In order to prepare this submission, the following resource(s) has been used:

- Discussion with fellow students: *Victor Gan*, *Sarah Huber*, and *Evan Peng*.

1. We noted in class that $n-1$ comparisons are necessary and sufficient, in the worst case, to determine the maximum (or the minimum) element of a set of n numbers.

   (a) Show that if we want to determine both the maximum and the minimum element, it suffices to use $\lceil 3n/2 \rceil - 2$ comparisons.
   Proof. The idea is to separate these numbers into two groups, and then use the assumption in the problem statement to prove the point.
   Let's say we have two empty sets A and B, and all the numbers are in the set S. Each time, we pick two numbers from S and compare them. The bigger number is put in set A and the smaller number is put in set B. After $\lfloor n/2 \rfloor$ comparison, we have $|A| = |B| = \lfloor n/2 \rfloor$. If n is odd, then we will have one number left in S, so we just put the last number in A. At the end $|A| = \lceil n/2 \rceil$ and $|B| = \lfloor n/2 \rfloor$. Given that the maximum number will always be bigger in comparisons, and minimum number will always be smaller in comparisons, then the maximum number will be in A and minimum number will be in B. So the new problem is to find the maximum in A and minimum in B, which based on the assumption in the problem statement will require $\lceil n/2 \rceil - 1$ and $\lfloor n/2 \rfloor - 1$ respectively. Hence, after $\lfloor n/2 \rfloor + \lceil n/2 \rceil - 1 + \lfloor n/2 \rfloor - 1 = \lceil 3n/2 \rceil - 2$ we can identify the minimum and maximum.

   (b) We want to devise an adversary strategy to show that $\lceil 3n/2 \rceil - 2$ comparisons are also necessary, in the worst case.
   Show that, at any stage, an algorithm playing against this adversary is forced to make at least $f(A, B, C) = \lceil 3|A|/2 \rceil + |B| + |C| - 2$ more comparisons before it has determined the maximum and the minimum.
   Proof. Let the triplet $(X, Y, Z) = (|A|, |B|, |C|)$ denote the initial state of the problem. We claim that the answer will be known only when you're at the final state $(0, 1, 1)$, which is equivalent to $|A| = 0, |B| = 1, |C| = 1$ meaning that only one number is candidate for max and min. We show that we need at least $\lceil 3|A|/2 \rceil + |B| + |C| - 2$ steps to reach the final state.
   Given that the group D has both won and lost, we're no longer interested in them because they contain neither min or max. Let XY denote performing a binary test $<$ test on an element of X and an element of Y. All possible comparison and their respective results in state transition can be summarized as below:

      i. $AA : (X, Y, Z) \rightarrow (X- = 2, Y++, Z++)$
      ii. $AB : (X, Y, Z) \rightarrow (X--, Y, Z++)$
      iii. $AC : (X, Y, Z) \rightarrow (X--, Y++, Z)$
      iv. $BB : (X, Y, Z) \rightarrow (X, Y--, Z)$
      v. $BC : (X, Y, Z) \rightarrow (X, Y, Z)$
      vi. $CC : (X, Y, Z) \rightarrow (X, Y, Z--)$

   For example, AC means comparing $a \in A$ with $c \in C$. Given the descriptions, the outcome will always be $a > c$. So a will be put in B, resulting in transition $(X, Y, Z) \rightarrow (X--, Y++, Z)$. Notably, the comparison BC has no effect in transition, so we don't consider this

transition from now.
The goal is to start from $(|A|, |B|, |C|)$ and go to the final state $(0, 1, 1)$ using minimum number of transitions above, and this is equivalent to the original problem. After investigating the dependency between possible choices and their respective results, one can easily observe that the order of transitions has no impact on the minimum number required to reach the final state. Only transitions AA, AB, AC can help us go from $|A|$ to 0, and AA reducing X by 2 is the obviously faster choice. After doing $\lfloor |A|/2 \rfloor$ of AA, depending on $|A|$ being odd or even, we're either at $X = 1$ or $X = 0$. If we're at $X = 1$, we'd do the comparison AB (or AC, doesn't really make a difference). So after $\lceil |A|/2 \rceil$, we've reached the state $(0, |B| + \lfloor |A|/2 \rfloor, |C| + \lceil |A|/2 \rceil)$. Now to reach $Y = 1$ and $Z = 1$ ($X = 0$, thus this will be the final state), we need at least $|B| + \lfloor |A|/2 \rfloor - 1$ of BB and $|C| + \lceil |A|/2 \rceil - 1$ of CC. So after at least $\lceil |A|/2 \rceil + |B| + \lfloor |A|/2 \rfloor - 1 + |C| + \lceil |A|/2 \rceil - 1 = \lceil 3|A|/2 \rceil + |B| + |C| - 2$, we can determine the minimum and maximum number.

(c) Argue that the lower bound ($\lceil 3n/2 \rceil - 2$) follows from part (b).
Proof. Using the same notation from previous proof, we argue that the minimum number of transitions from start state $(|A|, |B|, |C|)$ to $(0, 1, 1)$ can't be longer than the ($\lceil 3n/2 \rceil - 2$), thus proving a lower bound in the worst case scenario.
As we've seen before, comparison BC has no effect. Furthermore, CC, BB, and AA will have the same result irrespective of the adversary strategy, because there's always one loser and one winner in test $<$ and being in group B, C will always reduce the number of elements by one, and being in A will always reduce the number of elements in A by two and increase number of elements in B and C by one. So adversary, having in mind that we must take as much steps as possible in the path to the final state, needs to decide for the comparisons AB and AC. The argument we're about to discuss for AB applies in the same way for AC. The assumption for the previous problem was that the elements in B always win the $>$ test. This will make the transition $(X, Y, Z) \to (X--, Y, Z++)$. What's the alternative? If the adversary decides not to follow this rule, and allows the $b \in B$ to lose; b will be joining useless group D, and $a \in A$ will be replacing b in B. The transition will be $(X, Y, Z) \to (X--, Y, Z)$. This will be in favor of us, because we wouldn't have to decrease Z in order to reach $Z = 1$ in contrast to the original strategy, and will reduce the minimum number of steps by one. The same argument goes for AC. Thus adversary being interested in making the number of transitions as much as possible, will never break the rules from previous question. Thus the upper bound number of minimum steps required to reach the final state will remain to be ($\lceil 3n/2 \rceil - 2$), and this will be the lower bound on minimum number of comparisons required to determine min and max.

2. Suppose we are given a set S containing n elements drawn from the universe $U = \{0, 1, ..., m-1\}$, and suppose that we are only able to make comparisons of the form $x_i \geq c$, where x_i is an element of S and c is some specified element of U (for example $x_5 \geq 17$).

(a) Show that $O(nlg|U|)$ such unary predicate evaluations suffice to determine the maximum element of S, in the worst case.
Proof. The idea is to reduce the number of elements that can be candidate for the maximum position through binary search separation.
Let C denote the set of maximum candidates, and intitially set $C = S$. Let the range $[L, R)$

denote the range in U that holds the maximum element, and initially set $[L, R) = [0, m)$. Each time, we compare $(R - L)/2 + L$ against all the elements in C and put the lesser elements in C_1 and greater elements in C_2. When $|C_2| = 1$, we have identified the maximum element. If $|C_2| = 0$, $C = C_1$, else $C = C_2$ for the next iteration. We also update the $[L, R)$ accordingly. Roughly, in the worst case $|C|$ remains to be n until the very last end making us compare them all with at most $lg(U)$ elements, resulting in asymptotic complexity $O(nlg|U|)$ evaluations.

(b) Give the best (largest) lower bound that you can (expressed as a function of both n and $|U|$) on the worst-case number of unary predicate evaluations needed to determine the maximum element of S.

Proof. The key observation is the fact that if the maximum candidate set C with members in range R ($C \subseteq R$), has more than 1 members, the only condition in which you can determine the max member of S is when R has only 1 member. In other words if $|C| = 1$ you have determined the max member; and if $|C| \neq 1$ the only way you can say you have found max member is when the range of candidate members has only one number, which happens when there are more than one copies of the max member in S. Moreover, having n elements, means there are U^n combinations of S. In each comparison, you can only cut the range of each element in half, leading to a reduction in size of space by 2. If it's the case that all the elements are the same, in order to be able to identify the maximum member, you have to reduce the range of maximum number to only 1 number, because the candidate set C will always have more than one members. This is equivalent to reducing the size of space to 1 possible combination, which requires at least $lg(U^n) = nlg(U)$ comparisons.

3. Suppose that we are given $D[1 : n]$, an array of real-valued keys sorted in increasing order, and we wish to use D as a dictionary, i.e. for each x_j in a sequence $x_1, x_2, \ldots, x_m$ of queries we want to return the location $l(j) \in \{1, \ldots, n\}$ of an element in $D[1 : n]$ that minimizes $|D[l(j)] - x_j|$ (in which case we say that key $D[l(j)]$ has been accessed.)

(a) Design an algorithm for searching array $D[1 : n]$ that exploits locality of reference. Specifically, show that query x_j in the sequence $x_1, x_2, \ldots, x_n$ can be answered at a cost proportional to $1 + lg(1 + \Delta j)$, for all $j > 1$.

Answer. (i) We first compare $D[l(j-1)]$ with x_j, if x_j was greater we'd go right, else we'd go left. Without losing generality, let's assume we decided to go right. Let $s = 1, c = l(j-1)$ initially, (ii) on each step we compare $D[c + s]$ with x_j, if x_j was less, we stop; otherwise we set $c+ = s, s* = 2$ and retry. After n tries, we have the distance $2^n - 1$ from $l(j-1)$. Hence, we will stop after $\lceil lg(\Delta_j + 1)\rceil$ tries. At this point, x_j is within the range of last step s which is 2^{n-1}. (iii) If we do a binary search within this region, it'll require $lg(2^{n-1}) + 1$ comparisons to find $l(j)$ (added +1 is because we need to locate the closest number, thus once we have found a number, we have to compare it against the last node in the binary tree again). Hence we have, $\lceil lg(\Delta_j + 1)\rceil - 1 + 1$. Adding (i, ii, iii), leads us to $2\lceil lg(\Delta_j + 1)\rceil + 1$.

(b) Let A be any comparison-based algorithm for searching $D[1 : n]$. Prove that for every $k, 1 \leq k \leq lgn$, there exists an index $i_k \in [l(j-1) - 2^k, l(j-1) + 2^k]$ such that A requires at least k comparisons to search for x_j, knowing $l(j-1)$, when it turns out that $l(j) = i_k$. [Hint: recall the fact that every binary tree has at most 2^d leaves at depth d, for every $d \geq 0$.]

Answer. Every binary tree with depth d has $2^{d+1} - 1$ nodes. Furthermore, i_k could be any of $2^{k+1}+1$ numbers in the range $[l(j-1)-2^k, l(j-1)+2^k]$. So, any comparison based algorithm trying to find i_k after d_{th} comparisons has gone over a tree with $2^{d+1}-1$ numbers. So the minimum number of d such that $2^{d+1}-1 \geq 2^{k+1}+1$ will be the answer to the minimum number of comparisons, which is k.

4. In class we discussed x-fast tries ...

   (a) Show that x-fast tries are particularly efficient in handling predecessor and successor queries when the query itself is an element of S.
   Answer. When the query itself is an element of S, you can access it in $\theta(1)$ and since it includes a link to its successor and predecessor, you can access them in $\theta(1)$ too! Taking $\theta(1)$ in total, how much faster you want it to be?

   (b) Prove that if none of ancestor(x, h), left-neighbour(ancestor(x, h)), or right-neighbour(ancestor(x, h)) are marked then $\Delta > 2^h$.
   Answer. Ancestor(x, h) = $\lfloor \frac{x}{2^h} \rfloor$ and the area below is $[2^h\lfloor \frac{x}{2^h} \rfloor, 2^h\lfloor \frac{x}{2^h} \rfloor + 2^h - 1]$ which has the length 2^h. If neither left-neighbour(ancestor(x, h)) or right-neighbour(ancestor(x, h)) are marked, it means there are no values in range $[2^h\lfloor \frac{x}{2^h} \rfloor - 2^h, 2^h\lfloor \frac{x}{2^h} \rfloor - 1]$ or $[2^h\lfloor \frac{x}{2^h} \rfloor + 2^h, 2^h\lfloor \frac{x}{2^h} \rfloor + 2^{h+1} - 1]$ respectively, which means $\Delta > 2^h$.

   (c) Prove that if ancestor(x, h) is unmarked but at least one of left-neighbour(ancestor(x, h)) or right-neighbour(ancestor(x, h)) is marked then we can find the predecessor and successor of x, using information stored at these marked nodes and at the leaves, in O(1) time.
   Answer. If the left-neighbour(ancestor(x, h)) is marked, then the predecessor will be the max value pointer stored at that node, and the successor will be the successor stored in the max value pointer. If the right-neighbour(ancestor(x, h)) is marked, then the successor will be the min value pointer stored at that node, and the predecessor will be predecessor stored in the min value pointer.

   (d) Prove that if ancestor(x, h) is marked then we can find the predecessor and successor of x, by searching for the lowest marked node on the path from x to ancestor(x,h), in $O(lgh)$ time.
   Answer. If ancestor(x, h) is marked, it means for all $h' > h$, ancestor(x, h') will be also marked. Using this attribute, we can perform a binary search within the range $[1, h]$ in order to find the lowest marked ancestor. Since this will be the lowset marked ancestor, it means the child node which includes x is not marked, thus the other child must be marked, which means the condition for the previous part is met, so we can access the predecessor and successor in $O(1)$ time.

   (e) Using the results from parts (b), (c) and (d) above, describe how to find the predecessor and successor of x in an x-fast trie in $O(lglg\Delta)$ time, when $\Delta \geq 4$.
   Answer. When we locate the lowest marked node on the path from x to root, as previously shown, the predecessor and successor can be identified in $O(1)$. Hence, it suffices to show that the lowest marked node can be found in $O(lglg\Delta)$. Similar to the approach in problem 3, on iteration i, we'll check the condition (b) with $h = 2^i - 1$. When the condition is met, $i = lg(\Delta)$. Now using the part (d) and the fact that $h = lg(\Delta)$, we can find the predecessor and successor in $O(lglg\Delta)$.