Homework 2

ECE345 - Group 16

October 13th, 2023

Total pages: 21

Team Member	**Student Number**
Ardavan Alaei Fard	1007934620
Rowan Honeywell	1007972945
Isaac Muscat	1007897135

Contents

Question 1

(a)

Consider the pseudocode below:

```
FindExactChips(C, target)
1  HeapSort(C)
2  left = 1
3  right = C.length
4  while left < right
5      sum = C[left] + C[right]
6      if sum < target
7          left = left + 1
8      elseif sum > target
9          right = right - 1
10     else
11         return (C[left], C[right])
12 return (NIL, NIL)
```

This algorithm first sorts the array of chips using heapsort. Then, to find two chips which sum to the target, a *left* pointer is set to the left-most element of the sorted array and a *right* pointer is set to the right-most. If the sum of the chips is larger than the target value, then it must be decreased (decrement the *right* pointer). If the sum is smaller than the target value, it must be increased (increment the *left* pointer). Finally, if the sum is equal to the target value, then we can select (return) the two chips at the *left* and *right* indices.

For time complexity, the use of heapsort forces this algorithm to $\mathcal{O}(n \lg n)$ as this is heapsort's worst-case (and best-case) performance (CLRS, 3rd Edition, Exercise 6.4-4). Next, in the worst case (no chips whose values sum to the target), the **while** loop (lines 4 - 11) must iterate over each chip at most once. Therefore, this section of the algorithm is $\mathcal{O}(n)$.

To show that $\mathcal{O}(n) + \mathcal{O}(n \lg n) = \mathcal{O}(n \lg n)$, by definition (CLRS, 3rd Edition, Pg. 47):

$$n + n \lg n \leq cn \lg n$$

$$n + n \lg n \leq n \lg n + (c-1)n \lg n$$

Subtracting $n \lg n$ from both sides when $c = 2$:

$$n \leq n \lg n$$

Therefore the algorithm is $\mathcal{O}(n \lg n)$.

In terms of space, it is known that heapsort maintains an $\mathcal{O}(1)$ space complexity since it sorts in-place. The rest of the algorithm remains $\mathcal{O}(1)$ in space as well, as the only additional data required are the *left* and *right* pointers, and the *sum*.

(b)

To begin, we must first define what would constitute the worst case. Since the black box returns **all** possible pairs of chips that add to the target, the worst case would be achieved when every possible pair of chips sums to the target. For this to occur, each chip must take on the value $\frac{target}{2}$ since this would allow any two chips to sum to the target value. At the very least, the black box is bound by the number of possible pairings that it must return. For a set of n chips, this value is calculated as:

$$\binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} = \Omega(n^2)$$

Q.E.D

(c)

Consider the pseudocode below:

```
FindBestChips(C, target)
1  HeapSort(C)
2  left = 1
3  right = C.length
4  while left < right
5      sum = C[left] + C[right]
6      difference = target − sum
7      if difference > 0 and difference < min-difference
8          min-difference = difference
9          closest-pair = (C[left], C[right])
10     if sum < target
11         left = left + 1
12     elseif sum > target
13         right = right − 1
14     else
15         return (C[left], C[right])
16 return closest-pair
```

Like part a), the implementation of heapsort forces this algorithm to $\mathcal{O}(n \lg n)$ since this is heapsort's worst-case (and best-case) time complexity (CLRS, 3rd Edition, Exercise 6.4-4). The only difference between this algorithm and the algorithm in part a) is within the **while** loop (lines 4-15). Instead of only checking if a correct pair is found, we must track the best pair (closest *sum* $\leq$ *target*). Regardless, in the worst case (no perfectly-summing chips) we still must iterate over each chip at most once. Therefore this section of the algorithm remains $\mathcal{O}(n)$. By part a), the full algorithm is still $\mathcal{O}(n \lg n)$.

Since, in terms of memory, the only difference between this algorithm and part a) are the *closest-pair* and *min-difference* variables, this algorithm remains $\mathcal{O}(1)$ in space.

Inductive Proof of Algorithm Correctness:

To preface, this proof only addresses the section of the algorithm that succeeds heapsort since it is known that the heapsort algorithm is correct. Because of this, we assume that all arrays are pre-sorted.

Basis:

Consider the following sorted array: $A = \{1, 2\}$, let $target = 4$.

After setting the *left* and *right* pointers to 1 and 2 respectively (lines 2, 3), the **while** loop is engaged. The $difference = 4 - (1 + 2) = 1$, and the *closest-pair* will be set as 1 and 2 since it is the first pair to be checked. Next, *left* will be incremented by 1 since $1 + 2 = 3 < target$ (lines 10, 11). This will break the **while** loop's condition since *left* is no longer less than *right*. Finally, the *closest-pair* is returned as $(1, 2)$, which is indeed the closest pair in the array whose sum does not exceed the target value. Therefore, this algorithm is correct for the array $A = \{1, 2\}$.

Inductive Hypothesis:

Assume that this algorithm is correct for a sorted array A of length n.

Inductive Step:

By adding an element to the sorted array A (assuming it remains sorted), the arrays length increases to $n + 1$. Let k represent the value of the element that was added.

The value of k can be:

$$k > \mathit{target} - \min(A)$$

OR

$$k \leq \mathit{target} - \min(A)$$

Where $\min(A)$ represents the smallest value in A before k was added. Beginning with the first case, $(k > \mathit{target} - min(A))$ by rearranging this inequality:

$$k + \min(A) > \mathit{target}$$

It can be shown that for all elements in A, k will be too large to form an appropriate sum. Therefore, by lines 12 and 13, it is guaranteed that k will be iterated over, leaving the algorithm with an array that is effectively unchanged from before k was added.

In the second case, k may not affect the outcome that existed before it was added. If this happens, then it will just be iterated over either by lines 12 and 13, or by lines 10 and 11. However, if it happens that the addition of k creates a new best pairing, k will either be iterated over (and tracked by *closest-pair*) or simply returned immediately if the its addition with another chip is equal to *target*.

Therefore the addition of some k does not affect the correctness of the algorithm.

Q.E.D

Proof of Termination:

In lines 2 and 3, *left* and *right* are set to the start and end of the input array respectively. Therefore, before entering the **while** loop, $\mathit{left} \leq \mathit{right}$. If equal, the loop will not be entered whatsoever, and an empty pair will be returned since no single chip can be used for both of the two broken chips. Otherwise, given that $left > right$, the loop will be engaged. In each iteration of the loop, the two values either approach each other or will be returned immediately if there is a perfectly summing pair. If there is no perfectly summing pair, then either *left* must be incremented or *right* decremented in each iteration. Therefore, for an input array of size n, it will take a maximum of n iterations for $left = right$ and the termination of the **while** loop and algorithm.

Q.E.D

Question 2

(a)

The procedure has three recursive calls that split the input in different ways. Other than this, only two if statements are called that perform a comparison and a swap which take only $\Theta(1)$ time. We can simply add up the work being done and scale the inputs to the recursive calls accordingly. $T(n) = 2T(n/2) + T(n-1) + \Theta(1)$

(b)

Induction Hypothesis: after calling sort(A, i, j), with j - i < k, A[i..j] is sorted.

Basis: If k = 0, then i = j and the subarray has one index which is trivially already sorted.

Inductive Step: We use strong induction to assume if j - i < k + 1 for sort(A, i, j), then the subarray A[i..j] is sorted. Then we use this to prove that if j - i = k + 1 and sort(A, i, j) is called, A[i..j] is sorted.

The first if statement handles the basis step. The first call to sort calls it for i = i and j = m = floor((i + j)/2) which we can assume is sorted if j - i < k + 1. $floor((i+j)/2) - i \leq (j-i)/2 < j - i = k + 1 \implies$ A[i, m] is sorted by IH.

Similarly, the second call to sort calls it for i = m + 1 and j = j which we can assume is sorted if j - i < k + 1. $j - floor((i+j)/2) + 1 \leq (j-i)/2 + 1 < j - i = k + 1 \implies$ A[m + 1, j] is sorted by IH.

The second if statement takes the largest elements from both sorted halves of the array and places the biggest one at the end of the array. This means only A[i..j-1] needs to be sorted. The last call to sort(A, i, j - 1) ensures that the rest of the array is sorted by the inductive hypothesis if j - 1 - i < k + 1 which it is in this case.

Q.E.D

Question 3

(a)

Assume S_n represents sequence S with size of n elements:

```
FindIntersection(A_n, B_n, n)
1  I = empty list
2  a = 0, b = 0, most_recent = -1
3  while a < n and b < n
4      if A[a] is B[b]
5          if A[a] is not most_recent
6              I.insert(A[a])
7              most_recent = A[a]
8          a = a + 1
9          b = b + 1
10     elseif A[a] < B[b]
11         a = a + 1
12     elseif A[a] > B[b]
13         b = b + 1
14 return I
```

Description and Complexity:

In this algorithm, we use pointers in both of our arrays to find the common elements between them. The algorithm is a **while** loop to find the common elements until one of the pointers gets to the end of the array (lines 3 - 13). This is simply done by inserting the number whenever the elements that our pointers point at have the same value. And whenever the values are different, the pointer pointing at the smaller number would be incremented, hoping that it would point at a number closer (or maybe equal) to the other one. It is also necessary to keep updating the most recent value that was added to the list to avoid adding duplicates. In terms of space complexity, we are adding a new array for the intersection with a maximum size of n (if the arrays are identical), resulting $\mathcal{O}(n)$ complexity. In terms of time, both pointers increment towards the end of their arrays through the loops, which means it has $\mathcal{O}(n+n) = \mathcal{O}(n)$ time complexity in the worst-case scenario.

Inductive Proof of Algorithm Correctness:

Basis:

Consider the following arrays: $A = \{1\}$, $B = \{1\}$. In this case, our first value would be inserted during the first loop. Since both of our pointers are incremented up to the size of the arrays, the **while** loop is ended. This means we are left with $I = \{1\}$ which is the correct answer.

Inductive Hypothesis:

Assume that the algorithm is correct for the two sorted arrays $A = \{a_0, a_1, \ldots, a_n\}$ and $B = \{b_0, b_1, \ldots, b_n\}$ when $1 \leq n \leq k-1$. Now we need to prove that it is still correct for $A = a_0, a_1, \ldots, a_k, B = b_0, b_1, \ldots, b_k$. Let's also assume that the pointers of arrays A and B are represented by a and b respectively.

Inductive Step:

Based on our inductive hypothesis, we can assume that the algorithm works correctly up to either a_{k-1} or b_{k-1}. Let's assume that the pointer of array A reaches the end first, which means that $\max(A) \leq \max(B)$. This is because the pointer of an array is incremented if and only if the value of its number is less or equal to the value of the other pointer. In other words, pointer of A can only get from k to $k+1$ when the last element of A is less or equal to the current element of B ($A[a_k] \leq B[b]$). Here we have two possibilities to analyze:

1. If $A[a_k] = B[b]$, then the values would be compared and added to the intersection list before the end of the **while** loop. And since this is the last element of array A, there wouldn't

be any common element left.

2. If $A[a_k] < B[b]$, then all the common elements have already been inserted since any number after $B[b]$ would be greater or equal to $B[b]$.

The same scenario would be valid in the opposite case, when the pointer of B reaches the end first instead, which means that the algorithm is correct in either case.

Q.E.D

Proof of Termination:

The **while** loop ends when either of the pointers reaches the end of array (i.e. the pointer would be equal to the size of array, n). In each loop, at least one of the pointers is getting incremented, meaning that it will take less than $n+n$ loops for the termination of the **while** loop.

(b)

Assume S_n represents sequence S with size of n elements:

```
FindUnion(A_n, B_n, n)
U = empty list
a = 0, b = 0, most_recent = -1
while a < n and b < n
    if A[a] is B[b]
        if A[a] > most_recent
            U.insert(A[a])
            most_recent = A[a]
        a = a + 1
        b = b + 1
    elseif A[a] < B[b]
        if A[a] > most_recent
            U.insert(A[a])
            most_recent = A[a]
        a = a + 1
    elseif A[a] > B[b]
        if B[b] > most_recent
            U.insert(B[b])
            most_recent = B[b]
        b = b + 1
if a is n
    for b to n - 1
        if A[n - 1] < B[b] and B[b] > most_recent
            U.insert(B[b])
            most_recent = B[b]
elseif b is n
    for a to n - 1
        if A[a] > B[n - 1] and A[a] > most_recent
            U.insert(A[a])
            most_recent = A[a]
return U
```

Description and Complexity:

In this algorithm, we use pointers in both of our arrays to find the union of them. The algorithm starts with a **while** loop to insert all unique elements until one of the pointers gets to the end of the array (lines 3 - 19). This is simply done by inserting the number whenever the elements that our pointers point at have different values. And whenever the values are the same, only one of them would be added while both pointers get incremented. After the **while** loop, as one of the pointers get to the end, the algorithm starts a **for** loop in the other array to insert all the values left (lines 20 - 29). It is also necessary to keep updating the most recent value that was added to the list to avoid adding duplicates. In terms of space complexity, we are adding a new array for the union with maximum size of $n+n$ (if the arrays have no common elements), resulting $\mathcal{O}(n)$ complexity. In terms of time, both pointers increment towards the end of their arrays through the loops, which means it has $\mathcal{O}(n+n) = \mathcal{O}(n)$ time complexity.

Inductive Proof of Algorithm Correctness:

Basis:

Consider the following arrays: $A = \{1\}$, $B = \{2\}$. In this case, the first value that would be added is 1, since it is less than 2. After that, as the pointer in array A reaches the end, we move to the **for** loop for array B, inserting all unique values from the current pointer to the end of this array (in this case, only one element). This means we are left with $U = \{1, 2\}$ which is the correct answer.

Inductive Hypothesis:

Assume that the algorithm is correct for the two sorted arrays $A = \{a_0, a_1, \ldots, a_n\}$ and $B = \{b_0, b_1, \ldots, b_n\}$ when $1 \leq n \leq k-1$. Now we need to prove that it is still correct for

$A = a_0, a_1, \ldots, a_k, B = b_0, b_1, \ldots, b_k$. Let's also assume that the pointers of arrays A and B are represented by a and b respectively.

Inductive Step:

Based on our inductive hypothesis, we can assume that the algorithm works correctly up to either a_{k-1} or b_{k-1}. Let's assume that the pointer of array A reaches the end first, which means that $\max(A) \leq \max(B)$. This is because the pointer of an array is incremented if and only if the value of its number is less or equal to the value of the other pointer. In other words, pointer of A can only get from k to $k+1$ when the last element of A is less or equal to the current element of B ($A[a_k] \leq B[b]$). Here we have two possibilities to analyze:

1. If $A[a_k] = B[b]$, then the values would be compared and added to the intersection list before the end of the **while** loop. And since this is the last element of array A, there wouldn't be any element in this array to be added to our list. The **for** loop would then add all the values in array B while making sure there isn't any duplicate from $b+1$ up to $B[b_k]$ as expected.

2. If $A[a_k] < B[b]$, then the value of $A[a_k]$ would be inserted to the list at the end of the **while** loop. And since this is the last element of array A, there wouldn't be any element in this array to be added to our list. The **for** loop would then add all the values in array B while making sure there isn't any duplicate from b up to $B[b_k]$ as expected.

The same scenario would be valid in the opposite case, when the pointer of B reaches the end first instead, which means that the algorithm is correct in either case.

Q.E.D

Proof of Termination:

The **while** loop ends when either of the pointers reaches the end of array (i.e. the pointer would be equal to the size of array, n). In each loop, at least one of the pointers is getting incremented, meaning that it will take less than $n+n$ loops for the termination of the **while** loop. The **for** loop will also be terminated in maximum of n loops as it gets incremented every single time with exit point of when it is equal to the size of the array.

(c)

Assume S_n represents sequence S with size of n elements:

```
FINDDIFF(A_n, B_n, n)
1   D = empty list
2   I = FINDINTERSECTION(A_n, B_n, n)
3   a = 0, i = 0, size = I.size, most_recent = −1
4   while a < n
5       if i is size or A[a] < I[i]
6           if A[a] is not most_recent
7               D.insert(A[a])
8               most_recent = A[a]
9           a = a + 1
10      elseif A[a] > I[i]
11          i = i + 1
12      else
13          a = a + 1
14  return D
```

Description and Complexity:

In this algorithm, we first find the intersection between the two arrays using the algorithm provided in part a). Then, we use pointers in both of our arrays (A and I, the intersection) to find the unique elements that are not in array B. The algorithm includes a **while** loop to find the elements in array A that are not in I (i.e. not common with B) until the end of array A (lines 4 - 13). This is simply done by inserting the number whenever the element pointed in array A is smaller than the one pointed in I. And whenever its the other way around, the intersection array pointer would be incremented, hoping that it would point at a number closer (or maybe equal) to the other one so we can decide whether it is just in A or in both A and I. It is also necessary to keep updating the most recent value that was added to the list to avoid adding duplicates. In terms of space complexity, we are adding a new array for the difference with maximum size of n (if the arrays have no element in common) and an array for finding the intersection, resulting $\mathcal{O}(n+n) = \mathcal{O}(n)$ complexity. In terms of time, both pointers increment towards the end of their arrays through the loops, which means it has $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$ time complexity.

Inductive Proof of Algorithm Correctness:

Basis:

Consider the following arrays: $A = \{1\}$, $B = \{2\}$. In this case, there is no common element between the two, which means $I = \{\emptyset\}$. In the **while** loop, since the pointer is the same as the size (both equal to 0), the first element in A is inserted and the pointer is incremented. The algorithm ends here as the pointer of A has reached the end of the array. This means we are left with $D = \{1\}$ which is the correct answer.

Inductive Hypothesis:

Assume that the algorithm is correct for the two sorted arrays $A = \{a_0, a_1, \dots, a_n\}$, $B = \{b_0, b_1, \dots, b_n\}$ and $I = \{i_0, i_1, \dots, b_m\}$ when $1 \leq m \leq n \leq k-1$. Now we need to prove that it is still correct for $A = a_0, a_1, \dots, a_k, B = b_0, b_1, \dots, b_k$. Let's also assume that the pointers of arrays A and I are represented by a and i respectively.

Inductive Step:

Based on our inductive hypothesis, we can assume that the algorithm works correctly up to

a_{k-1}. We know that $\max(A) \geq \max(I)$. This is because the elements of I are already in A, meaning that the maximum value of A cannot be smaller than the maximum value of I. Here we have two possibilities to analyze:

1. If $A[a_k] = \max(I)$, then both pointers will be incremented until they both point at their last element. In that loop, since they values are equal, no element will be inserted and the **while** loop will be ended as the pointer a would reach the end of the array

2. If $A[a_k] > \max(I)$, then because of the **else if** statement, the pointer i would be incremented until it reaches the end of the array I. Then, since i is now equal to $size$, $A[a_k]$ would be inserted to the list as an element that is not commone between A and B.

Both cases will give us a correct result, which means that the algorithm is correct. **Q.E.D**

Proof of Termination:

The **while** loop ends when pointer of array A reaches the end of the array (i.e. the pointer would be equal to the size of array, n). Until I pointer reaches the end of the array, at least one of the pointers is getting incremented in our while loop. After that, only the A pointer gets incremented, meaning that it will take less than $n + n$ loops for the termination of the **while** loop.

Question 4

Note: Leftist heap will be abbreviated as LH.

(a)

Let s represent the largest complete sub-tree of an LH L starting from the root. Since the rank of the root of L will be the length of the shortest path from the root to the leaf, the height of s will have a height of the rank of the root of L (otherwise s would not be complete). If m is the number of nodes in s, the height of s will be $\mathcal{O}(\lg m)$ which will be the same as the rank of the root. If n is the number of nodes in L, then since s is a sub-tree of L, $n \geq m \implies \lg n \geq \lg m \implies$ the rank of the root of an LH is $\mathcal{O}(\log n)$.

Q.E.D

(b)

From (a), we know that the rank of the root of an LH is $\mathcal{O}(\log n)$ which is the same as the length of the rightmost path. We also know that to merge two sorted sequences using **MERGE** (CLRS, 4th, page 38), it takes $\Theta(n)$. If the size of two leftist heaps l_1 and l_2 have sizes n_1 and n_2, then to merge the rightmost paths of l_1 and l_2, the **MERGE** procedure will have to iterate over $\mathcal{O}(\log n_1) + \mathcal{O}(\log n_2) = \mathcal{O}(\log n)$ elements. Therefore, to merge l_1 and l_2, it takes $\mathcal{O}(\log n)$ time. To show that the order invariant is maintained, suppose that the root of an LH l_1 with no right child is being added to the right child of the root of another LH l_2 with its right child removed in the LH merge procedure (where merging two LHs splits both of them into sub-trees with their root's right child removed). Since the key of the root of l_1 is larger (because we are assuming this is happening in the merge procedure) and all other nodes of l_1 are larger than its root by the definition of an LH, all other nodes in l_1 will be larger than the root of l_2. The left side of l_2 and l_1 are not changed and they were already LHs, so the order invariant is already maintained for the left side of both trees.

Q.E.D

(c)

Since merging the rightmost paths of two trees t_1 and t_2 involves recursively removing the right child from the produced sub-trees, merging the rightmost paths of two trees can be modelled as merging the right path of m sub-trees with their root not having a right child. After splitting t_1 and t_2 into m sub-trees, only the right child of the root of each sub-tree will be changed. Take one of the subtrees m_1. If a new tree of arbitrary shape and size is set as the right child of m_1, then the left child subtree of m_1 would not have been changed. Since the rank of a node is defined as $1 + minrank(left(x)), rank(right(x))$ which depends on the rank of the children and the left subtree of m_1 was not changed, the rank of the left child does not change as well. In contrast, the rank of the right child of m_1 will be of arbitrary size as it will have its right child assigned to an arbitrary sized tree in the recursive call to merge.

Q.E.D

(d)

From (b) we know that the process of merging the rightmost path of two LHs takes $\mathcal{O}(\log n)$ time and that it keeps the order invariant. Therefore, we need to show that the rank update step takes $\mathcal{O}(\log n)$ time and that it maintains the balance invariant and the order invariant. Suppose we are left with an LH l of size n that is the result of merging the rightmost path of two LHs. We will prove that the sub-tree T_k of size k is an LH after applying the rank update step using induction.

Inudction Hypothesis: In the merge procedure, by applying the rank update step to the root node of a tree T_k of size k, T_k is a LH for $k > 0$. **Basis:** For k = 1, the tree has one root node with no children and it trivially maintains the balance and order invariant, so it is a LH. **Inductive Step:** T_{k+1}, can have zero, one, or two children that are sub-trees, so the number of nodes in these sub-trees will be less than the number of nodes in T_{k+1}. By the IH, since the number of nodes is less than k + 1, then these sub-trees are LHs. From (b) we have already shown that the merge procedure ensures that the order invariant is maintained. The merge procedure will move whichever child has a smaller rank to the right child so that the balance invariant is maintained and it does not change the height of the children, so the order invariant from (b) is not modified. Therefore, the resulting tree T_{k+1} is also a LH.

From (a), we also know that the length of the rightmost path of the merged LH will be $\mathcal{O}(\log n_1) + \mathcal{O}(\log n_2) = \mathcal{O}(\log n)$. From (c) we know that merging the rightmost paths will only change the rank of nodes on the rightmost path, so the rank update step only needs to be applied to the rightmost path. If the rightmost path of the merged tree is traversed from the bottom rightmost leafnode, then $\log n$ nodes will be traversed with their children being optionally swapped. Since swapping the child of each parent is just swapping two pointers which takes $\Theta(1)$ time, the rank update step takes $\mathcal{O}(\log n)$ time.

Q.E.D

(e)

To implement **DeleteMin** and **Insert**, we can utilize the **Merge** procedure that runs in $\mathcal{O}(\log n)$ time. The subtree starting at each node of an LH is also an LH or else the order and balance invariant would not be maintained which is why we can use the merge procedure here. Both **DeleteMin** and **Insert** run in $\mathcal{O}(\log n)$ time because they call the **Merge** procedure once. For DeleteMin, since we know that the root node is the smallest node in the tree by the order invariant, we can remove the root node, so we are left with the root's two child sub-trees that are also LHs which can have the merge procedure applied to them to create a single LH. For Insert, we can imagine that the node being inserted is itself a LH. Since it needs to be placed into another LH, we can also use the Merge procedure to create a final LH. The following pseudocode assumes that the merge procedure returns the merged LH and that each node of an LH is also an LH:

```
DELETEMIN(H)
  l = root.left
  r = root.right
  H = MERGE(l, r)

INSERT(H, i)
  H = MERGE(H, i)
```