faster than algorithm B on the same input set.

   a. Dependent on input.

13- [TRUE/FALSE] Bipartite graphs cannot have any cycles

   a. Consider the graph A-B-C-D-A. This is bipartite and has a cycle.

14- [TRUE/FALSE] In the stable matching problem, it is possible for a woman to end up with her highest ranked man when men are proposing.

   a. Consider question 11 itself.

15- [TRUE/FALSE] If we double the number of nodes in a binomial heap the number of binomial trees in the heap will go up by one.

   a. Consider there is only one node.

16- [TRUE/FALSE] Consider items A, B, and C where items A and B are more similar to each other than items A and C. The K-clustering algorithm (discussed in lecture) involving n items that include A, B, and C could results in a clustering where A and C are in the same cluster, but B is in a separate cluster.

a. There are *n* items and not just 3 items. If you did a 2-clustering, you may end up with A, …, X, …, Z, … C in one cluster and B, …, P, …, Q, …, R in another cluster.

17- [TRUE/FALSE] The first edge added by Kruskal's can be the last edge added by Prim's algorithm.

a. Consider A - B - C with edge weights A-B 1000, B-C 1. Kruskals' first edge will be B-C and Prim's may start at A-B, and have the final edge added as B-C.

Question 2 - Divide and Conquer (15 pts)

Consider a full binary tree representing k generations of the Vjestica family tree (i.e. family tree with k levels). It so happens that the Vjestica family is generally very tall, and they have kept records of all its member's heights which has now sparked a researcher's interest in this data. The researcher wants to find out if there is anyone in this family tree who is taller than its parent (if the parent exists in the tree) AND his/her two children (if the children exist in the tree). In other words, the member we are looking for should be taller than anyone they are immediately related to in the family tree.

a) How many Vjestica's are there in this family tree? (2 pts)

$n = 2^k - 1$ (2pts for this answer)

b) Show that if heights are unique, we can always find such a family member. (5 pts)

Solution 1 (THE SIMPLEST)
The tallest member among all Vjestica must be one as required since he/she is strictly taller than its parent (if the parent exists in the tree) AND his/her two children (if the children exist in the tree) - since the heights are unique.

Solution 2 (Traversal + stopping guarantee)
If the root is taller than the two children, we are done. Otherwise, pick the taller of the two children and see if this child satisfies the criteria.
Do this recursively (which ensures each node thus visited is taller than the parent). If we stop at an internal node, it must satisfy the criteria by having both children smaller, or if we reach a leaf node, then that satisfies the criteria due to having no children (and being taller than the parent as aforementioned).

Solution 3 (Induction)
Base case: property due to having no children or the parents.
Ind Hyp: For some n >= 1, suppose such a member always exists in a tree with n levels.
Ind step: For a tree T with n+1 levels, first check if root is a solution - i.e. if it's taller than both the children. If it's not, there's a taller child, say left child L. Consider the subtree T' rooted at L. By ind hyp, since T' has all distinct heights, it has a member M as desired (relative to T'). If $M \neq L$, then M has both children and parents and it satisfies the property in T as well. if M = L, then L satisfies the property in T' by being taller than the children and having no parent.

Solution 4:
Proof by contradiction. Assume there is no such family member.
So for leaf nodes at level k, they must be smaller than their parents in level k-1. And since nodes in level k-1 are not the members (with the desired property) despite being taller than the children, they must be smaller than their respective parents in level k-2… Continuing up to level

1 the root node must be larger than both the children, ie. root is the member we are looking for, a contradiction.

5pts for any correct proof
~3 pts for correct proof but missing some relatively unimportant steps
0pts for wrong proof.
* Incomplete & Insufficient Proof are not count as correct proof.

c) Assuming that the heights are unique, design a divide and conquer solution to find such a member in no more than O(k) time. (5 pts)

You can assume full binary tree is saved into an array for easy notation.

```
Locally tallest (i) \\Assumes (ensures by definition) that i is taller than its parent IF IT EXISTS

    \\Check to see if we found what we are looking for
    if children exist and person at node i is taller than them, return i

    Else if no children exist, return i

    Else if child at 2i (left child) is taller than child at 2i+1 (right child) then return Locally tallest (2i)

    Else return Locally tallest (2i+1)
```

Notes: If both children are taller, you can choose either child, no need to choose the taller one of the two children. The fastest approach is you compare with the left child first, if the left child is taller, just go with the left child. If not, compare the node with the right child, if the right is smaller, then you return the node. If the right child is taller, you go with the right child.

So each time in the iteration or recursion, If node is not left node, you do:

```
if node.left > node.val:
    func(node.left)
else if node.right > node.val:
    fucn(node.right)
else:
    return node
```

Call Locally tallest (root)

if the algo does not find the answer: 0pt,

if the algo finds the answer but slow and not using D&C on Tree Structure: 1pt,

if the algo finds the answer but slow and using D&Q on Tree Structure: 2pt, if the algo finds the answer and O(k) and not using D&Q on Tree Structure: 2pt,

if using pseudo code have minor errors using O(k) and D&Q on Tree Structure: 2-4 pt,

if all correct: +5pt,

else: return 0pts;

d) Show your complexity analysis using the Master Method. (3 pts)

F(n) = θ (1)
T(n) = T(n/2) + f(n)
n^(log base b of a) = n^(log base 2 of 1) = n^0 = θ (1)
Case 2 - > T(n) = θ (log n) = θ (k)

if C is not D&C: 0pts.

if C) is D&C:

-1pts for wrong f(n)

-1 pts for wrong T(n) (for your answer in part c) or not clearly shown

-1pts for wrong log b base a,

-1pts for wrong case(chose case 2) or give the right comparison (n^{log_b a} = f(n))

-1pts missing master method,

-1pts for wrong or missing answers.

Question 3 - Greedy (15 pts)

There are n distinct jobs, labeled J_1, J_2, ...,J_n, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs.
Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed on PCs at the same time. However, the supercomputer can only work on a single job a time without any interruption. For every job, as soon as the preprocessing is done on the supercomputer, it can be handed off to a PC for finishing.
Let's say that a schedule is an ordering of the jobs for the supercomputer, and the completion time of the schedule is the earliest time at which all jobs have finished processing on the PCs.

a) Design an algorithm that finds a schedule with as small a completion time as possible. Your algorithm should not take more than O(n^2) time in the worst case. (6 pts)

Sort jobs in order of decreasing f_i

b) Prove that your algorithm produces an optimal solution. (7 pts)

Proof is identical to discussion problem with athletes (swimming, then biking+running).

We define an inversion here as a job i with a higher f_i being scheduled after job j with lower f_j.
We can then show that if we take an optimal algorithm which has inversions, then we can remove the said inversions without increasing the overall completion time for all the jobs, until the optimal solution turns into our solution of scheduling jobs in decreasing order of f_i.

We can remove inversions without increasing the completion time of the schedule:

1) If there is an inversion between two non-adjacent jobs i & j, we can always find two adjacent jobs somewhere between i & j such that they have an inversion between them. Therefore, let us assume the case where we have two adjacent jobs which are an inversion. Let job i with higher f_i be scheduled immediately after job j with f_j as its regular PC processing time.
2) We remove the inversion by scheduling job i before job j. Now doing this does not increase the completion time for job i as not only does the supercomputer portion of job i finish faster, but consequently so does its PC processing finish time.
3) Now that job j is scheduled *after* job i, the total time completion for the schedule until job j would change from the initial time of $p_i + p_j + f_i$ to max($p_i + p_j + f_j$, $p_i + f_i$). We know that $f_i \geq f_j$, therefore our total completion time will never increase with the resolution of this inversion.

4) We can then remove every inversion in the optimal algorithm without increasing the total completion time of the schedule.
5) Once all inversions have been removed, it is easy to see that the optimal solution after removing inversions will be the same as our greedy solution of decreasing f_i times

-2 points for not defining what the inversions in this case are.

-2 points for not generalizing the proof

-1 points for not showing that the algorithm is now the same as our greedy algorithm

c) Analyze the complexity of your solution (2 pts)

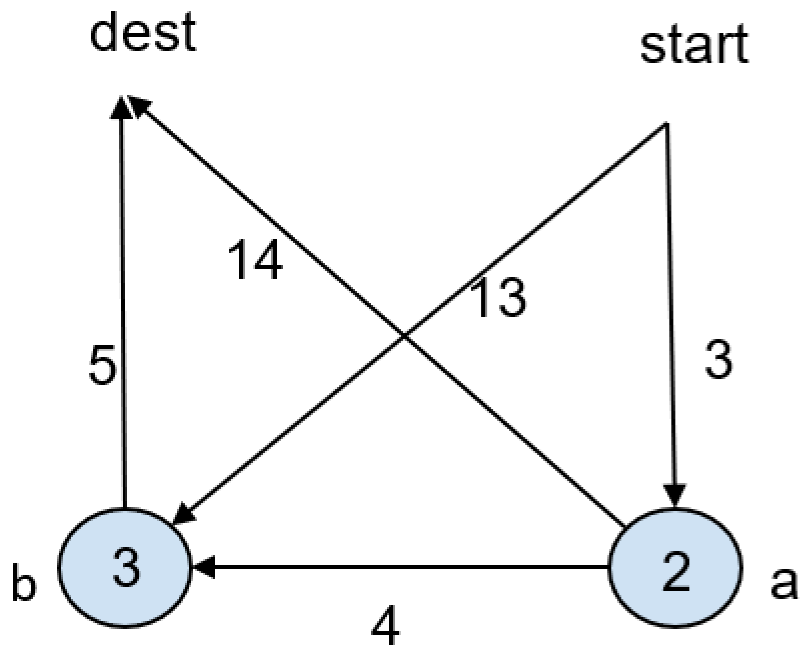
Only requires a Sort which takes θ (nlog n)

- If you analyze your algorithm (which may be incorrect) incorrectly, all points will be deducted. Conversely, if you correctly analyze your incorrect algorithm, you will be awarded full points for this sub-question.

Question 4 - Shortest Path (12 pts)

You are in the city called $start$, and you want to drive your new electric car to the city called $dest$. There are many possible paths to take for this trip, but the good news is that if you take any of these paths, the recharging stations along the way are designed such that if you stop at every recharging station on your way, you will always have enough charge to reach $dest$. You can consider the map as a **directed weighted** graph with recharging stations at some of its nodes. The edge weights C_e represent the time it takes to drive through edge e. You are currently in the city $start$ with a full charge and you want to optimize your trip to get to $dest$ as fast as possible. What you also need to consider is that some charging stations are faster than others. In fact, for each charging station i, we know that we need to wait a fixed amount of time C_i to recharge. Give an efficient algorithm to find out the fastest path from city $start$ to city $dest$ (minimum total time) assuming that you must stop and recharge at every charging station on that path.

Example: Assume charging in cities a and b takes 2 and 3 units of time respectively.



The optimal path is start -> a -> b -> dest. The total time is 17:

3 hrs driving between start to a

2 hours charging at a

4 hours driving from a to b

3 hours charging at b

5 hours driving from b to dest

Solution: Create a new graph G' where we split each node *u* with a charging station into two nodes u_1 to u_2 with an edge going from u_1 to u_2. connect all incoming edges into u to u_1. All outgoing edges from u will go out of u_2. Assign edge costs to new edges equal to the charging time at that charging station. Run Dijkstra's to find shortest path.

graph modification (8 pt):

- solution I: split charging station node into two nodes, one receiving incoming edges and one sending outgoing edges (*as described above*)
- solution II: adding the charging node weight (charging time) to the cost (traveling time) of all coming or all outgoing edges weights
- solution III: run Dijkstras and add each node value into account during each step

no modification to graph (-8 points)

modification not specified (-6 points)

completely incorrect modification (-6 points)

other errors such as not accounting for edge weights and incomplete modification (-4 points)

add node values to both incoming and outgoing edges or not clear what is added to what(-3 points)

path finding (4 pt):

- Use some shortest path finding algorithm on weighted graphs (like Dijkstras)

modified version of Dijkstras has errors (-2 points)

incorrect path finding or exponential time (-4 points)

Question 5 - Heaps (12 pts)

The United States Commission of Southern California Universities (USCSCU) is researching the impact of class rank on student performance. For this research, they want to find a list of students ordered by GPA containing every student in California. However, each school only has an ordered list of its own students by GPA and the commission needs an algorithm to combine all the lists. There are a few hundred colleges of interest, but each college can have thousands of students, and the USCSCU is terribly underfunded so the only computer they have on hand is an old vacuum tube computer that can do about a thousand operations per second. They are also on a deadline to produce a report so every second counts. Find the fastest algorithm for yielding the combined list and give its runtime in terms of the total number of students (m) and the number of colleges (n).

```
Use a minheap H of size n.
Insert the first elements of each sorted array into the heap. The objects entered into the heap will consist of the pair (GPA, college ID) with GPA as the key value.
Set pointers into all n arrays to the second element of the array CP(j) = 2 for j=1 to n
Loop over all students (i= 1 to m)
        S = Extract_min(H)
        CombinedSort (i) = S.GPA
        j = S. college_ID
        Insert element at CP(j) from college j into the heap
        Increment CP(j)
endloop
```

Build min heap of size n (4 points)
Extract the min element (2 points)
Keep pointer to insert the next element in array for extracted element. (2 points)
Recursively/ In Loop do it for all the students (2 points)
Runtime complexity - O(mlog n) (2 points)

SOLUTION 2: Divide & Conquer works too:
https://leetcode.com/problems/merge-k-sorted-lists/solution/#approach-5-merge-with-divide-and-conquer