2.35

CS161 Final Exam, Summer 2022

You have **180 minutes** for the exam. You may use up to 10 pages (each front and back) of notes that you may have prepared; you may not use other materials or confer with anyone. The last two pages (i.e. the last printed page front/back) contain information that you may find convenient, although it is not an exhaustive resource. You may detach it before the exam begins, and you do not need to hand it in.

Unless explicitly otherwise stated in a question, assume that:

- Big-O and big-Ω are meant in their informal "tight" sense. E.g., if we ask for the big-O or big-Ω running time of MergeSort, it would **not** be acceptable to say $O(n^2)$ or $\Omega(1)$.
- Operations such as addition, multiplication, array lookups – the ones that we generally consider to take $O(1)$ time in our course – also take $O(1)$ time here.
- Except in Karger's algorithm, multiedges (and self-loops) don't exist in graphs.
- You may use algorithms from class (where "class" is broadly construed to include the pre-homeworks, homeworks, and exams) without re-implementing them / writing them out. You may cite results from class without proving them.

Because some people are taking the exam remotely, **we will not provide clarifications during the exam itself.** There are no intentional ambiguities, but if you believe you have encountered an ambiguity, please briefly note any assumptions you are making and move on.

Some sections of the exam will tell you that no justification is required/considered. In other cases, you might not need to provide justification or show your work, but can do so if you'd like in case it might earn you partial credit if your answer turns out to be wrong. If we require justification, we will say that too.

We don't anticipate the same time pressure of the midterm, but we still recommend that you not get too hung up on any one problem, especially not before you've looked at everything. Earlier questions aren't necessarily easier, but we recommend doing Questions 6 and 8 later.

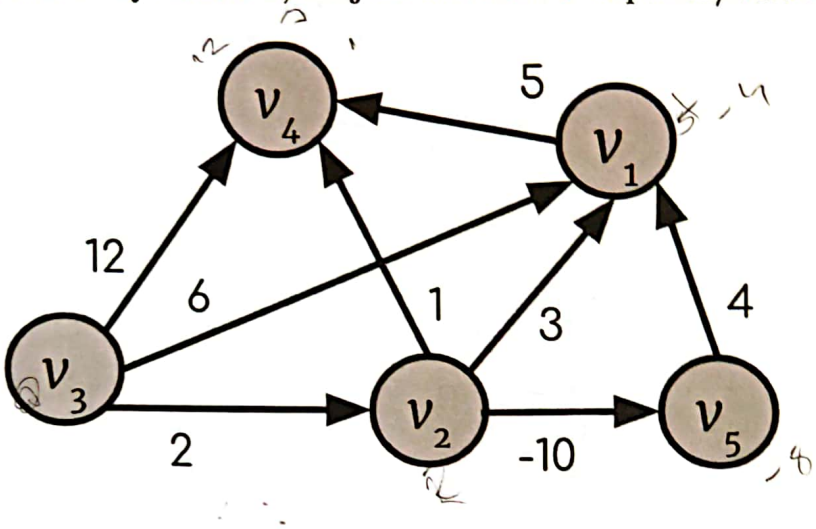
Good luck! The Algorators are rooting for you to do your best!

Name: Melis Oktayoglu SUID (e.g., itullis): meleyto

1	2	3	4	5	6	7	8	Total
45	*30*	*30*	*25*	*15*	*10*	*15*	*10*	**180**

Question 1: Graph Computations (45 pts.)

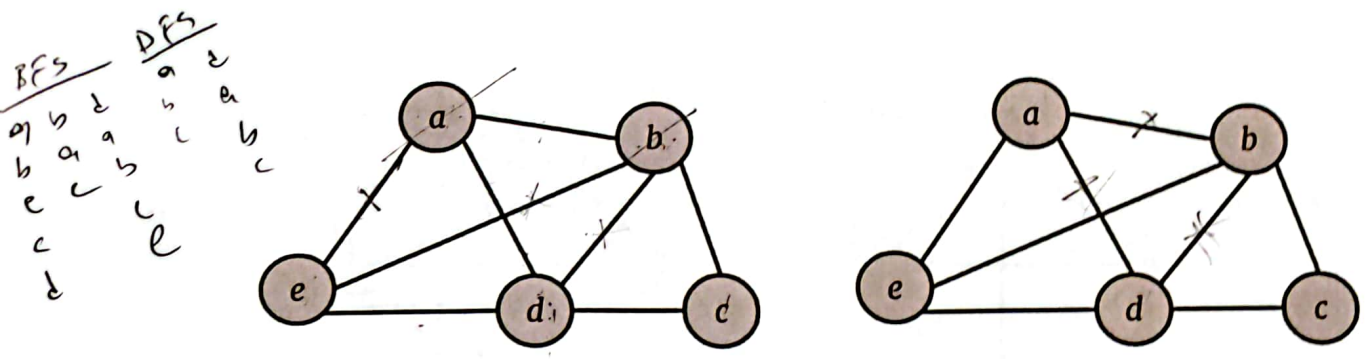
****** For all parts of Question 1, no justification is required/considered. ******



(a) (18 pts.) Let d_{34} denote the shortest distance from vertex v_3 to vertex v_4. Consider the following table of *estimates* of d_{34}. **Fill in the remaining six cells.**

- For positive k, after k rounds of...
  - ...Dijkstra's, k nodes in total have been marked "sure".
  - ...Bellman-Ford, each edge has been processed/relaxed exactly k times.
  - ...Floyd-Warshall, the algorithm has considered paths using nodes v_1 through v_k, inclusive, as intermediates.
- Dijkstra's and Bellman-Ford are called with v_3 as the source.
- Calling Dijkstra's on a graph with negative edge weight(s) is a bad idea in general, but here we're doing it anyway. (The algorithm will still produce estimates.)
- Assume here that Bellman-Ford uses the same edge order every round: it processes edges in order of *increasing edge weight*: -10, 1, 2, 3, 4, 5, 6, 12.

	Initially	After one round	After two rounds
Dijkstra's Algorithm	∞	12	3
Bellman-Ford	∞	12	1
Floyd-Warshall	12	11	11



The graph on the right is just another copy, if you want it.

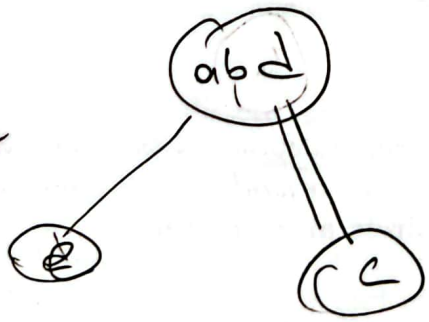
(b) (i) (4 pts.) Suppose we run two separate searches, one breadth-first and one depth-first. *In both searches, alphabetically earlier vertices win all tiebreaks.* **At which vertex** should we start if we want the BFS and the DFS to discover the same vertices in the same order? **(Circle exactly one.)**

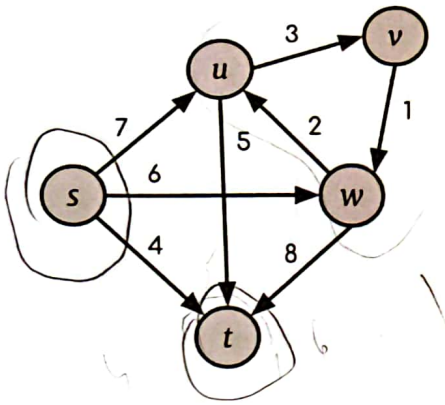
a b c d e

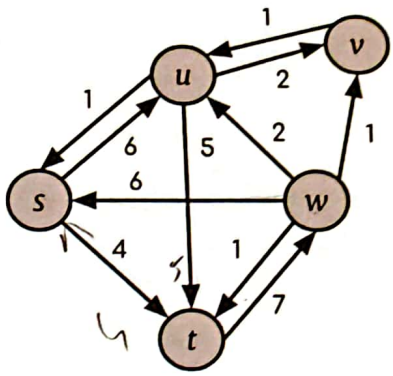
(ii) (6 pts.) **Show any intermediate state** that Karger's Algorithm could reach when run on this graph, such that

- there are exactly three vertices left, and
- the probability of success (finding a min cut) from this point on equals 2/3.

Be sure to label the vertices in your drawing. For vertices that are the result of merging, label them with all the letters of the merged vertices (in ABC order). No explanation required/considered, **just the labeled drawing.**







(c) (i) (4 pts.) The graph on the right is the residual graph resulting from taking the graph on the left and processing ~~two~~ Ford-Fulkerson augmenting paths in some order. **List both of those augmenting paths**, in either order.

swt

suvwt

(ii) (3 pts.) What is the **maximum flow** from s to t in the graph on the *left*?

16

(iii) (4 pts.) *(The edge weights are not relevant for this and the next subpart.)* **How many strongly connected components (SCCs)** are there in the graph on the *left*?

3

(iv) (6 pts.) Suppose we run a depth-first search on the graph on the *left*. Which of these will *always* be true, *regardless of the starting vertex* and of the tiebreaker rules for vertices? **Circle all that apply.**

- s will have the largest finishing time.
- t will have the smallest finishing time.
- u will have a larger finishing time than w. X

Question 2: True or False (30 pts.)

Determine whether each of the following statements is true or false and **circle the correct answer.** False statements are false for a major reason, not a minor technicality. No justifications required / considered.

- (T) F $\log_{10} n$ is $\Theta(\log_2 n)$.
- T (F) If $f(n) = O(g(n))$, then $g(n)$ cannot be $O(f(n))$.
- T (F) In each level of its recursion tree, MergeSort performs $\Theta(\log n)$ work in total.
- T (F) Once RadixSort puts two elements in the same bucket in some round, those two elements must always share the same bucket in any subsequent rounds.
- (T) F When QuickSort is run on a list of 10 distinct elements, the probability that the smallest and largest elements are never compared at any point is exactly $\frac{4}{5}$.
- T (F) If $\mathcal{H}$ is a universal hash family hashing to n buckets (with $n > 1$), then for any uniformly randomly selected hash function h in the family, any element u in the universe, and any two different buckets b_j and b_k, it must be true that $\Pr(h(u) = b_j) = \Pr(h(u) = b_k)$. (Here the probability is over the choice of h.)
- T (F) A Bloom filter might declare that it has not seen an item even if it actually has seen that item before.
- (T) F When run on a best-case n-element list, heapsort takes $\Theta(n)$ time.
- (T) F An inorder traversal of a binary search tree with n elements necessarily takes $\Omega(n \log n)$ time, because it produces a sorted list and BSTs are comparison-based.
- (T) F The purpose of using a Fibonacci heap when implementing Dijkstra's Algorithm is to allow estimates within the heap to be decreased efficiently.
- T (F) All directed graphs have at least one topological ordering on the edges, such that there are no contradictions.
- T (F) If each of the n vertices in a directed graph G has at most 3 edges (incoming and/or outgoing), then running Bellman-Ford once per vertex (i.e. with each vertex as the source exactly once) takes $O(n^3)$ time.
- T (F) In the unbounded knapsack problem, if some item a weighs less than some item b and is worth more than b, b will never be part of any optimal solution.
- T (F) Suppose e is a largest-weight edge in a graph, and all other edges have lower weights. Then no minimum spanning tree found by Prim's Algorithm can include e.
- (T) F Suppose a list is known to be sorted and to contain only 1s, 2s, and 3s. Then the sum of all of its elements can be found in $O(\log n)$ time.

Question 3: Median Maintenance (30 pts.)

(a) (12 pts.) Suppose we want to design a data structure that operates on 64-bit integers and supports the following operations. (Note: Deletion does not need to be supported.)

- `Insert`(x): Add an integer x to the structure, in $O(\log n)$ time.
- `FindMedian`(): Find the median of the n integers in the structure, in $O(1)$ time. (For simplicity, assume this is called *only when n is odd.*)

Consider each of the following potential solutions; assume they are implemented as efficiently as possible given the descriptions. For each one, please **circle T or F.** No explanations required/considered.

(i) Use a red-black tree. For `Insert`(x), use the red-black tree's own Insert operation. For `FindMedian`(), return the root of the tree.

- T (F) It is possible for a call to `Insert`(x) to exceed the time bound.
- (T) F It is possible for a call to `FindMedian`() to return a wrong answer.

In parts (ii) and (iii), assume that the array will never need to be resized (or that the resizing essentially takes constant time.)

(ii) Use an array in which the contents are always kept sorted. For `Insert`(x), use binary search to find an appropriate place in the array (that maintains the sorted order) and insert the new element in place, shifting values to the right as needed. For `FindMedian`(), return the central element of the array.

- (T) F It's possible for a call to `Insert`(x) to exceed the time bound.
- T (F) It is possible for a call to `FindMedian`() to return a wrong answer.

(iii) Use a min heap implemented as an array-based balanced binary tree, as in class. For `Insert`(x), use the min heap's own Insert operation. For `FindMedian`(), return the central element of the array.

- T (F) It is possible for a call to `Insert`(x) to exceed the time bound.
- (T) F It is possible for a call to `FindMedian`() to return a wrong answer.

(b) (18 pts.) Design a fundamentally *heap-based* data structure to solve this problem within the given time bounds. (Your data structure may also include a constant number of variables, but do not use other data structures besides heaps.) You may assume for convenience that *all items inserted into the structure are unique.* You may also choose whether a heap is a min or a max heap.

We are expecting: a brief description of the pieces of your structure and how it operates (specifically what happens on an `Insert`(x) call and what happens on a `FindMedian`() call). No justifications of correctness / running time are required / considered, but your description should be detailed enough for us to be able to assess these things.

We can use a binary min heap for this. We can exploit the fact that heaps are comparison based.

First when inserting we would also keep track of how many levels it skipped while comparing and migrating it to the correct topological place. Since there are $\log n$ levels and it will be compared once in each level it will cost us $O(\log n)$

Then for finding the median we would always keep a pointer to the node with comparison number $\log n/2$

Question 4: Weightiest Directed Path (25 pts.)

Suppose we have a directed weighted graph G, with vertices numbered 0 through $n-1$, and m edges (assume $m \geq 1$), each with a *positive* edge weight. G has the special property that *all edges go from lower-numbered vertices to higher-numbered vertices.*

Say we want an algorithm `WeightiestDirectedPathWeight`(G) that returns the largest possible *total combined weight* of the edges of a path *anywhere* in G. Paths cannot repeat vertices.

(a) (5 pts.) **Either:**

- briefly explain how to use the Floyd-Warshall algorithm (perhaps with a small modification) to solve this problem and get the final answer, and briefly justify why it is correct, **or**
- briefly explain why this cannot be done for G.

In Floyd-Warshall we want to solve the shortest path problem. But here we look for the largest path. To modify Floyd-Warshall for this we can make the initial estimates as $-\infty$ and whenever an intermediate estimate is larger than the current estimate we'll update the estimate.
Then we'll return the largest entry of the array
And because of G's special property there can't be any cycles leading us to infinity.

(b) (20 pts.) You are given graph G as an adjacency list L containing the neighbors of each vertex (in arbitrary order), *and* as an adjacency matrix M containing the edge weights (or a 0 when there is no edge). (The purpose of the adjacency matrix is to allow you to look up edge weights easily and quickly.)

Write an algorithm `WeightiestDirectedPath`(n, L, M) that has the same specs as `WeightiestDirectedPathWeight`, but returns a weightiest path (a path with the largest possible total combined weight) itself – i.e., a list of the vertices of any such path, in order; for (a lot of) partial credit, you may return just the total weight of such a path instead. Only consider pairs of vertices where you can actually reach the second vertex from the first. *Your solution must run in $O(m+n)$ time for full credit.*

Be precise; this problem is much better suited to pseudocode (or code) than description.

This page is for your algorithm for Question(4)(b).

This will be a modification of Dijkstra's. but using DFS. Also, let's make nodes hold parent info as well

```
WeightiestDirectedPathWeight(n, L, M):
    dist = [0] * n   # hold distances to the vertices
    visited = [false] * n
    parents = [0 ---- n-1]
    current = 0
    queue = L[current]
    While any visited is false:

        for neighbor in L[current]:                                   (*)
            dist[neighbor] = max(dist[neighbor], M[current][neighbor] + dist[current]
            If (*) was bigger:
                parent[neighbor] = current

        visited[current] = True
        current = queue.pop()
        queue.add(L[current])

    end while
    path = []
    max_end = dist.indexOf(max(list))
    i = max_end
    while parents[i] != i:
        path.addLeft(parents[i])
        i = parents[i]

    return path
```

Again because of the special property, always lower numbered vertices will serve as source so we don't have to worry about source picking as the paths

Question 5: Mysterious Recurrence (15 pts.)

Consider the recurrence

$$T(n) = \begin{cases} 0 & n = 0 \\ T(n/2) & n \text{ is positive and even} \\ T(n-1)+2 & n \text{ is positive and odd} \end{cases}$$

For example, the values of $T(n)$ for $n = 0, 1, 2, 3, 4, 5, 6, 7$ are $0, 2, 2, 4, 2, 4, 4, 6$.

Let's prove, via induction, that $T(n) = O(\log n)$. We will provide scaffolding and ask you to fill in the details. *Put on your Terry hats; unjustified jumps in logic may result in deductions.* If you would like, you may use (without citing it) the fact that $\log_2(x)$ is monotonically increasing.

Claim / Inductive Hypothesis:

Let $c =$ _____, $n_0 =$ 5 . For all integers $n \geq n_0$, $T(n) \leq c \log_2 n$.

(You may fill in any values above that are consistent with the rest of your proof.)

Base case: (3 pts.) *(Write your base case below!)*

Inductive step:

Suppose that the claim holds for all integers n such that $n_0 \leq n < k-1$, for some arbitrary integer k. Then we will show that it holds for $n = k$.

We will break this into two cases according to whether k is even or odd.

(Nothing to write here; go on to the next page.)

(5 pts.) First suppose that k is even. Then... *(Show that the claim is true in this case.)*

we know that it holds for $k-2$

$$T(k-2) = T\left(\frac{k-2}{2}\right) \leq c\log_2(k-2) \quad \frac{T(\frac{k-2}{2})}{\log(k-2)} \leq c$$

we wanna check,

$$T(k) = T\left(\frac{k}{2}\right) \leq c\log_2 k \quad \Rightarrow \quad \frac{T(k/2)}{\log_2 k} \overset{?}{\leq} c$$

we know that

$$T(k-2) \leq T\left(\frac{k}{2}\right) \quad \text{and} \quad \log_2(k-2) < \log_2 k$$

$$\frac{T(\frac{k-2}{2})}{T(k/2)} \leq 1 \quad \text{and} \quad \frac{\log(k-2)}{\log_2 k} < 1$$

(7 pts.) On the other hand, suppose that k is odd. Then... *(Show that the claim is true in this case. Note: if you end up stuck with a term that seems hard to work with, it might help to consider whether you know if it is odd or even...)*

we know that it holds for $k-2$,

$$T(k-2) = T(k-3) + 2 \leq c\log_2 n$$

Conclusion: We see that $T(n) \leq c\log n$ holds for all integers $n \geq n_0$. Accordingly, $T(n) = O(\log n)$. *(Nothing to write here! The proof is complete!)*

Question 6: Messing With Medians (10 pts.)

*****In Question 6, explanations aren't needed but will be considered. This one may take some time to understand; you may wish to do other questions first. It is intentionally worth fewer points relative to its difficulty.*****

Waverly finds it strange that even when we use k-Select to find some element other than the median, we still use the median to partition, regardless of the value of k. She proposes `30thPercentileSelect`(L), a variant of k-Select that finds the 30th percentile of the list L:

Input: A list of n distinct values.

Output: The 30th percentile of the list. (That is, this algorithm selects the smallest value that is strictly greater than 30 percent of the data. For example, the 30th percentile of the integers from 1 to 25, inclusive, is 8.)

Algorithm `30thPercentileSelect`(L):

- Divide the list into $n/5$ groups of 5: the first five (in the given order), then the next five, etc. (If n is not evenly divisible by 5, just pad the last group with ∞s.) For each of these groups, find the *second-smallest* value. (The intention is that this is a good estimate of the 30th percentile of the group.)
- Call k-Select (*not* `30thPercentileSelect`) on a list of these second-smallest values to find the *median.* Call this estimate est_{30} the "median of 30th percentiles".
- Partition L around est_{30}. If est_{30} really is the actual 30th percentile, return it. Otherwise, recursively call `30thPercentileSelect` on the appropriate part of the list, according to whether the estimate is too low or too high.

(a) (2 pts.) To warm up and make sure you understand the algorithm, imagine running `30thPercentileSelect` on the list $[1, 2, 3, ..., 25]$ – that is, the first 25 positive integers, in order. (You do not need to imagine working through k-Select – just assume that it correctly finds the median.)

What does est_{30} end up being? (If you get 7, look again at the algorithm.) **And what is the list that we pass into that first recursive call to `30thPercentileSelect`?**

est_{30}: _____

The list for the recursive call: _____

(b) (3 pts.) Now let's think *in general*, rather than about that particular example, though *assume here (for convenience) that n is a multiple of 5 and* $\frac{n}{5}$ *is odd.* In terms of n, what is the largest number of values in L that could be *less* than est_{30}? **Give an exact answer.** (For the original k-Select algorithm, this was $\frac{3n}{10} + \frac{3}{2}$.)

(c) (2 pts.) To spare you from doing a similar calculation, we can find that the largest number of values in L that could be *greater* than est_{30} is $\frac{2n}{5} + 1$.

In light of that and your result from (i), **what is a tight upper bound** on the size (in terms of n) of the input to the recursive call to `30thPercentileSelect`? Your answer should end up being a constant times n; don't worry about a possible, e.g., ± 1 here. (For the original k-Select algorithm, this term was $\frac{7n}{10}$.)

(d) (3 pts.) Using your result from (iii) and the known fact that k-Select is $O(n)$:

- Write a **recurrence** for the running time of `30thPercentileSelect`. It should include a *single* tight big-O term covering all the non-recursive work, including calls to other functions. (Here you do not need to worry about divisibility issues or give base cases; assume base cases take constant time.)
- Use this to determine a **tight big-O running time bound.**

(If you find yourself coming up with a complicated recurrence or big-O analysis method, you should reconsider! If you are not sure about your answer to (iii), you may proceed as if the answer were kn for some unknown $k < 1$.)

Recurrence: $T(n) =$ _____ **Running time:** $O($_____$)$

Question 7: Activity Selection (15 pts.; (a)/(b) are independent)

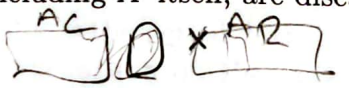
Recall the "activity selection" problem from lecture: we given a list of activities (s_i, f_i), each defined by a starting time s_i and an finishing time f_i, with $s_i \leq f_i$. Our goal is to select a set of activities S such that:

- no two activities in S overlap (even at a single point in time), and
- S contains as many activities as possible.

In this problem, we only care about returning the maximum possible size of such a set of activities, not the activities themselves.

(a) Consider the following randomized divide-and-conquer solution. Let $T(A)$ be the maximum total score we can get using only activities in the list A. If A is empty, we return 0. Otherwise we calculate $T(A)$ as follows:

- Choose an activity $A' = (s', f')$ in A uniformly at random from all activities in A.
- Go through the *other* activities in A and divide them into two sublists A_L and A_R, such that A_L contains all activities in L with a *finishing* time **strictly** earlier than s', and A_R contains all activities in R with a *starting* time **strictly** later than f'. (Any other activities, including A' itself, are discarded.)
- Return $T(A_L) + 1 + T(A_R)$.



To solve the overall problem, we compute $T(A)$ for the full list of activities.

(i) (2 pts.) Any individual run of this algorithm might return a wrong answer. Suppose we run this algorithm a very large number of times on some list. **In one sentence (or even a few words), explain** what we should do with the results to get a final answer. (Note: the final answer might itself still be incorrect. Don't worry about the probability of correctness. This is not necessarily a *good* algorithm.)

We will be left with a list of possible max scores. We should get max of these.

(ii) (3 pts.) Let's think about the expected running time of this algorithm. Recall that the expectation of a randomized algorithm is taken over the randomness of the algorithm's choices, not over the input itself. We choose an input in the most adversarial way possible, to try to make it take as long as possible to run; we know exactly how the algorithm works, but we do not get to preview or control the exact random choices the algorithm will make.

Assume that choosing an activity uniformly at random takes $O(1)$ time, and determining whether **one** activity belongs in A_L, A_R, or neither takes $O(1)$ time.

Your task here is to **construct/describe a worst-case input** for this problem for arbitrary (very large) n. You should be able to describe it pretty concisely.

The worst case input would be the selecting the activities with smallest start time so A_R would be in a recursion such as - $T(A_R) = T(n-1)$ So recursion will very slowly decrease subproblem size but left side will end abruptly.

(iii) (4 pts.) With a worst-case input, what are the **expected and worst-case running times** of a single run of this algorithm, as tight big-O expressions in terms of n? No explanation required/considered. (Hint: We are not expecting you to have to do any calculations here.)

- **Expected:** $O(n \log n)$
- **Worst-case:** $O(n^2)$

(b) (6 pts.) Waverly finds a (sadly, incorrect) solution to the activity selection problem:

Repeat the following until the list of activities A is empty:

- For each activity, determine how many other activities in A it overlaps with.
- Select an activity with the fewest overlaps. Remove that activity (and all activities that overlap it) from A.

Waverly offers this proof of correctness. **Underline the EARLIEST assertion that is incorrect.** (The rest of the proof is structured correctly; there is one significant huge mistake, and it is not a technicality, a lack of rigor, etc. Do *not* try to come up with a counterexample; we believe they are all too large to find during an exam.)

Let S^* be an *ordered* set of activity selections that is optimally large. If S^* is the same set of selections that our algorithm would pick, we are done. Otherwise, there is a first time at which S^* selects an activity x that does *not* have the fewest overlaps. Then at that time, there necessarily existed some *other* activity y in A that had the fewest conflicts.

Now, we must be in one of exactly two cases: S^* selected y later on, or S^* never selected y. In the former case, we simply swap x and y in S^*. This does not change the optimality of S^*, but now we have removed an activity with the fewest overlaps.

In the latter case, notice that because y has strictly fewer conflicts than x, it must be strictly within x. (This also explains why S^* didn't select both x and y.) Then selecting y is no worse, since any activity that was not removed by selecting x is not removed by selecting y. (It is also no better: selecting y may result in some additional activities remaining around that wouldn't have stuck around after removing x, but these can't allow us to do better overall, or else S^* would not be optimal.)

Repeating this process, inductively, we can transform S^* into a solution that always selects an activity within the fewest conflicts. Therefore a solution that always selects an activity with the fewest conflicts is also optimal.

What's wrong with your underlined part, in one sentence:

we cannot be in either of these cases because if y is an activity with fewer conflicts it had to be chosen before x and as algorithm keeps on picking activity with fewest overlaps it cannot not chose it ever.

Question 8: De-cliquing (10 pts.)

*****We strongly recommend making this one of the last questions you attempt.*****

Indy's social media company is trying to forge connections between different groups of people. He gives you a directed, unweighted, and weakly connected[1] graph G with n vertices and m edges. *Moreover, it is guaranteed* that G's metagraph of SCCs (i.e. replacing each SCC with one vertex) would be a tree if the edges were made undirected and any multiedges were replaced with single edges.[2]

Write an algorithm that determines where to add *one new directed edge* to the original G in a way that makes the number of strongly connected components decrease by as much as possible (given that you are adding only one edge), and returns where to put the edge.

For full credit, your algorithm must run in $O(m+n)$ time. Partial credit will be awarded for ideas and progress. Justifications of correctness or running time are not required/considered, but your algorithm must be clear enough for us to know if it is correct/fast enough.[3] We think you can solve this one relatively briefly and descriptively, but you can use pseudocode if you want.

After running Kosaraju's which runs in O(m+n) we will add a new edge to the vertices in any node in metagraph node from the metanode without any outgoing edges to the one with no incoming edges.

[1] Recall that this means that if all the edges were made undirected, the graph would be connected.

[2] It is subtle why this extra guarantee is important, and you don't really need to worry/think about it.

[3] You can invoke algorithms from class, homeworks, exams, etc. without writing them out, and it's fine to say, e.g., "then we take X information from Y algorithm". As an example (which is not relevant to this problem), finding the median using k-Select incidentally gives us partition information "for free".