CSC265 Homework 8 | Fall 2021
Student 1: Haojun Qiu | Student Number: 1006900622
Student 2: Runshi Yang | Student Number: 1006734223

Q1

(a) Proof:

- Let $x_1, \cdots, x_n$ be an input sequence of n distinct numbers.
- Assume for a contradiction that there is a pair of adjacent elements in sorted order that has not been compared in the on the path from the root to a leaf taken by this input sequence, say elements x_i, and x_j.
- Since $x_1, \cdots, x_n$ are distinct numbers, by the correctness of the comparison tree, this input sequence will take the path from the root to a leaf with value **FALSE**, referring that there is no repetition in this sequence.
- Now let $x_1, \cdots, x_{j-1}, x_i, x_{j+1}, \cdots, x_n$ be another sequence. (Replace x_j by x_i in the original sequence) Since x_j is replaced by value x_i, this new sequence contains a repetition (two values of x_i). We know by the correctness of comparison tree that this new sequence will take a path from the root to a leaf with value **TRUE**, referring that this sequence contains a repetition.
- Since x_i and x_j are adjacent elements in the sorted order in the original sequence, we know that for all $x \in \{x_1, \cdots, x_n\} \setminus \{x_i\}$, if $x < x_j$, then $x < x_i$ and if $x > x_j$, then $x > x_i$.
- By our assumption, there is no comparison between x_i and x_j on the path from the root to a leaf taken by the input sequence, and we have shown that if we replace x_j by x_i and does not compare x_j with x_i, the results of the comparisons are the same as the results without replacing x_j with x_i, so we can conclude that the new sequence will take the same path from the root to the leaf with value **FALSE** as the original sequence did.
- Thus this is a contradiction with the correctness of the comparison tree, so our assumption must be false, which means there must be a comparison between each pair of adjacent elements in the sorted order.

(b) Proof:

- Define question P' : sorting a sequence of n distinct numbers.

  Input: A sequence of n distinct numbers $x_1, \ldots, x_n$.

  Output: A permutation π of $\{1, \ldots, n\}$ such that $x_{\pi(i)} < x_{\pi(i+1)}$ for $i = 1, \ldots, n-1$.
- Trim the 3-way comparison tree into a binary tree by deleting all nodes connected by the "=" comparison and their subtrees. Then I will show that we can relabel the remaining leaves to permutations of $\{1, \ldots, n\}$ to solve problem P'.

  Since the numbers in the input sequence of P' are distinct, using comparison tree solving problem P to process this sequence will not pass through any node executing equaling comparison. So there is no effect on solving P' if we delete all the nodes connected by the "=" comparison and their subtrees.

  In Q1 (a) we have shown that on the path from the root to a leaf taken by an input sequence of n distinct numbers, there must be a comparison between each pair of adjacent elements in the sorted order. Knowing the result of these comparisons, we can easily relabel the leave on the end of that path such that the relabeled output is sorted in the correct order. (If we know $x_1 < x_3$, $x_3 < x_2$, then we can easily know that the permutation x_1, x_3, x_2 is the correct output)
- Since the tree after trimming solves problem P' as shown above, and that there are $n!$ different permutations of $\{1, \ldots, n\}$ as different possible outcomes of problem P', the height of the tree after trimming is at least $\lceil \log_2 n! \rceil$.

- The height of the tree after trimming is no greater than the original tree since we only delete but not add nodes. So the height of the original tree is at least $\lceil \log_2 n! \rceil$. And since $\lceil \log_2 n! \rceil \geq \log_2 n!$, the original comparison tree has height at least $\log_2 n!$.

Q2

Proof:

Lower Bound on C(P): Problem P: Determine whether every input sequence $x_1, x_2, \cdots, x_n$ of n numbers contains a repetition.

Problem P': Determine a mode of a sequence of n numbers.

Algorithm A: Does nothing.

Algorithm B: Delete a mode element from the input array, and then search for the element in the input array, output true (successful search) or false (unsuccessful search) as output of problem P. In code, this is

```
1  DELETE(X, y')
2  return SEARCH(X, y')
```

where X is the input sequence $x_1, x_2, \ldots, x_n$ of problem P, and y' is the output of problem P'.

Proof of Reduction Correctness: We want to show that problem P reduces to P' by algorithm A and B.

Fix an arbitrary sequence $x_1, \ldots, x_n$ of numbers to be input for problem P, and y' to be the output of problem P'. Then, notice that A does nothing, and problem P' needs a sequence of numbers as input, hence A does the right form of (identical/trivial) mapping. The algorithm B maps from output y' of P' with input sequence $x_1, \ldots, x_n$, to a boolean as output for problem P, so it has correct form of mapping as well.

Next, assume y' is the mode element of the sequence $x_1, x_2, \ldots, x_n$ (i.e., the output y' of P' is correct). Then, it suffices to show that the output boolean for algorithm B (with input being y' as well as the input sequence for P), is a correct output for problem P (i.e., determining whether there is a repetition in the sequence $x_0, x_1, \ldots, x_n$).

- If Algorithm B returns TRUE at line 2. Then, since we have deleted the mode y' from the sequence once (in line 1 of algorithm B), the following search still finds y' indicates that there are at least 2 elements equal to y' in the original input sequence $x_1, \ldots, x_n$. Thus, the output of algorithm B (TRUE) is a correct output of problem P.
- If Algorithm B returns FALSE at line 2. Then, since we have deleted the node y' from the sequence once, the following search does not find y' again indicates that there is only 1 occurrence of y' in the original sequence. Since y' is the mode of the original sequence, we know that no other elements in the original sequence have its occurrence more than once. This is saying that the original input sequence contains no repetition, hence the output of algorithm B (FALSE), is a correct output for problem P.

■

Then, notice that algorithm A and B both takes $O(n)$, which is $o(n \log n)$. By Q1, we have $C(P) \geq \log_2(n!) \in \Omega(n \log n)$, that is, $C(P) \in \Omega(n \log n)$. Therefore, we can conclude that $C(P') \in \Omega(n \log n)$.

Upper Bound on C(P):

Define an algorithm FIND-MODE(X) (where input X is sequence of numbers) with execution as follows:

```
1  Convert the sequence into an array, by traversing.
2  Sort the array using QUICKSORT.
3  Traverse through the sorted array and create a linked list, with each nodes' key represent a number in the
   sequence and attribute count as the number of occurance of its key in the array.
4  Traversing through the linked list, during which, keep track of the key k in the node with max count so far.
5  Return k.
```

For any input sequence of size n, line 1 takes at most $O(n)$, and line 2 takes at most $O(n \log n)$. Line 3 takes at most $O(n)$ since the created linked list length is at most the array's length, and line 4 takes at most $O(n)$. Hence FIND-MODE(X) that finds the mode of a sequence, in worst case, takes $O(n \log n)$. Therefore, $C(P) \in O(n \log n)$.

Since $C(P) \in O(n \log n)$ and $C(P) \in \Omega(n \log n)$, we have $C(P) \in \Theta(n \log n)$.

Q3

(a) Proof:

Let $n = 1$. Assume there is an algorithm correctly solves the problem by not reading both bits. Define our adversary strategy as follows:

- If $A[1]$ is read, we answer $A[1] = 0$, assign this value accordingly.
- If $B[1]$ is read, we answer $B[1] = 0$, assign this value accordingly.

Then, since only 1 read is made, there is at least 1 bit is not read, say $A[1]$ (the argument follows by symmetry if $B[1]$ is not read) is not read.
Then, if the algorithm returns that A contains more 1's, we assign $A[1] = 0$, and all the not assigned slot (possibly $B[1]$ is not read as well) to 0.
If the algorithm returns that A and B contains same number of 1's. We assign $A[1] = 1$, and all the not assigned slot (possibly $B[1]$ is not read as well) to 0.
If the algorithm returns that B contains more 1's. We assign $A[1] = 0$, and all the not assigned slot (possibly $B[1]$ is not read as well) to 0.

This input $A[1], B[1]$ is consistent with all the answers the adversary has given. However, the algorithm gives the wrong answer on this input, which contradicts with the correctness of the algorithm.

So, in the worst case, any algorithm correctly solves the problem when $n = 1$, requires to read both bits.

Let $n = 2$. Assume there is an algorithm correctly solves the problem by not reading all four bits. Define our adversary strategy as follows:

- If $A[1]$ is read, we answer $A[1] = 0$, assign this value accordingly.
- If $A[2]$ is read, we answer $A[2] = 1$, assign this value accordingly.
- If $B[1]$ is read, we answer $B[1] = 0$, assign this value accordingly.
- If $B[2]$ is read, we answer $B[2] = 1$, assign this value accordingly.

Then, since at most 3 bits has been read, there is at least 1 bit is not read.

- **Case 1:** $A[1]$ is not read.

  If the algorithm returns that A contains more 1's **or** that B contains more 1's, then we assign $A[1] = 0$.
  If the algorithm returns that A and B contain the same number of 1's then we assign $A[1] = 1$.
  In both cases, we assign all the other unread bits (if not all other 3 bits have been read) as defined strategy.

- **Case 2:** $B[1]$ is not read.

  The strategy follows by symmetry from case 1 (swapping "A" and "B").

- **Case 3:** $A[2]$ is not read.

  If the algorithm returns that A contains more 1's **or** that A and B contains same numbers of 1's, then we assign $A[2] = 0$.
  If the algorithm returns that B contains more 1's, then we assign $A[2] = 1$.
  In both cases, we assign all the other unread bits (if not all other 3 bits have been read) as defined strategy.

- **Case 4:** $B[2]$ is not read.

  Then, the argument follows by symmetry to case 3.

This input of A and B is consistent with all the answers the adversary has given. However, the algorithm gives wrong answer on this input, in all cases. This contradicts with the correctness of the algorithm.

So, in the worst case, any algorithm correctly solves the problem when $n = 2$, requires to read all 4 bits.

(b) Lower Bound: Adversary Strategy: Given array $A[i..e]$ and $B[i..e]$. Let $n = e - i + 1$. When a read $A[j]$ (or $B[j]$) $(i \leq j \leq e)$ is made,

- if $j \geq \lfloor n/2 \rfloor + 1$, we answer that $A[j..e]$ and $B[j..e]$ are all 1's;
- if $j < \lfloor n/2 \rfloor$, we answer that $A[i..j]$ and $B[i..j]$ are all 0's.

Claim: Let $k \geq 0$. Any algorithm that solves this problem with $|A| = n \geq 2^k$, then you must read $\geq k+1$ bits.

Proof: We prove by induction on k.

Base Case: Let $k = 0$. If the $|A| = n \geq 2^0 = 1$, then, by part (1), you must reads at least $2 \geq 0 + 1$ bits.

Induction Step: Let $k \geq 0$. Assume that the claim holds for k. Then, consider a read from algorithm, say $A[j]$ or $B[j]$. Let's just suppose bit being read is $A[i]$ without lose.

Notice that as the strategy defined above, in all cases, the algorithm can reduce (recurse) to a problem with at least 2^{k-1} bits. In specific,

- If $j \geq \lfloor n/2 \rfloor + 1$. Then, $j - 1 \geq \lfloor n/2 \rfloor \geq \lfloor 2^k/2 \rfloor = \lfloor 2^{k-1} \rfloor = 2^{k-1}$. And in this case, with answer given by adversary, the program makes best outcome by recursing problem to $A[1..j-1]$ and $B[1..j-1]$ (finds which of $A[1..j-1]$ and $B[1..j-1]$ contains more 1's or conclude both contain same number of 1's), both of which has size $j - 1$, which is at least 2^{k-1} as shown above
- If $j < \lfloor n/2 \rfloor + 1$. Then, $-j + n > -\lfloor n/2 \rfloor - 1 + n$. Hence, $n - j \geq n - \lfloor n/2 \rfloor \geq 2^k - \lfloor 2^{k-1} \rfloor = 2^k - 2^{k-1} = 2^{k-1}$. And in this case, with answer given by adversary, the program makes best outcome by recursing problem to $A[j+1..n]$ and $B[j+1..n]$, both of which has size $n - j$, which is at least 2^{k-1} as shown above.

So, by the induction hypothesis, it must reads at least $k + 1$ more bits, so in total makes at least $k + 2$ bits as wanted. ■

We can therefore conclude that any algorithm solves this problem of size n (which reads at least $n = 2^{\log_2 n} \geq 2^{\lfloor \log_2 n \rfloor}$ bits of course), so by the claim, you must reads at least $\lfloor \log_2 n \rfloor + 1$ bits.

Upper bound:

We want to define an algorithm that solves the problem.

Step 1: Find the index of first occurrence of 1 in array $A[1..n]$ if it exists else raise an error.

Using idea of binary search. The code is:

```
1  FIND-FIRST-1-INDEX(A, i, e)
2  if i = e
3       if A[i] = 1
4            return i
5       else error
6  else
7       m = ⌈(i + e)/2⌉
8       if A[m] = 0
9            return FIND-FIRST-1-INDEX(A, m + 1, e)
10      else // A[m] = 1
11           if FIND-FIRST-1-INDEX(A, i, m − 1) raise error
12                return m
13           else
14                return FIND-FIRST-1-INDEX(A, i, m − 1)
```

We call FIND-FIRST-1-INDEX$(A, 1, n)$, which will returns the first occurrence of 1 in $A[1..n]$, error is raised if no occurrence of 1 in A.

Step 2: Determine the relationship of the number of 1's in A and B.
If FIND-FIRST-1-INDEX$(A, 1, n)$ raise an error
  If $B[n] = 1$, then we return that B contains more 1's
  Else $(B[n] = 0)$, we return that A and B contains same number of 1's

Else (FIND-FIRST-1-INDEX$(A, 1, n)$ returns a index, say j)
  If $B[j] = 0$, then we return that A contains more 1
  If $B[j] = 1$, then
    If $B[j-1]$ exists and equals 1, then we return that B contains more 1
    If $B[j-1]$ does not exists (index out of range) **OR** $B[j-1]$ exists and equals 0, then we return that A and B contains same number of 1.

Algorithm Correctness:

Step 1: If A only contains 1 element, then line 3, 4, 5 will check if it is 1 or not: if $A[i] = 1$ then i is index of first 1 in $A[i..e] = A[i]$, else there is no 1 in $A[i..e]$ so we should raise an error.

If A contains 2 or more elements, line 7 will find the "mid point" m of A. If $A[m]$ is 0, we know that $A[1 : m+1]$ are all 0's, so the recursive call on line 9 will find the first occurrence of 1 in $A[m+1 : n]$. If $A[m]$ is 1, we know that the first occurrence of 1 is in $A[1 : m-1]$ or is $A[m]$ itself. If the recursive call on line 11 find there is no occurrence of 1 in $A[1 : m-1]$, then $A[m]$ must be the first occurrence of 1, so m is returned line 12. Else, the recursive call on line 14 find the first occurrence of 1 in $A[1 : m-1]$ successfully, so it must be the first occurrence of 1 in A.

Step 2: If we found that there is no 1 in A, we just need to find out if there is 1 in B by reading the last bit of B. If it is 1, then B has more 1's; if it is 0, then there is also no 1 in B, so A and B contains the same number of 1's which is 0.

If we found the first occurrence of 1 in A which is $A[j]$, we know that there are $n - j + 1$ occurrence of 1 in A. If $B[j] = 0$, then we know that there are at most $n - j$ occurrence of 1 in B, which must be less than the number of 1's in A. If $B[j] = 1$ and $B[j-1] = 1$, then we know that there are at least $n - j + 1$ occurrence of 1 in B which must be greater than the number of 1's in A. If $B[j] = 1$ and $B[j-1] = 0$ **OR** $B[j = 1]$ and $B[j-1]$ does not exist, then we found that $B[j]$ is also the first occurrence of 1 in B, so the number of 1's in A and B must be equal. ■

Claim $P(n)$: For $n \geq 1$, step 1 of the above algorithm, calling FIND-FIRST-1-INDEX(A, i, e), if $n = e - i + 1$, then it reads at most $(\log_2 n) + 1$ bits in the worst case.

Proof:
Base Case: Let $k = 1$. The algorithm only reads $A[1]$, which is $1 \leq (\log_2 1) + 1$ bit. So $P(k)$ holds.

Induction Step: Let $k > 1$. Assume for all $j \in \mathbb{N}, 1 \leq j < k, P(j)$ holds. The algorithm reads $A[m]$ before doing recursion, $m = \lfloor \frac{i+e}{2} \rfloor$.

If $i+e$ is even, then $e-(m+1)+1 = e - \lfloor \frac{i+e}{2} \rfloor = \frac{e-i}{2} = \frac{k-1}{2} \leq \lfloor k/2 \rfloor$ and similarly $(m-1)-i+1 = \lfloor \frac{i+e}{2} \rfloor - i = \frac{i+e}{2} - i = \frac{e-i}{2} \leq \lfloor k/2 \rfloor$.

If $i+e$ is odd, then $e-(m+1)+1 = e - \lfloor \frac{i+e}{2} \rfloor = e - \frac{i+e-1}{2} = \frac{e-i+1}{2} = \frac{k}{2} \leq \lfloor k/2 \rfloor$ (since $i+e$ is odd, $e-i+1$ must be even) and similarly $(m-1)-i+1 = \lfloor \frac{i+e}{2} \rfloor - i = \frac{i+e-1}{2} - i = \frac{e-i-1}{2} = \frac{k}{2} - 1 \leq \lfloor k/2 \rfloor$.

So, we recurse on size at most $\lfloor k/2 \rfloor$. So, it makes at most $(\log_2 \lfloor k/2 \rfloor) + 1$ reads by the induction hypothesis. That is, the entire algorithm makes at most $\big((\log_2 \lfloor k/2 \rfloor) + 1\big) + 1 = \big(\log_2(\lfloor k/2 \rfloor \times 2)\big) + 1 \leq (\log_2 k) + 1$ (since $\lfloor k/2 \rfloor \times 2 \leq k$) bits, as wanted. ■

So step 1 calling FIND-FIRST-1-INDEX$(A, 1, n)$ reads at most $\log(n - 1 + 1) = \log n$ bits in the worst case.

Next, it is easy to observed that step 2 of the algorithm reads at most 2 bits in the worst case. Hence the entire algorithm reads at most $(\lceil \log_2 n \rceil + 1) + 2 = (\log_2 n) + 3$ bits.

Conclusion: Any algorithm that solves this problem must read at least $\lfloor \log_2 n \rfloor + 1$ bits in the worst case. And the algorithm we provide solves this problem, reads at most $(\log_2 n) + 3 = \lfloor \log_2 n \rfloor + 1 + O(1)$ bits.