ECE385: Homework 02

Farbod Mohammadzadeh
(1008360462)
Connor Ludwig
(1008333028)

13 October 2023

12 Pages

Problem 1

Part (a)

For this part we can generalize the problem to be: find two elements in a list that sum to a given value. The algorithm we will use is as follows:

```
# sort the array
quickSort(Array, 0, arr_size-1) # O(nlogn)
leftIndex = 0
rightIndex = arr_size-1
# Search Array for a pair that sums to sum
while leftIndex < rightIndex: # O(n)
    if (Array[leftIndex] + Array[rightIndex] == sum):
        print (Array[leftIndex], Array[rightIndex])
    elif (Array[leftIndex] + Array[rightIndex] < sum):
        leftIndex += 1
    else:
        rightIndex -= 1
```

This will give us a final time complexity of $O(n \lg n) + O(n)$. Which we can simplify to $O(n \lg n)$.

Part (b)

We know that upper bound of this problem is $O(n^2)$, since if you check each chip with all other chips to see if they sum to the target value, you will have to check n^2 pairs.

Part (c)

We can modify the algorithm from part (a) to be:

```
def modifiedBinarySearch(A, low, high, searchKey):
    m = 0
    while (low <= high):
        m = (high + low) // 2
        if (A[m] < searchKey and A[m+1] > searchKey):
            return m
        elif (A[m] < searchKey):
            low = m + 1
        else:
            high = m - 1
    return 0

# sort the array
quickSort(Array, 0, arr_size-1) # O(nlogn)

closestSum = 0
closestPair = []
for i in range(0, arr_size-1): # O(n)
    target = sum-Array[i]
    # O(logn)
    index = modifiedBinarySearch(Array, i+1, arr_size-1, target)
    if(index):
        pairSum = Array[i] + Array[index]
        if ( sum-pairSum < sum-closestSum ):
            closestSum = pairSum
            closestPair = [Array[i], Array[index]]
```

In this algorithm we are using a modified binary search that will find the closest sum for each index (if it exists), and will pick the pair that produce the closest sum to the target. This will give us a final time complexity of $O(n \lg n) + O(n \lg n)$. Which we can simplify to $O(n \lg n)$.

Problem 2

Part (a)

When sort(A, i, j) is called as sort$(A, 0, n-1)$, $i = 0$, and $j = n-1$. However, because arrays are indexed starting at zero, and $n-1$ is the last index, j can be treated as n. Because of these values of j and i, $m = \left\lceil \frac{i+j}{2} \right\rceil = \left\lceil \frac{n}{2} \right\rceil$ this means that sort(A, i, m) and sort$(A, m+1, j)$ call the function with half the values, giving the following start to the recurrence:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \textit{additional terms}$$

The if statement and swap take constant time independent of array size, adding $\Theta(1)$ to the recurrence equation. Additionally, at the end of the sort function, it calls itself with j decremented by 1, adding a term of $T(n-1)$ to the recurrence. Putting this all together, the recurrence characterizing the runtime of sort$(A, 0, n-1)$ is:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + T(n-1) + \Theta(1)$$

Part (b)

Proof by strong induction

Basis step: j-i=0, $\rightarrow i = j$ for the case of $i = j$, the array has one item and therefore must be sorted. $j - i = 1$ In this case this is a 2 item array and $m = 0$, so sort(A, i, m) and sort$(A, m+1, j)$ access the $i = j$ case, and immediately return. The if statement comparing $A[m]$ and $A[j]$ sorts the 2 item array as if $A[0] > A[1]$, they are swapped, otherwise nothing happens as the array is already sorted.

Hypothesis: Assume after calling sort(A, i, j) where $j - i < k$, $A[i \ldots j]$ is sorted

Induction: From the basis and hypothesis, when sort(A, i, j) is called with j-i=k-1, $A[i \ldots j]$ is sorted, the goal here is to prove that when this condition is given, calling sort(A, i, j) with j-i=k results in $A[i \ldots j]$ being sorted. When the function is called with j-i=k, the function calls itself as sort(A, i, m) and sort$(A, m+1, j)$ where $m = \left\lfloor \frac{i+j}{2} \right\rfloor$. This calls sort on both the bottom and top half of the array. If $j - i = k$, either $k = 0$ (which is known to be a sorted case), or $m - i < k$ and $j - (m+1) < k$. The proof for $m - i < k$ is: $i < j, \rightarrow \frac{i+j}{2} < j$, and the proof for $j - (m+1) < k$ is: $i < j, \rightarrow \left\lfloor \frac{i+j}{2} \right\rfloor \geq i \rightarrow \left\lfloor \frac{i+j}{2} \right\rfloor + 1 > i$

At this point, both the top and bottom half of the array must be sorted. The if statement and swap (lines 7 and 8), place the largest value between the 2 sorted halves at index $A[j]$, because both halves are sorted, this ensures that $A[j]$ is the largest value in A. Then on line 10, sort is called as $\text{sort}(A, i, j-1)$. Because $j-i=k$, $j-1-i<k$ which means that $A[i \ldots j-1]$ is a sorted array. Because $A[j]$ is greater than any value in $A[i \ldots j-1]$, $A[i \ldots j]$ must also be sorted. Therefore, if calling $\text{sort}(A, i, j)$ where $j-i<k$, $A[i \ldots j]$ is sorted, calling $\text{sort}(A, i, j)$ where $j-i=k$, $A[i \ldots j]$ is sorted. **Q.E.D.**

Problem 3

Part (a)

We can use the following algorithm to find the intersection of two sorted lists:

```
def removeDuplicates(Array, n):
    temp = list(range(n))
    j = 0
    for i in range(0, n-1):
        if Array[i] != Array[i+1]:
            temp[j] = Array[i]
            j += 1
    temp[j] = Array[n-1]
    j += 1
    for i in range(0, j):
        Array[i] = temp[i]
    return temp[:j]
A = [1, 2, 2, 2, 6, 6, 7, 8, 10, 10]
B = [2, 2, 6, 8, 10, 10, 10, 10, 10, 10]
B = removeDuplicates(B, len(B)) # O(n)
print(B)
intersection = []
j = 0
for i in range(0,len(A)): # O(n)
    if(A[i] == B[j]):
        intersection.append(A[i])
        j += 1
    elif(A[i] < B[j]):
        continue
    else:
        while(A[i] > B[j]):
            j += 1
            if(A[i] == B[j]):
                intersection.append(A[i])
```

```
                j += 1
                break
```

Part (b)

We can use the following algorithm to find the union of two sorted lists:

```
def removeDuplicates(Array, n):
    temp = list(range(n))
    j = 0
    for i in range(0, n-1):
        if Array[i] != Array[i+1]:
            temp[j] = Array[i]
            j += 1
    temp[j] = Array[n-1]
    j += 1
    for i in range(0, j):
        Array[i] = temp[i]
    Array = Array[:j]
A = [1, 2, 2, 2, 6, 6, 7, 8, 8, 10]
B = [2, 2, 6, 8, 10, 10, 10, 16, 18, 20]
A = removeDuplicates(A, len(A)) # O(n)
B = removeDuplicates(B, len(B)) # O(n)
union = []
j = 0
print(A, B)
if (len(A) > len(B)):
    for i in range(0,len(A)): # O(n)
        if(A[i] == B[j]):
            union.append(A[i])
            if (j < len(B)-1):
                j += 1
        else:
            union.append(A[i])
            union.append(B[j])
```

```
            if (j < len(B)-1):
                j += 1
else:
    for i in range(0,len(B)): # O(n)
        if(A[i] == B[j]):
            union.append(B[i])
            if (j < len(A)-1):
                j += 1
        else:
            union.append(A[i])
            union.append(B[j])
            if (j < len(A)-1):
                j += 1
```

Part (c)

We can use the following algorithm to find the difference of two sorted lists:

I DON'T KNOW

Problem 4

Part (a)

A leftist tree is a binary tree with the property that for every node, the rank of its right child is equal to or less than its left child. If rank($root$) = x, then the tree must have minimally $2^x - 1$ nodes. This can be proven using induction: **Basis:** case of x=0, if x=0, there is only 1 node in the tree. $2^0 - 1 = 0,\ 1 > 0$ **Hypothesis:** Assume if root of the leftist tree has rank x = k, the tree has at least $2^k - 1$ nodes for an arbitrary k value greater than or equal to 1.

Induction Step: The aim is to prove that given the hypothesis holding for $x = k$, it must also hold for $x = k + 1$. If the rank of the root is $x = k + 1$, then by the property of a leftist tree, the rank of the right subtree is k, and the rank of the left subtree is $\geq k$. Because the rank of the right subtree is k and it also has leftist tree properties, this is simply the already proven case for k, meaning that it must have minimally $2^k - 1$ nodes. Additionally, because the left subtree also has leftist tree properties and a rank greater than or equal to k, it must also minimally have $2^k - 1$ nodes. The root is also 1 node. Summing all of these node totals together, we reach the conclusion that if the rank of a leftist tree is $k+1$, it must have minimally: $2^k - 1 + 2^k - 1 + 1 = 2 * 2^k - 2 + 1 = 2^{k+1} - 1$ nodes. Therefore, if the claim is true for $x = k$, it is also true for x=k+1. Given that $n \geq 2^x - 1$ where x is the rank of the tree, $\log(n) \geq x log(2) - \log(1) = x$. This means that log(n) is an upper bound on the rank of the tree, to state this in other words, therefore the rank of the root in a leftist heap is $O(logn)$. **Q.E.D.**

Part (b)

When merging two leftist heaps, one analyzes the root value of the two heaps, for whichever has the smallest root value, the root and its left subtree are removed and added to the new merged heap. The subtree added is added at the root, and then every subsequent subtree is added at the end of the rightmost path of the merged tree. Because we are traversing the rightmost path of both h1 and h2 which are $O(logn1)$ and $O(logn2)$ in length respectively where n1 and n2 are the number of nodes in h1 and h2 respectively, this step in merging the 2 heaps takes $O(logn)$ time where n is the total number of nodes in the new merged heap. The last step

in merging the two heaps is reordering the subtrees to maintain the leftist heap property. To do this, starting at the bottom of the rightmost path and traversing upwards, the rank of the left and right child are compared, if the rank of the right child is equal or lesser than that of the right child, nothing is done, if the rank of the right child is greater, the right and left subtree are swapped. This operation takes constant time for each node, and is only done on the nodes in the rightmost path, which has $O(logn)$ nodes, giving this a runtime of $O(logn)$. Because each step has a runtime of $O(logn)$, the total runtime is: $2 * (logn) = O(logn)$. The tree remains partially ordered because when adding new subtrees to the rightmost path, it is being done in order from smallest root to largest root. This means that when any subtree becomes a new child of a node, the root of the subtree must be greater than the node it is a child of. Additionally, every subtree being added has the partially ordered property. This means that the new merged tree must maintain the partially ordered property. Therefore, merging two leftist heaps h1 and h2 takes $O(logn)$ time, and yields a partially ordered tree. **Q.E.D.**

Part (c)

As discussed in part b, when merging two trees, all subtrees are only inserted into the rightmost path. Additionally, swapping children is only done on the nodes in the rightmost path. Because new subtrees are only introduced as right children to nodes in the rightmost path, the only nodes which can have changed rank are nodes on the rightmost path. Any node that is not part of the rightmost path will have the same subtrees as before the merge happened and therefore will maintain its prior rank. **Q.E.D.**

Part (d)

DeleteMin The minimum value of a leftist heap is always the root, so to delete it, one can simply pop the root. When the root is popped, the result is 2 leftist heaps which were initially the left and right subtree of the root. These subtrees can then be merged into one leftist heap using the previously discussed merge and rank reordering algorithm to yield a new leftist heap with the minimum value deleted. In pseudocode, this would look like:

```
def DeleteMin(A):
```

```
root = A[0]
A = merge(left(root), right(root)
Return A
```

In this function, A is the heap, A[0] is the root of the heap. The function identifies the left and right subtrees of the pre-existing heap, and sets the heap to be the result of the two subtrees merged, which excludes the root. As previously discussed, because the leftist heap has the minimum value at the root, removing the root and merging the two subtrees removes the minimum value, and due to properties of the merge algorithm with rank reordering, the leftist heap properties are maintained, ensuring correctness. This algorithm runs in $O(logn)$ time, as removing the root is takes constant time, and as previously discussed in part B) and D), merging two leftist heaps with rank reordering takes $O(logn)$ time. Insert To insert an element in a leftist heap in a way that maintains the leftist heap property, one can simply call the merge function on the original heap and the single element. Pseudocode for this function would look like:

```
Def Insert(A, newItem)
A = merge(A, newItem)
```

In this function, A is the original heap, and newItem is the item to be inserted into the heap. By properties of the merge function with reordering discussed in part B) and D), this new resulting heap will maintain the leftist heap properties and will contain the new item. This ensures correctness in the function. This function runs in $O(logn)$ time as the only operation happening is the merge operation, which as previously discussed in part B) and D), takes $O(logn)$ time.

Part (e)

DeleteMin The minimum value of a leftist heap is always the root, so to delete it, one can simply pop the root. When the root is popped, the result is 2 leftist heaps which were initially the left and right subtree of the root. These subtrees can then be merged into one leftist heap using the previously discussed merge and rank reordering algorithm to yield a new leftist heap with the minimum value deleted. In pseudocode, this would look like:

```
def DeleteMin(A):
```

```
root = A[0]
A = merge(left(root), right(root)
Return A
```

In this function, A is the heap, $A[0]$ is the root of the heap. The function identifies the left and right subtrees of the pre-existing heap, and sets the heap to be the result of the two subtrees merged, which excludes the root. As previously discussed, because the leftist heap has the minimum value at the root, removing the root and merging the two subtrees removes the minimum value, and due to properties of the merge algorithm with rank reordering, the leftist heap properties are maintained, ensuring correctness. This algorithm runs in $O(logn)$ time, as removing the root is takes constant time, and as previously discussed in part B) and D), merging two leftist heaps with rank reordering takes $O(logn)$ time. Insert To insert an element in a leftist heap in a way that maintains the leftist heap property, one can simply call the merge function on the original heap and the single element. Pseudocode for this function would look like:

```
Def Insert(A, newItem)
A = merge(A, newItem)
```

In this function, A is the original heap, and newItem is the item to be inserted into the heap. By properties of the merge function with reordering discussed in part B) and D), this new resulting heap will maintain the leftist heap properties and will contain the new item. This ensures correctness in the function. This function runs in $O(logn)$ time as the only operation happening is the merge operation, which as previously discussed in part B) and D), takes $O(logn)$ time.