Homework 2

ECE345 - Group 16

October 13th, 2023

Total pages: TBD

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:

$$k > target - \min(A)$$

OR

$$k \leq target - \min(A)$$

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

$$k + \min(A) > target$$

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, $left \leq right$. 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. QED.

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 if a is n
15     for b to n − 1
16         if A[n − 1] is B[b] and B[b] is not most_recent
17             I.insert(B[b])
18             most_recent = B[b]
19 elseif b is n
20     for a to n − 1
21         if A[a] is B[n − 1] and A[a] is not most_recent
22             I.insert(A[a])
23             most_recent = A[a]
24 return I
```

(b)

Assume S_n represents sequence S with size of n elements:

```
FindUnion(A_n, B_n, n)
 1  U = 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] > most_recent
 6                    U.insert(A[a])
 7                    most_recent = A[a]
 8              a = a + 1
 9              b = b + 1
10        elseif A[a] < B[b]
11              if A[a] > most_recent
12                    U.insert(A[a])
13                    most_recent = A[a]
14              a = a + 1
15        elseif A[a] > B[b]
16              if B[b] > most_recent
17                    U.insert(B[b])
18                    most_recent = B[b]
19              b = b + 1
20  if a is n
21        for b to n − 1
22              if A[n − 1] < B[b] and B[b] > most_recent
23                    U.insert(B[b])
24                    most_recent = B[b]
25  elseif b is n
26        for a to n − 1
27              if A[a] > B[n − 1] and A[a] > most_recent
28                    U.insert(A[a])
29                    most_recent = A[a]
30  return U
```

(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 and i ≤ size
5      if A[a] < I[i] or i is size
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
```

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 $\mathcal{O}(\log n)$. QED

(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. QED

(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. QED

(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. QED

(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)
1 $l = root.left$
2 $r = root.right$
3 $H = \text{Merge}(l, r)$

Insert(H, i)
1 $H = \text{Merge}(H, i)$