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. The chips at each pointer are then summed and compared to the target value. If the sum is larger than the target value, then it must be decreased (decrement the *right* pointer). Otherwise, 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.

In terms of the time complexity, 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). 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 $\mathcal{O}(n \lg n)$.

(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:

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

(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 $difference < min\text{-}difference$
 8          $min\text{-}difference = difference$
 9          $closest\text{-}pair = (C[left], C[right])$
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 $closest\text{-}pair$
```

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 $\mathcal{O}(n \lg n)$.

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($A_n, B_n, n$)
$I$ = empty list
$a = 0$, $b = 0$, $most\_recent = -1$
while $a < n$ and $b < n$
    if $A[a]$ is $B[b]$
        if $A[a]$ is not $most\_recent$
            $I.insert(A[a])$
            $most\_recent = A[a]$
        $a = a + 1$
        $b = b + 1$
    elseif $A[a] < B[b]$
        $a = a + 1$
    elseif $A[a] > B[b]$
        $b = b + 1$
```

```
if $a$ is $n$
    for $b$ to $n-1$
        if $A[n-1]$ is $B[b]$ and $B[b]$ is not $most\_recent$
            $I.insert(B[b])$
            $most\_recent = B[b]$
elseif $b$ is $n$
    for $a$ to $n-1$
        if $A[a]$ is $B[n-1]$ and $A[a]$ is not $most\_recent$
            $I.insert(A[a])$
            $most\_recent = A[a]$
return $I$
```

(b)

Assume S_n represents sequence S with size of n elements:

```
FindUnion($A_n, B_n, n$)
$I$ = empty list
$a = 0$, $b = 0$, $most\_recent = -1$
while $a < n$ and $b < n$
    if $A[a]$ is $B[b]$
        if $A[a] > most\_recent$
            $I.insert(A[a])$
            $most\_recent = A[a]$
        $a = a + 1$
        $b = b + 1$
    elseif $A[a] < B[b]$
        if $A[a] > most\_recent$
            $I.insert(A[a])$
            $most\_recent = A[a]$
        $a = a + 1$
    elseif $A[a] > B[b]$
        if $B[b] > most\_recent$
            $I.insert(B[b])$
            $most\_recent = B[b]$
        $b = b + 1$
```

(c)

Question 4

Note: Leftist heap will be abbreviated as LH.

(a)

Let s represent the smallest complete sub-tree of an LH L starting from the root. Since the rank 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 the 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 an LH l_1 with $rank = 0$ 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 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. 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 without a right child at their roots. 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 $1 + minrank(left(x)), rank(right(x))$ and the left subtree of m_1 was not changed, the rank does not change as well. In contrast, the rank of the right child of m_1 will be of arbitrary size. 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. Since the rank update step is executed after their rightmost paths are merged, we need to show that the rank update step takes $\mathcal{O}(\log n)$ time and that it maintains the balance invariant.

Suppose we are left with a LH l that is the result of merging the rightmost path of two LHs. We will prove the balance invariant is maintained after the rank update step using induction where P(n + 1) represents the parent of the node. Basis: the rightmost leaf subtree of l has no children so it has a rank of 0 which is the lowest. Inudction Hypothesis: If the children of the root of the subtree are LHs, then swapping the children of the root of the subtree if the rank of the left child is greater than the rank of the right child will turn the subtree into a LH. Inductive Step: From (c) we know that merging the rightmost paths will only change the rank of nodes on the rightmost path. If we apply the update step, the right child will have a smaller rank. Since the right child is an LH, the left child is an LH, and the length of the rightmost path of the subtree root is the smallest, the subtree will be an LH. In From (a), we also know that the length of the rightmost path of the merged LH will be $\mathcal{O}(\log n_1) + \mathcal{O}(\log n_2) = \mathcal{O}(\log n)$. 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.

(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.

```
DeleteMin($H$)
1   $l$ = Leftist-Heap($H.root.left$)
2   $r$ = Leftist-Heap($H.root.right$)
3   $H$ = Merge($l, r$)
```

```
Insert($H, i$)
1   $H$ = Merge($H$, Leftist-Heap($H.i$))
```