This homework is due on gradescope Friday November 8th at 11:59pm on gradescope. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommended though not required.

**Question 1** (Deterministic Order Statistics, 35 points). There is in fact a deterministic algorithm for order statistics that runs in linear time. See the pseudocode below:

\begin{verbatim}
DeterministicSelect(L,k) \ returns the kth largest element of L
If |L| < 20 compute the order statistic manually and return the answer
Split L into sets of size 5
Compute the median of each set and let M be the set of medians
Pivot <- DeterministicSelect(M,|M|/2)
Split L into L_+, L_=, L_- based on comparisons to Pivot
If |L_+| => k
   Return DeterministicSelect(L_+,k)
Else if |L_+| + |L_=| => k
   Return Pivot
Else
   Return DeterministicSelect(L_- , k - |L_+| - |L_=|)
\end{verbatim}

The rest of this question will be about analyzing this algorithm.

(a) Prove that if \( L \) has size \( n \), then the Pivot selected in this algorithm will have at least \( \frac{3n}{10} + O(1) \) elements of \( L \) on either side of it. [15 points]

(b) Give a recurrence relation for the runtime of \texttt{DeterministicSelect}. [5 points]

(c) Show that this algorithm runs in at worst linear time on all inputs. Note that you cannot use the Master Theorem here, and will have to instead prove it directly by induction. Make sure to be careful to show that there is a universal constant for your Big-O, and that this does not change with \( n \) (i.e. don’t make the mistake from the last part of the last question in Homework 0). [15 points]

**Solution.**

a) Let \( L' \) denote the set of 5-sets obtained by splitting of \( L \). So, \( |L'| \leq n/5 + c_1 \). Let \( M \) be the set of medians of each 5-set in \( L' \), and hence \( |M| = |L'| \). Note that each element of \( M \) is mapped to a single 5-set in \( L \) and consequently unique elements of \( L \).

The pivot, say \( P \), obtained using

\( \text{P<DeterministicSelect}(M,|M|/2) \)

gives the \((|M|/2)\text{th} \) largest element of set \( M \). Hence, there are \(|M|/2 + c_2 \) elements of \( M \) on either side of \( P \). But each element of \( M \) is itself a median of the corresponding 5-set in \( L' \), consequently \( \forall m_i \in M \), except last element, there are atleast 3 elements \( \leq m_i \) and 3 elements \( \geq m_i \) in the corresponding 5-set. But we know that the elements of each 5-set correspond to unique elements of \( L \). Putting these two together, we have atleast 3\(|M|/2 + c_3 \) i.e. \( 3n/10 + c_4 \) elements of \( L \) on either side of \( P \).

b) We can see that there are two recursive calls made to \texttt{DeterministicSelect} in the pseudo-code. The
size of inputs to second recursive call varies based on \(|L_+|\) and \(|L_-|\). The first recursive call on set \(M\) takes \(T(|M|) = T(n/5) + c_1\) amount of work and the second recursive call takes atmost \(T(|L_+|), T(|L_-|) \leq T(7n/10) + c_9\). With respect to non-recursive parts of function call, note that sorting a set of constant size, 5 here, is constant operations per sort. There are \(n/5\) such sorts. Similarly, splitting into sets of size 5 takes \(O(n)\) operations. Comparison to pivot and creating \(|L_+|, |L_-|, |L_-|\) is also \(O(n)\) as we just need to compare each element of \(L\) to pivot and place it in the correct set. The rest of the operations are constant time. Putting these together, the recursion equation is

\[
T(n) \leq T(n/5) + T(7n/10) + c'n + c''
\]

we can absorb the \(c''\) into \(c'\) by creating, say \(c = c' + c''\). So, recursion equation becomes

\[
T(n) \leq T(n/5) + T(7n/10) + cn
\]

c) We prove by induction that this algorithm indeed runs in linear time i.e. we wish to prove \(T(n) \leq c_0n\) for some constant \(c_0\). Assume \(c_0 \geq 10c\)

Consider the strong form of induction and assume \(T(j) = c_0j \forall j \leq k\).

\[
T(k+1) \leq T((k+1)/5) + T(7(k+1)/10) + c(k+1)
\]

\[
T(k+1) \leq c_0(k+1)/5 + c_0(7(k+1)/10) + c(k+1)
\]

\[
T(k+1) \leq (k+1)(c_0/5 + 7c_0/10 + c)
\]

\[
T(k+1) \leq (k+1)(9c_0/10 + c)
\]

\[
T(k+1) \leq (k+1)(9c_0/10 + c_0/10)
\]

\[
T(k+1) \leq (k+1)c_0
\]

To satisfy the base case, we can choose \(c_0 \geq T(j)/j \forall j \leq 20\) so that \(T(j) \leq j c_0 \forall j \leq 20\). Hence, we found a constant \(c_0\) which bounds \(T(n)\) by \(c_0n\) i.e. \(T(n)\) is \(O(n)\).

**Question 2** (Cheapest Bookings, 25 points). Jamie runs a travel agency. For a particular trip, he has a calendar for the \(n\) day season during which that trip is possible. Since airfare and hotel costs are variable, for each day he has a record of the cost of the cheapest trip leaving on that day. He also has a list of \(n\) clients. For each client, they have a date range in which they are willing to travel. Jamie wants an algorithm that given this list of the prices for each of the \(n\) days and the date ranges of each of the \(n\) clients allows him to compute for each client the price of the cheapest day in their date range.

**Give an algorithm for doing this. For full credit, your algorithm should run in time \(O(n \log(n))\) or better.**

**Formalizing problem statement:**
The question can be stated formally as follows. We are given an array \(A\) of \(n\) numbers. We are also given a range of intervals \((L_i, R_i)\) for \(i\) ranging from 1 to \(n\). For each interval \((L_i, R_i)\), we need to identify the minimum number in the sub-array \(A[L_i, L_i + 1, L_i + 2, ..., R_i - 1, R_i]\).

**Solution 1:**

**Algorithm:**
The naive way to calculate the minimum number within each interval would be to go through all the numbers that are within an interval and keep track of the minimum. In the worst case, we might need to iterate through the entire array \(A\) for each interval. This would take \(O(n^2)\) time.

It is worth noticing that the intervals might be overlapping. Hence, going through all numbers in an interval and repeating that for all the intervals is redundant work. To avoid this redundancy, we can try to pre-compute the minimum number for certain sub-arrays of \(A\) and use that to potentially calculate the minimum of each interval much faster.

More formally, we will store the minimum number for the sub-array \(A[1,2,....n]\) of size \(n\). Then, we store the minimum number for sub-arrays of size \(n/2\) i.e. \(A[1,....n/2]\) and \(A[1+n/2,....n]\). This would be followed
by sub-arrays of size n/4 i.e. A[1,...,n/4], A[1+n/4,...,n/2], A[1+n/2,...,3n/4] and A[1+3n/4,...,n]. And so on till we reach sub-arrays of size 1. The minimum of all the sub-arrays are stored in a hash-map.

This can be done as follows:

H = Hash-Map()
def calculate_and_store_minimum(A,left,right):
    if(left == right):
        H (left,right) = A[left]
        return A[left]
    else:
        mid = (left + right)/2
        min_left = calculate_and_store_minimum(A,left,mid)
        min_right = calculate_and_store_minimum(A,mid+1,right)
        minimum = min(min_left,min_right)
        H(left,right) = minimum

To compute the minimum in the interval [L,R] of A[1,2,...,n], we can split the queries into two. The first query would be to compute the minimum in A[1,...,n/2] that also lies in A[L,....,R] and the second query would be to compute the minimum in A[1+n/2,...,n] that also lies in A[L,....,R]. Taking the minimum of these two values gives us the minimum in the range [L,R] in A[1,...,n]. This process can be repeated recursively. We only recurse down the tree if the sub-array has some overlap with [L,R].

- If we reach a sub-array A[l,...,r] such that the [L,R] contains (l,r) and we have the minimum number in sub-array A[l,...,r] stored in H, we can directly return H(l,r).

Note that [L,R] refers to the indices in the original array A and not in the sub-array for which we are recurring.

Let Compute(A,left,right,L,R) be the function that computes the minimum value in the sub-array A[left,...,right] such that the index of the value selected lies in between L and R.

The minimum for an interval [L, R] will be calculated by calling Compute(A,1,n,L,R). This is defined as follows:

Define Compute (A, left, right, L, R):
    if (left, right) is contained within [L,R):
        return H[ left, right ]
    if (left, right) has no overlap with [L,R]:
        return INF
    else:
        mid = (left + right) / 2
        if [left,mid] overlaps with [L,R]:
            min_left = Compute(A,left,mid,L,R)
        else: min_left = INF
        if [mid+1,right] overlaps with [L,R]:
            min_right = Compute(A,mid+1,right,L,R)
        else: min_right = INF
        return min(min_left,min_right)

Correctness: We can prove via induction that Compute(A,left,right,L,R) computes the minimum value in the sub-array A[left,...,right] such that the index of the value selected lies in between L and R.

Base Case ( A[left,...,right] has size 1 ): If L,R contains left (or right), our algorithm returns A[left] and otherwise returns INF (i.e. the minimum doesn’t exist here).
Assume that the proposed algorithm computes the correct answer for \( A[1, \ldots, 1+k-1] \) (array of size \( k \times n \)). When we call Compute\( (A,1,n,L,R) \), if \([L,R]\) contains the interval \((1,n)\), then we return the minimum number in \( A[1,\ldots,n] \). If not, we split \( A \) into two halves. If \([L,R]\) overlaps with the left half, we call Compute\( (A,1,n/2,L,R) \). If \([L,R]\) overlaps with the right half we call Compute\( (A,1+n/2,L,R) \). By the inductive assumption, both the recursive calls (if made) return the correct minimum. We then compare these to get the overall minimum in \([L,R]\).

**Runtime:** The proposed algorithm includes two main components. (a) Storing minimum values for the intervals in a hash-map and (b) Calculating the minimum value for each query \((L_i, R_i)\).

(a) The function call to calculate the minimum for an \( n \)-length sub-array recurses on it’s two halves. It then sets the minimum for the \( n \)-length sub-array to be the minimum of the minimas calculated in the two halves. This can be represented as \( T(n) = 2T(n/2) + O(1) \). Hence, by the Master Theorem, this takes \( O(n) \) time.

(b) Consider the recursion tree generated while calling Compute\( () \). Each recursive call calculates the minimum element in a sub-array such that the element lies in \([L,R]\) of the original array \( A \). The following figure is a good-representation.

The claim is that at-most four sub-arrays would be evaluated at each level and we will prove that via contradiction.

Consider that the sub-arrays evaluated at a level are \((l_1, r_1), (l_2, r_2), \ldots, (l_K, r_K)\). We know that \([L,R]\) overlaps with all the sub-arrays otherwise we would not have made a recurse call to that sub-array. Moreover, the sub-arrays which are evaluated at any level must be contiguous (one following the other). This is because if \([L,R]\) is overlapping with \((l_i, r_i)\) and \((l_{i+2}, r_{i+2})\) then it must also overlap with \((l_{i+1}, r_{i+1})\).

Consider four cases:

**Case 1:** \((l_1, r_1)\) is a right child in the recursion tree and \( K \) is odd. If \( K \) less than 4 we are done. Otherwise, the sub-arrays evaluated at this level would include only pairs of siblings except the first sub-array whose left sibling is not included. Now, \([L,R]\) contains the sub-arrays represented by all pairs of siblings except (possibly) the last pair. In these cases, the recursive call at the previous level would have directly returned the value stored in \( H \) for these sub-arrays. Hence, the only sub-arrays that could be evaluated are \((l_1, r_1)\), \((l_{K-1}, r_{K-1})\) and \((l_K, r_K)\).

**Case 2:** \((l_1, r_1)\) is a right child and \( K \) is even. If \( K \) less than 4 we are done. Otherwise, the sub-arrays evaluated at this level would include only pairs of siblings except the first sub-array whose left sibling is not included and the last sub-array whose right sibling is not included. Similar to Case 1, \([L,R]\) contains all the sub-arrays represented by the pairs of siblings and the previous level would have straightaway returned the value stored in \( H \). Hence, the only sub-arrays that could be evaluated are \((l_1, r_1)\) and \((l_K, r_K)\).

**Case 3:** \((l_1, r_1)\) is a left child and \( K \) is odd. If \( K \) less than 4 we are done. Otherwise, the sub-arrays evaluated at this level would include only pairs of siblings except the last sub-array whose right sibling is not included. \([L,R]\) contains the sub-arrays represented by all pairs of siblings except possible the first pair (since the \([L,R]\) might not completely contain the first sub-array) and the previous level would have straightaway returned the value stored in \( H \). Hence, the only sub-arrays that could be evaluated are \((l_1, r_1)\), \((l_2, r_2)\) and \((l_K, r_K)\).

**Case 4:** \((l_1, r_1)\) is a left child and \( K \) is even. If \( K \) less than 4 we are done. Otherwise, the sub-arrays evaluated at this level would include only pairs of siblings. \([L,R]\) contains the sub-arrays represented by all pairs of siblings except possible the first pair (since the \([L,R]\) might not completely contain the first sub-array) and the last pair. The previous level would have straightaway returned the value stored in \( H \) for these sub-arrays. Hence, the only sub-arrays that could be evaluated are \((l_1, r_1)\), \((l_2, r_2)\), \((l_{K-1}, r_{K-1})\) and \((l_K, r_K)\).

Hence, at each level, at-most four sub-arrays are evaluated. Evaluation could mean that a recursive call is made to the next level, or the value stored in \( H \) for the sub-array is returned. There are at-most \( O(\log n) \) levels and hence the time to calculate the minimum in the range \((L_i, R_i)\) is \( O(\log n) \). The overall run-time for \( n \) such intervals would be \( O(n \log n) \).
Solution 2:
First, initialize a hash table $H_1$ to hold for each index $k$ of array, the list of clients whose range of dates either begin or end in $k$. Also initialize another hash table $H_2$ to hold the best date for each client.

Algorithm:
We define sub-problems on sub-arrays $A[x:y]$ as assigning cheapest day to each client $i$ whose range is entirely contained in $x:y$ i.e. $x \leq L(i) \leq R(i) \leq y$, where $L(i)$ is the first day in client $i$’s date range and $R(i)$ is the last day in their date range. Clearly, our problem is one of the sub-problems with $x = 0, y = n - 1$.

In each iteration of the algorithm, we split each sub-problem into two halves $A[x: (x+y)/2]$ (the left half) and $A[(x+y)/2 : y]$ (the right half). For every client $i$ with date range from $L(i)$ to $R(i)$, there are three possibilities: either $L(i)$ and $R(i)$ are both within the first half, $L(i)$ and $R(i)$ are both within the second half, or $L(i)$ lies in the first half and $R(i)$ lies in the second half. The first two cases can be solved through recursion, but we will need to handle the third case.

First, we compute solution for clients $i$ whose range has one end in left half and the other end in right half i.e. $x \leq L(i) \leq (x+y)/2$ and $(x+y)/2 \leq R(i) \leq y$. Let $cross(x,y)$ be the list of such intervals. To compute these solutions, We start at the middle index $(x+y)/2$ and traverse to the left, keeping track of the minimum, until we reach $x$. In this process, whenever we encounter a starting point $L(i)$, we first check whether the corresponding interval is $cross(x,y)$, i.e. we check if $R(i) \geq (x+y)/2$ and $R(i) \leq y$ and if it is, we update the minimum corresponding to that interval in $H_2$. We then apply the same method vice-versa traversing to the right. Once we are done, we remove all these clients from $H_1$ because we are essentially done computing their best dates.

Next, we recurse on the subarrays $A[x: (x+y)/2], A[(x+y)/2 : y]$.

At the end, we have $H_2$ storing the best date for each client.

Pseudocode for this is below:

MinDates(A): // A is array holding cheapest trips for each day
    H1 <- Hashmap with entries for each day, holding all
    clients whose intervals either start or end at each day k
    H2 <- Hashmap()
    Compute(A, 0, n-1)
    Return H2

Compute(A, x, y):
    if H1 is empty, return
    mid <- (x+y) / 2
    // compute min for all clients with intervals that cross
    for i from mid to x:
        min_so_far = min(min_so_far, A[i])
        if we encounter starting point of cross interval:
            let c be the corresponding client
            H2[c] <- min(H2[c], min_so_far)
    repeat process for i from mid to y
    remove all cross intervals from H1
    // recursively compute mins for each half
    Compute(A, x, mid)
    Compute(A, mid, y)

Correctness:
Each client is assigned the best date by the algorithm:
Consider a client $i$. First, note that once we remove the endpoints $L(i), R(i)$ from $H_1$, the corresponding solution $H2[i]$ will no longer be modified by subsequent recursive calls. This is because we need $L(i)$ and
$R(i)$ to be in $H1$ in order for $H2[i]$ to be updated, otherwise $H2[i]$ remains unchanged. Next, note that when we update the solution $H2[i]$ for client $i$, it will be a cross interval: in other words, it will be part of a recursive call such that the middle index $(x + y)/2$ is larger than $L(i)$ and smaller than $R(i)$. Since our algorithm will start from the middle and travel to the left until $L(i)$ is reached, and keep track of the minimum found along the way, by the time we reach $L(i)$ we will have the correct minimum. We will then see that $R(i) \geq (x + y)/2$, so will update the minimum correctly in $H2$. A similar argument applies to the right, and we can conclude that the value the algorithm finds the correct best date.

The best date that the algorithm finds for each client is a valid date:
Consider a client $i$. We will show that the value stored in $H2[i]$ corresponds to a valid date that is between $L(i)$ and $R(i)$. $H2[i]$ is only updated if the interval is a cross interval, and as part of a recursive call such that the middle index $(x + y)/2$ is larger than $L(i)$ and smaller than $R(i)$. When calculating the minimum, we look at all days left of the middle index until the day of $L(i)$ is reached, then all days right of the middle index until the day of $R(i)$ is reached. So, after this step is completed, when we update the value for client $i$, we have looked at all possible dates between $L(i)$ and $R(i)$, and no dates outside of this interval. We can therefore conclude that the value in $H2[i]$ is a day between $L(i)$ and $R(i)$, and so it is a valid date.

Runtime:
Each problem $T(n)$ divides into two sub-problems of size $T(n/2)$. When we are computing the solution of $cross(x,y)$ for a sub-problem from $x : y$ where $|y - x| = n$, we traverse left and right while updating the minimum.

Now, we claim that at each level of recursion in the recursion tree, i.e. say at level $k + 1$, with $2^k$ sub-problems, all these $2^k$ sub-problems together make at most $O(n)$ checks for cross, at most $O(n)$ updates to $H2$, at most $O(n)$ queries on the $H1$. This can be seen by the fact that at each recursion depth $k + 1$, our sub-problems are disjoint i.e. the sub-problems deal with disjoint parts of $A$ $[0 : n/2^k], A[n/2^k : 2 \cdot n/2^k], ...$.

For each client $i$, the left end of range is contained in exactly one of these sub-problems, and the right end of range is contained in exactly one of these sub-problems. So, all these sub-problems together look at each client $i$ at most twice at recursion depth $k + 1$. Now that querying from hashmap, updating hashmap, checking for cross are all $O(1)$ operations

Since the depth of the recursion tree is $\log(n)$, the total run-time is $O(n \log n)$

Question 3 (Guessing Game, 40 points)

Mary and Duncan are playing a game. Mary secretly picks two numbers from \{1, 2, ..., $n$\}. Duncan can ask yes/no questions about the numbers and Mary will respond with an answer that is correct for at least one of her two secret numbers. For example, if Mary’s numbers were 11 and 25 and Duncan asks if her number was more than 20, she could say “yes” because 25 is more than 20 or could say “no” because 11 is not more than 20. However, if Duncan asked if here number was even, she would be forced to say “no” as neither of her numbers are even. Duncan’s goal here is to find a list of at most two numbers that is guaranteed to contain at least one of Mary’s secret numbers (you can show that it is impossible for him to find both of her numbers, or narrow the search to a single possibility).

(a) Show that if Duncan knows two disjoint sets $S$ and $T$ where one of Mary’s numbers is known to be in $S$ and the other is known to be in $T$, that he can ask a constant number of questions which are guaranteed to cut the number of possibilities for at least one of her numbers in half. [15 points]

(b) Show that if Duncan knows a set $S$ of size at least 3 where one of Mary’s numbers is guaranteed to be in $S$, he can ask a constant number of questions and either deduce a smaller set $S'$ that must contain one of Mary’s numbers, or conclude that one of Mary’s numbers is in $S$ and the other is not in $S$. [15 points]

(c) Combine the above to show that Duncan can complete his goal with $O(\log(n))$ questions. [10 points]

Solution.

(a) Let \{s_1, ..., s_m\} be the elements of set $S$, and \{t_1, ..., t_n\} be the elements of set $T$. Split the set $S$ into two sets $S_1$ and $S_2$, where $S_1 = \{s_1, ..., s_{n/2}\}$ and $S_2 = \{s_{n/2+1}, ..., s_m\}$. Similarly, split the set $T$ into two sets $T_1 = \{t_1, ..., t_{n/2}\}$ and $T_2 = \{t_{n/2+1}, ..., t_n\}$. Duncan can then ask these four questions:
1. Is a number in \( S_1 \cup T_1 \)?
2. Is a number in \( S_1 \cup T_2 \)?
3. Is a number in \( S_2 \cup T_1 \)?
4. Is a number in \( S_2 \cup T_2 \)?

We will now show that Duncan will always be able to cut the number of possibilities for at least one number in half, regardless of what Mary’s answers are to the questions.

First, notice that Mary cannot answer \texttt{YES} to all four questions, or \texttt{NO} to all four questions. Since there are only two numbers and we know one is in \( S \) and one is in \( T \), this means that one number must be in either \( S_1 \) or \( S_2 \) (not both), and the other must be in either \( T_1 \) or \( T_2 \). This means that at least one of the four unions listed above cannot contain either of the numbers, which will force Mary to answer \texttt{NO} to at least one of the questions. But it also means that at least one of the four unions must contain both of the numbers, which will force Mary to answer \texttt{YES} to at least one of the questions.

So, Mary will answer \texttt{YES} to exactly 1, 2, or 3 of the questions. If she answers \texttt{YES} to one of the questions, then the union in the question she answered \texttt{YES} to will contain both of the numbers, which cuts the possibilities for both numbers in half. If she answers \texttt{YES} to three of the questions, then the union in the question she answered \texttt{NO} to will not contain either of the numbers, which also cuts the possibilities for both numbers in half.

If she answers \texttt{YES} to two of the questions, they cannot be questions 1 and 4, or questions 2 and 3. If they were, then since one number must be in either \( S_1 \) or \( S_2 \), and the other number must be in either \( T_1 \) or \( T_2 \), there will be another question whose union must contain both of the numbers, forcing her to answer \texttt{YES} to a third question. So, the two questions Mary answers \texttt{YES} to must have a nonempty intersection \((S_1, S_2, T_1, T_2)\). This common set will contain one of the numbers, which will cut the possibilities for that number in half.

(b) We will split the set \( S \) into roughly equal thirds. If \( S = \{s_1, \ldots, s_m\} \), then define \( S_1 = \{s_1, \ldots, s_{m/3}\} \), \( S_2 = \{s_{m/3+1}, \ldots, s_{2m/3}\} \), and \( S_3 = \{s_{2m/3+1}, \ldots, s_m\} \). Duncan can then ask these three questions:

1. Is a number in \( S_1 \cup S_2 \)?
2. Is a number in \( S_1 \cup S_3 \)?
3. Is a number in \( S_2 \cup S_3 \)?

We will now show that Duncan will always be able to either deduce a smaller set that contains one of Mary’s numbers, or conclude that one of Mary’s numbers is in \( S \) and the other is not in \( S \), regardless of what Mary’s answers are to the questions.

If Mary answers \texttt{YES} to any of the questions, then we have deduced a smaller subset \( S' \) of \( S \) that contains one of Mary’s numbers (since all of the unions in the questions have size smaller than \( S \)).

If one of Mary’s numbers is in \( S \), that number must be in either \( S_1 \), or \( S_2 \), or \( S_3 \). If Mary answers \texttt{NO} to all three questions, then we know that it is impossible for both numbers to be in \( S_1 \cup S_2, S_1 \cup S_3, \text{ or } S_2 \cup S_3 \). But if both of Mary’s numbers were in \( S \), either both numbers would be in the same subset \( S_i \), or one would be in \( S_i \) and the other would be in \( S_j \) \((i, j \in \{1, 2, 3\}, i \neq j)\). Since the questions contain all possibilities for \( S_1 \cup S_j \), Mary would be forced to answer \texttt{YES} to at least one of the questions. So we can conclude that one of Mary’s numbers is not in \( S \).

(c) **Algorithm.** We will use parts a) and b) to provide an algorithm for the guessing game. Given that we are trying to guess two numbers in the range \( \{1, \ldots, n\} \), our algorithm will first repeatedly ask part b)’s questions until one of the secret numbers is in \( S \) and one is not in \( S \), then will repeatedly ask part a)’s questions. We stop when our set containing a secret number is small enough to return.

More specifically, we define a function \texttt{Solve} that takes as input a set \( S \) and asks part b)’s questions on \( S \). Eventually, one of the secret numbers will be in \( S \) and the other will not be in \( S \). \( S \) and the complement of \( S \) are two disjoint sets, so we then use the function \texttt{PairSolve}, which takes two disjoint sets as input and asks part a)’s questions. We guess the numbers by calling \texttt{Solve}({1, \ldots, n}).
PairSolve(S,T):
If |S| < 3 return S, or if |T| < 3 return T (we have a list of at most two numbers guaranteed to contain at least one secret number)
Ask the questions in part a)
S' <- updated set S after asking part a’s questions
T' <- updated set T after asking part a’s questions
Return PairSolve(S', T')

Solve(S):
If |S| < 3, return S (we have a list of at most two numbers guaranteed to contain at least one secret number)
Ask the questions in part b)
If there is a smaller set S' that contains one of the numbers:
Return Solve(S')
If one of the numbers is in S and the other is not in S:
Return PairSolve(S, complement of S)

Correctness. We will prove this algorithm is correct by induction on n.
Base case: n = 3. Our algorithm will first ask the questions in part b), and since part b) is correct, it will find a smaller set S' that contains one of the numbers. Since we must have |S'| < 3, we are done.
Inductive step. Assume that the algorithm is correct for all sets of size up to n − 1. If Mary picks two numbers from {1,...,n}, then our algorithm will first ask the questions in part b) and get a smaller set S' that contains one of the numbers. By the induction hypothesis Solve(S') is correct, which means that eventually, we will end up with a set A such that one of the numbers is in A and the other is not in A. Our algorithm then calls PairSolve(A, {1,...,n} - A). Since both arguments are of size < n, by the induction hypothesis PairSolve is correct so our algorithm will find a set of at most two numbers containing a secret number.

Runtime. We know that part a) and part b) each take a constant number of questions, so asking the questions in part a) or part b) once takes O(1) time.
For PairSolve: We know that part a)’s questions either cut the size of S in half, or cut the size of T in half. This means that after an iteration of PairSolve we will have either |S'| = |S|/2 or |T'| = |T|/2, so the product of sizes |S'||T'| ≤ (|S||T|)/2. Define m = |S||T|. m is halved with every iteration, so PairSolve has recurrence relation \( T(m) = T(m/2) + O(1) \). This is \( O(\log m) \) by the Master Theorem, which is also \( O(\log n) \).
For Solve: We know that in the worst case, part b)’s questions result in a smaller set S' that has a size of 2|S|/3. Notice that every iteration of Solve either recursively calls Solve on S', or calls PairSolve. In the worst case, the maximum number of iterations that we can call Solve before we are either done or we call PairSolve is represented by the recurrence relation \( T(n) = T(2n/3) + O(1) \), which is \( O(\log n) \) using the Master Theorem.
Since our algorithm will in the worst case call Solve for \( O(\log n) \) iterations before calling PairSolve for \( O(\log n) \) iterations, the overall runtime is \( O(\log n) \).