Question 1

Part a)

Consider the following algorithm DecideCandidate(L) where L is a list of the n names that we are considering.

Note: The following algorithm is written in pseudocode with some Python-like syntax.

DecideCandidate(L)

```
1:  n ← length(L)                                   ▷ Note, L is a list of names
2:  if n ≤ 1 then
3:      return L
4:  end if
5:
6:  mid ← ⌊n/2⌋
7:  left ← DecideCandidate(L[0, ..., mid])          ▷ Note, indices in lists are inclusive, 0-based
8:  right ← DecideCandidate(L[mid + 1, ..., n − 1])  ▷ List slicing takes constant time
9:
10: potential ← left
11: for name in right do
12:     if name not in potential then potential.append(name)
13: end for
14:
15: candidates ← [ ]                                ▷ Initialize empty list
16: for name1 in potential do
17:     count ← 0
18:     for name2 in L do
19:         if name1 = name2 then count ← count + 1
20:     end for
21:     if count > ⌊n/3⌋ then candidates.append(name1)
22: end for
23:
24: return candidates
```

Part b)

Let $T(n)$ be the worst-case running time of DecideCandidate(L) where L is a list of the n names.

WLOG, assume n is a power of 2.

Steps 1-4 is our base case takes $\mathcal{O}(1)$ time if $n = len(L) \leq 1$.

If $n > 1$, then we make two recursive calls with runtime $T(\frac{n}{2})$. So we have $2T(\frac{n}{2})$ in lines 6-8.

For the combine step, Lines 11-13 take $\mathcal{O}(1)$ time since $right$ and $left$ each has at most 2 elements and hence this for loop takes constant time. Furthermore, since each of $left$ and $right$ have at most 2 elements each, we have that *potential* has at most 4 elements. Hence, lines 15-22 will only take $\mathcal{O}(n)$ time as the outer loop only has 4 iterations, whereas the inner loop takes $\mathcal{O}(n)$ iterations. The body of each of the for loops takes only $\mathcal{O}(1)$ time. Hence, lines 15-22 indeed takes $\mathcal{O}(n)$ time.

Therefore, the non-recursive part takes $\mathcal{O}(n)$ time.

Hence, we get the following recursive relation which ignores floors as we assume n is a power of 2.

$$T(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n \leq 1 \\ 2T(\frac{n}{2}) + \mathcal{O}(n) & \text{if } n > 1 \end{cases}$$

Now we will apply the Master Theorem. Looking at the above recurrence relation we have that $a = 2$, $b = 2$, $f(n) = n$. Hence, $\log_b a = \log_2 2 = 1$. Let $k = 0$. We have that $f(n) = \Theta(n) = \Theta(n^{\log_b a} \log^k n)$. Hence, by Case 2 of the Master Theorem, we get that $T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n) = \mathcal{O}(n \log n)$.

Therefore, $T(n) = \mathcal{O}(n \log n)$ as required.

Part c)

We will prove the correctness of DecideCandidate(L). Let $n = len(L)$.

Let $P(n)$ be the predicate "DecideCandidate(L) is correct on inputs of size n".

Show: $\forall n \in \mathbb{N}, P(n)$.

Proof. **Base Cases:** For $n = 0$, we have that $L = []$ is an empty list of names. Every name in L trivially appears more than $\frac{n}{3}$ times. Hence, DecideCandidate(L) correctly returns L. So $P(0)$ holds.

For $n = 1$, we have that L contains exactly 1 name. This name appears one time which is more than $\frac{n}{3} = \frac{1}{3}$. Hence, DecideCandidate(L) correctly returns L. So $P(1)$ holds.

Inductive Hypothesis: Assume $P(i)$ holds for all $0 \leq i < n$ where $n \in \mathbb{N}$ such that $n > 1$.

Want to Show: $P(n)$ holds.

WLOG, assume n is a power of 2. Since $n > 1$, we have two recursive calls on $left = DecideCandidate(L[0, ..., mid])$ and $right = DecideCandidate(L[mid + 1, ..., n - 1])$ where

$mid = \frac{n}{2}$. Since $P(\frac{n}{2})$ holds by inductive hypothesis, we have that $left$ and $right$ correctly return the names that appear more than $\frac{len(left)}{3} = \frac{n}{6}$ times and $\frac{len(right)}{3} = \frac{n}{6}$ times respectively within each sublist.

Note that for any name in L that appears more than $\frac{n}{3}$ times, we have that this name must appear more than $\frac{n}{6}$ times in at least one of $left$ or $right$. Hence, any name that must be correctly returned from DecideCandidate(L) must be part of $left$ or $right$.

Lines 10-13 combines $left$ and $right$ into *potential* without any possible duplicates.

Each of $left$ and $right$ has at most 2 elements each by inductive hypothesis. Hence, *potential* has at most 4 elements. And lines 15-22 finds the elements of *potential* that appear more than $\frac{n}{3}$ times in L and appends those names in the list *candidates*. And we return *candidates* which correctly contains all the required names. Hence, $P(n)$ holds.

Therefore, by induction we have that $\forall n \in \mathbb{N}, P(n)$ holds. Therefore, DecideCandidate(L) is correct, as required. □

Question 2

Part a)

On Piazza, the instructor indicated that we can assume that the treasure parameters of the form (x_i, y_i) can be part of a list.

So assume that L is a list of treasure parameters where each element is of the form (x_i, y_i) where x_i and y_i are the parameters 'easiness' and 'value' respectively. And we will assume that any two treasures will differ in at least one of their parameters.

Before we define our divide and conquer algorithm, we will do a global sorting of L in non-decreasing order by x coordinate, and if two treasures have the same x coordinate, then we continue to order in non-decreasing order by y coordinate. This is a form of lexicographic ordering or standard-string-ordering.

Note: The following algorithm is written in pseudocode with some Python-like syntax.

ValuableTreasures(L)

Sort L via mergesort in non-decreasing order by x coordinate, and if two treasures have the same x coordinate, then we continue to order in non-decreasing order by y coordinate.
return $Valuable(L)$

Valuable(L)

$n \leftarrow len(L)$ ▷ Assume distinct treasures differ in at least one parameter
if $n \leq 1$ **then**
    return L
end if
▷ List slicing takes constant time
$mid \leftarrow \lfloor \frac{n}{2} \rfloor$ ▷ Note, list indices are inclusive, 0-based
$left \leftarrow Valuable(L[0, ..., mid])$ ▷ Note, $left$ is in our lexicographical order
$right \leftarrow Valuable(L[mid + 1, ..., n - 1])$ ▷ Note, $right$ is in our lexicographical order

Let $smallest\_right \leftarrow right[0]$. Write $smallest\_right = (x_j, y_j)$ for convenience.
$new\_left \leftarrow [\ ]$
for each (x_i, y_i) in $left$ **do**
    if not ($x_j \geq x_i$ and $y_j \geq y_i$) **then**
        $new\_left.append((x_i, y_i))$
    end if
end for

return $new\_left + right$ ▷ List concatenation

Part b)

Let $n = len(L)$. Let $T(n)$ be the worst-case running time of ValuableTreasures(L).

We know Step 1 takes $\mathcal{O}(n \log n)$ since we are doing a global mergesort. And Step 2 is our divide and conquer algorithm. In Step 2 we make a call to Valuable(L). Let $S(n)$ be the worst-case running time of Valuable(L).

The base case of Valuable in lines 1-4 takes $\mathcal{O}(1)$ time when $n = len(L) \leq 1$. If $n > 1$, we have two recursive calls of length approximately $\frac{n}{2}$ in lines 6-8. WLOG, assume n is a power of 2. Hence, we have $2S(\frac{n}{2})$. Lines 10-11 takes constant time since it is just accessing the first element of *right* and declaring a new empty list. For the combine step, we have a for loop on lines 12-16 that has at most $\frac{n}{2}$ iterations. And the body of this loop takes constant steps. Hence, this for loop takes $\mathcal{O}(n)$ time. The final list concatenation on line 18 also takes $\mathcal{O}(n)$ time. Hence, the combine step takes $\mathcal{O}(n)$ time. Hence, consider the following recurrence for $S(n)$ which ignores floors as we assume n is a power of 2.

$$S(n) \leq \begin{cases} \mathcal{O}(1) & \text{if } n \leq 1 \\ 2S(\frac{n}{2}) + \mathcal{O}(n) & \text{if } n > 1 \end{cases}$$

Now we will apply the Master Theorem. Looking at the above recurrence relation we have that $a = 2$, $b = 2$, $f(n) = n$. Hence, $\log_b a = \log_2 2 = 1$. Let $k = 0$. We have that $f(n) = \Theta(n) = \Theta(n^{\log_b a} \log^k n)$. Hence, by Case 2 of the Master Theorem, we get that $S(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n) = \mathcal{O}(n \log n)$.

Therefore, $S(n) = \mathcal{O}(n \log n)$ as required.

Hence, we have that $T(n) = \mathcal{O}(n \log n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$ when taking into account the mergesort.

Therefore, the runtime of our algorithm ValuableTreasures(L) is $\mathcal{O}(n \log n)$.

c)

We will prove the correctness of ValuableTreasures(L). Step 1 is just a preprocessing step that does a global mergesort in lexicographical order. Hence, we just need to prove the correctness of Valuable(L).

Assume any of our lists L are sorted in our defined lexicographical order by the mergesort.

Let $n = len(L)$. Let $P(n)$ be the predicate "Valuable(L) is correct on inputs of size n".

Show: $\forall n \in \mathbb{N}, P(n)$.

Proof. **Base Cases:** For $n = 0$, we have that L is empty. Hence, trivially we correctly return L. For $n = 1$, we have that L has a single treasure. Hence, this single treasure is valuable

by definition. Hence, we correctly return L. So $P(0)$ and $P(1)$ hold.

Inductive Hypothesis: Assume $P(i)$ holds for all $0 \leq i < n$ where $n \in \mathbb{N}$ such that $n > 1$.

Want to Show: $P(n)$ holds.

WLOG, assume n is a power of 2. Hence, we have $mid = \frac{n}{2}$. Since $P(\frac{n}{2})$ holds by the inductive hypothesis, we have that $left$ and $right$ are the valuable treasures from $L[0, ..., mid]$ and $L[mid + 1, ..., n - 1]$ respectively.

Note, every treasure in $right$ must also be valuable with respect to L given our lexicographical ordering. Hence, the only possible treasures that may not be valuable with respect to L are contained in $left$.

We have that $smallest\_right = (x_j, y_j)$ is the first element of the list $right$ which is in our lexicographical ordering. Hence (x_j, y_j) is the treasure with the smallest x-coordinate in $right$. Since this treasure is valuable, it must be the case that y_j is the maximum y-coordinate in $right$. If y_j were not the maximum y-coordinate in $right$, we would be able to find a pair (x_k, y_k) in right such that $x_k \geq x_j$ and $y_k > y_j$ which would imply that the treasure with parameters (x_j, y_j) is not valuable in $right$, a contradiction.

Every treasure (x_i, y_i) in $left$ is such that $x_i \leq x_j$. Since y_j is the maximum y-coordinate in $right$, it is sufficient to simply compare each (x_i, y_i) in $left$ with (x_j, y_j) to check if (x_i, y_i) is not valuable with respect to L. Comparing (x_i, y_i) with any other treasure in $right$ is redundant as any other treasure in right has y coordinate smaller than y_j. Treasures (x_i, y_i) in $left$ that are not valuable with respect to L are not appended to new_left in our for loop in lines 12-16. All other treasures in $left$ are appended to new_left.

Finally we return $new\_left + right$ which concatenates new_left and $right$ which correctly contain all the valuable treasures with respect to L, as required. Hence, $P(n)$ holds.

Therefore, by induction we have that $\forall n \in \mathbb{N}, P(n)$ holds. Therefore, Valuable(L) is correct.

Therefore, ValuableTreasure(L) is correct, as required. □

Question 3

Part a)

ScheduleTwoClassrooms$([s_1, f_1), ..., [s_n, f_n))$

1: Sort and relabel each lecture interval by finish time so that $f_1 \leq f_2 \leq ... \leq f_n$.
2: Let $l_1 \leftarrow 0$ where l_1 is the finish time of the latest lecture scheduled in classroom 1.
3: Let $l_2 \leftarrow 0$ where l_2 is the finish time of the latest lecture scheduled in classroom 2.
4: Let $A_1 \leftarrow \{\}$
5: Let $A_2 \leftarrow \{\}$
6: **for** i = 1 to n **do**
7: Check if i is compatible with classroom 1 by checking if $s_i \geq l_1$
8: Check if i is compatible with classroom 2 by checking if $s_i \geq l_2$
9: Among the compatible classrooms above, find $j \in \{1, 2\}$ such that l_j is maximum.
10: Add i to A_j
11: $l_j \rightarrow f_i$
12: **end for**
13: **Return** (A_1, A_2)

Part b)

We will analyze the worst-case running time of ScheduleTwoClassrooms. Consider an input $[s_1, f_1), ..., [s_n, f_n)$ with n lectures.

Line 1 takes $\mathcal{O}(n \log n)$ time using a mergesort sorting algorithm. Lines 2-5 take $\mathcal{O}(1)$ time. The for loop has n iterations, and the body of the for loop takes $\mathcal{O}(1)$ time. Hence, the entire for loop on lines 6-12 take $\mathcal{O}(n)$ time.

Hence, ScheduleTwoClassrooms runs in $\mathcal{O}(n \log n)$ time.

Part c)

Consider the following definitions.

Let $S_0 = (\emptyset, \emptyset)$.

Let $S_j = (A_{j_1}, A_{j_2})$ be the partial schedule picked by our greedy algorithm from the first j jobs where $A_{j_1}, A_{j_2} \subseteq \{1, ..., j\}$ are what is scheduled in classrooms 1 and 2 respectively and $A_{j_1} \cap A_{j_2} = \emptyset$.

A partial solution S_j is promising if there is a way to extend it to an optimal solution $O_j = (O_{j_1}, O_{j_2})$ where O_{j_1}, O_{j_2} are what is scheduled in classrooms 1 and 2 respectively and

$A_{j_1} \subseteq O_{j_1}$ and $A_{j_2} \subseteq O_{j_2}$ and $O_{j_1} \cap O_{j_2} = \emptyset$.

From hereafter, we will only refer to S_j and O_j for each j without explicitly referring to $A_{j_1}, A_{j_2}, O_{j_1}, O_{j_2}$ to avoid cumbersome notation and for readability.

Want to Show: S_j is promising for all $j \in \{0, 1, ..., n\}$.

Base Case: For $j = 0$, we have $S_0 = (\emptyset, \emptyset)$. Any optimal solution extends S_0. Hence, S_0 is promising.

Inductive Hypothesis: Assume S_j is promising where $0 \leq j < n$.

Want to Show: S_{j+1} is promising.

By inductive hypothesis, since S_j is promising, we know there exists some optimal solution O_j extending S_j.

Now consider lecture $j + 1$. We have the following cases.

Case 1: Our greedy algorithm does not schedule $j + 1$. Hence, $S_{j+1} = S_j$. Hence, lecture $j + 1$ conflicts with some lecture in $S_{j+1} = S_j$ in some classroom. Since O_j extends S_j, we have that $j + 1$ conflicts with some lecture in O_j in some classroom. So let $O_{j+1} = O_j$. Since $O_{j+1} = O_j$ is optimal and extends $S_{j+1} = S_j$, we have that O_{j+1} is optimal and extends S_{j+1}. Hence, S_{j+1} is promising.

Case 2: Our greedy algorithm schedules $j+1$ in classroom 1 in S_{j+1}. Consider the following sub-cases.

Subcase 2 i) The optimal solution O_j also schedules $j + 1$ in classroom 1.

So let $O_{j+1} = O_j$. Clearly O_{j+1} extends S_{j+1} and O_{j+1} is optimal since $O_{j+1} = O_j$ where O_j is optimal. Hence, S_{j+1} is promising.

Subcase 2 ii) The optimal solution O_j schedules $j + 1$ in classroom 2.

We know that S_j and O_j have the same schedule up to the j-th iteration of our greedy algorithm. Consider l_1 and l_2 as defined in our greedy algorithm after the j-th iteration.

Since our greedy algorithm schedules $j + 1$ in classroom 1, we have that $s_{j+1} \geq l_1$ since we need compatability, and we have $l_1 \geq l_2$ by our greedy strategy. Hence, $s_{j+1} \geq l_1 \geq l_2$.

Now consider a solution O_{j+1} defined as follows. O_{j+1} is the same as O_j except that every lecture scheduled after time l_1 in classroom 1 is swapped with every lecture scheduled after time l_2 in classroom 2. We know we can make this swap since $j + 1$ was scheduled in O_j in classroom 2, and since $s_{j+1} \geq l_1 \geq l_2$, we have that every lecture after l_2 in classroom 2 (in-

cluding $j+1$) can be scheduled after l_1 in classroom 1. Similarly, since $l_1 \geq l_2$, every lecture scheduled after l_1 in classroom 1 can trivially be scheduled after l_2 in classroom 2. Since we simply swapped lectures, we have that O_{j+1} has the same number of lectures scheduled as O_j. Hence, O_{j+1} is optimal. Since O_{j+1} schedules $j+1$ in classroom 1, O_{j+1} extends S_{j+1}. Hence, S_{j+1} is promising.

Subcase 2 iii) The optimal solution O_j does not schedule $j+1$ in either classroom.

Since we assumed that our greedy solution schedules $j+1$ in classroom 1, we know that $j+1$ is indeed compatible with S_j when scheduled in classroom 1. And since O_j extends S_j, where $j+1$ is not compatible with O_j, but is compatible with S_j, we must have that $j+1$ is not compatible with some lecture $k \in \{j+2, ..., n\}$ for classroom 1. And, $j+1 < k$.

Now, assume for the sake of contradiction that there existed two distinct lectures $k_1, k_2 \in \{j+2, ..., n\}$ scheduled in classroom 1 in O_j such that k_1 and k_2 are incompatible with $j+1$ in classroom 1. We know $j+1 < k_1$ and $j+1 < k_2$. Since we ordered our lectures by finish time we know that $f_{j+1} \leq f_{k_1}$ and $f_{j+1} \leq f_{k_2}$. But since $j+1$ is incompatible with both k_1 and k_2 and since $f_{j+1} \leq f_{k_1}$ and $f_{j+1} \leq f_{k_2}$, we must have that k_1 and k_2 are incompatible with each other in classroom 1. This contradicts the fact that k_1 and k_2 are scheduled in O_j which is an optimal solution. Hence, there cannot exist two distinct lectures k_1 and k_2.

Hence, there is only one lecture $k > j+1$ scheduled in classroom 1 in O_j that is incompatible with $j+1$. So let O_{j+1} be the same solution as O_j except that we replace the scheduling of lecture k with the lecture $j+1$ in classroom 1. Since $j+1$ only conflicted with k, we know that this swap works. And since O_{j+1} has the same number of scheduled lectures as O_j, we have that O_{j+1} is optimal. And since O_{j+1} schedules $j+1$ in classroom 1, we have that O_{j+1} extends S_{j+1}. Hence, S_{j+1} is promising.

Case 3: Our greedy algorithm schedules $j+1$ in classroom 2. This Case 3 is symmetric to Case 2 except with the roles of classroom 1 and classroom 2 reversed.

In all 3 cases we have that S_{j+1} is promising.

Hence, we have shown that S_j is promising for all $j \in \{0, 1, ..., n\}$.

Since S_n is promising, we know that our algorithm does indeed find an optimal solution, as required.

Question 4

Part a)

Consider an ordering by the Earliest Finish Time (EFT) such that we have $f_1 \leq f_2 \leq ... \leq f_n$. We greedily select the next interval with the earliest finish time that makes the schedule contiguous (with possible overlap if necessary).

Consider the following counterexample which shows this strategy is not optimal.

Consider the three intervals $[0, 1), [0, 5), [5, 10)$.

Our greedy strategy will select all 3 intervals above to schedule between time 0 and time 10 since we select based on earliest finish time.

However, this solution is not optimal because an optimal solution can be just the two intervals $[0, 5), [5, 10)$ which covers the entire range in fewer intervals (2 instead of 3).

Therefore, this greedy strategy would not work.

Please see the next page for parts b), c), d).

Part b)

Note, we will assume that a set has no duplicates. i.e. If $S = \{5\}$ and we add 5 to S, we would get that $S = \{5\}$. Please keep this in mind when reading the algorithm below.

Greedy Strategy: Sort the intervals by Earliest Start Time (EST). If $[s_p, f_p)$ is the most recently scheduled interval, we will greedily select the next interval $[s_q, f_q)$ in our ordering such that $s_q \leq s_p$ and f_q is maximal.

TASchedule($[s_1, f_1), ..., [s_n, f_n)$)

```
1: Sort and relabel each lecture interval by earliest start time so that s_1 ≤ s_2 ≤ ... ≤ s_n.
2: start ← s_1
3: end ← start
4: schedule ← {}                                   ▷ A set has no duplicates
5: candidate ← 1                                   ▷ Initialize candidate to first interval
6:
7: for each i from 1 to n do
8:     if s_i ≤ start and f_i ≥ end then
9:         candidate ← i
10:        end ← f_i
11:    else
12:        start ← end
13:        Add candidate to schedule               ▷ A set has no duplicates
14:
15:        if end ≥ f_n then
16:            exit the for loop early.
17:        end if
18:
19:        if s_i ≤ start then                     ▷ Note, start was updated
20:            candidate ← i
21:            end ← f_i
22:        end if
23:
24:    end if
25: end for
26:
27: if end ≥ f_n then
28:    return schedule
29: else
30:    return None - No solution exists!
31: end if
```

Part c)

We will analyze the runtime of TASchedule.

Line 1 takes $\mathcal{O}(n \log n)$ time since we are sorting (i.e. via mergesort).

Clearly lines 2-6 take constant time. We have a for loop from lines 7-25 that has at most n iterations. Each iteration takes constant time. So the for loop runs in $\mathcal{O}(n)$ time.

Finally lines 26-31 run in constant time.

Therefore, TASchedule runs in $\mathcal{O}(n \log n)$ time in the worst case.

Part d)

We will prove that our greedy algorithm always produces an optimal solution.

Proof. Note, it is possible that an input of intervals $[s_1, f_1), ..., [s_n, f_n)$ may not have any solution if there is a gap not covered by any interval. Hence, our greedy algorithm cannot possibly find a solution either. Our greedy algorithm will in fact return **None** when there is such a gap.

Hence, consider an input of intervals $[s_1, f_1), ..., [s_n, f_n)$ and assume that there exists a solution. Hence, there exists an optimal solution.

Assume for the sake of contradiction that our greedy algorithm does not produce an optimal solution for input $[s_1, f_1), ..., [s_n, f_n)$.

Let $\{i_1, i_2, ..., i_k\} \subseteq \{1, ..., n\}$ be the solution given by our greedy algorithm sorted by start time. Note, we simply refer to the interval indices in our solution.

Let $\{j_1, j_2, ..., j_m\} \subseteq \{1, ..., n\}$ be an optimal solution sorted by start time that agrees with our greedy solution for the **most** indices as possible. i.e. Let r be the largest possible integer such that $i_q = j_q$ for all $q \in \{1, ..., r\}$.

Note, since we are assuming that our greedy algorithm is not optimal, we have that $m < k$.

We know $r \leq m$. Note, if $r = m$, then we would have that the first r intervals selected by our greedy solution would make an optimal solution. But then our greedy algorithm would have terminated after the first r iterations with an optimal solution which would be a contradiction. Hence, we must have $r < m$.

So we have that $r < m < k$.

Consider $r + 1$ where $r + 1 \leq m < k$. We have that $i_{r+1} \neq j_{r+1}$ by definition of r.

And since our schedules are contiguous without gaps, we know that $s_{i_{r+1}} \leq s_{i_r}$ and $s_{j_{r+1}} \leq s_{j_r}$.

Since $i_r = j_r$, we know that $[s_{i_r}, f_{i_r}) = [s_{j_r}, f_{j_r})$. Hence, $s_{i_r} = s_{j_r}$.

Since $s_{i_{r+1}} \leq s_{i_r}$ and $s_{i_r} = s_{j_r}$, we have that $s_{i_{r+1}} \leq s_{j_r}$.

Recall our greedy strategy. If $[s_p, f_p)$ is the most recently scheduled interval, then we greedily select the next interval $[s_q, f_q)$ in our earliest start time ordering such that $s_q \leq s_p$ and f_q is maximal.

Hence, by our greedy strategy and maximality, we must have that $f_{i_{r+1}} \geq f_{j_{r+1}}$.

Notice that we now have $s_{i_{r+1}} \leq s_{j_r}$ and $f_{i_{r+1}} \geq f_{j_{r+1}}$. Hence, in our optimal solution we can replace the interval with index j_{r+1} with the interval with index i_{r+1} and still have an optimal solution. The solution remains optimal after this swap since the number of intervals in our optimal schedule has not changed (we only made 1 swap), and our schedule remains contiguous since $s_{i_{r+1}} \leq s_{j_r}$ and $f_{i_{r+1}} \geq f_{j_{r+1}}$.

But now our modified optimal solution agrees with our greedy solution for $r+1$ many intervals.

This is a contradiction. Therefore, our greedy algorithm is optimal, as required. $\square$

Question 5

Citations/Sources: Our main idea is inspired from the famous Egg Drop Puzzle. The Egg Drop Puzzle is a well-known interview question given by Microsoft. We will use the Egg Drop algorithm as a sub-routine. Given n floors and k eggs, the egg drop algorithm determines the minimal number of floors needed to be checked to find the highest floor we can drop an egg without breaking. It is similar to the voltage optimization problem, except the egg drop problem finds a minimal number of floors, and not the floor itself. We will use a version of the egg drop algorithm in our subroutine **FewestTests**.

The Egg Drop problem and Brilliant article.

Source: `https://brilliant.org/wiki/egg-dropping/`

Consider the following algorithm. Assume A is our array of n non-increasing bulb voltages, and k is the number of bulbs we are willing to waste. Assume A is nonempty, and $k \geq 1$.

Note, if every bulb in the array A can withstand 120V, we will return $len(A)$. If no bulb in A can withstand 120V, we will return 0.

MaxVoltage(A, k)

```
1: return MaxIndex(A, k, 1, len(A))                    ▷ Note, 1-based indexing
```

MaxIndex(A, k, low, high)

```
1: if k = 1 then                          ▷ 1 bulb willing to break; do linear search
2:     for each i from low to high do
3:         if A[i] < 120 then
4:             return i − 1
5:         end if
6:     end for
7:     return high
8: end if
9:
10: n ← high − low + 1
11: index ← FewestTests(n, k)                    ▷ See next page for FewestTests
12: if A[index] < 120 then
13:     return MaxIndex(len(A[low, ..., index − 1]), k − 1)                ▷ if breaks
14: else
15:     return MaxIndex(len(A[index + 1, ..., high]), k)                   ▷ if intact
16: end if
```

```
FewestTests(n,k)        ▷ This is the Microsoft "Egg Drop" Algorithm; see citations.
1: if n = 1 then                                          ▷ n is number of bulbs
2:     return 1                                           ▷ 1 test for 1 bulb total
3: end if
4: if k = 1 then                                          ▷ k is number of bulbs willing to break
5:     return n                                           ▷ n tests for 1 bulb to break
6: end if
7:
8: m = +∞
9: for each i from 1 to n do
10:    breaks ← FewestTests(i − 1, k − 1)                 ▷ Consider bulbs before i if breaks
11:    intact ← FewestTests(n − i, k)                     ▷ Consider bulbs after i if intact
12:    temp = max(breaks, intact)
13:    m = min(m, temp)
14: end for
15: return m + 1
```

Note the above subroutine FewestTests can be faster if we used dynamic programming and memoization. However, in this question we are only concerned with the number of bulbs we are testing in MaxIndex, and hence we do not care about the runtime of FewestTests. Hence, we will not use dynamic programming and memoization here.