CSCI-570 Homework 2 Solutions

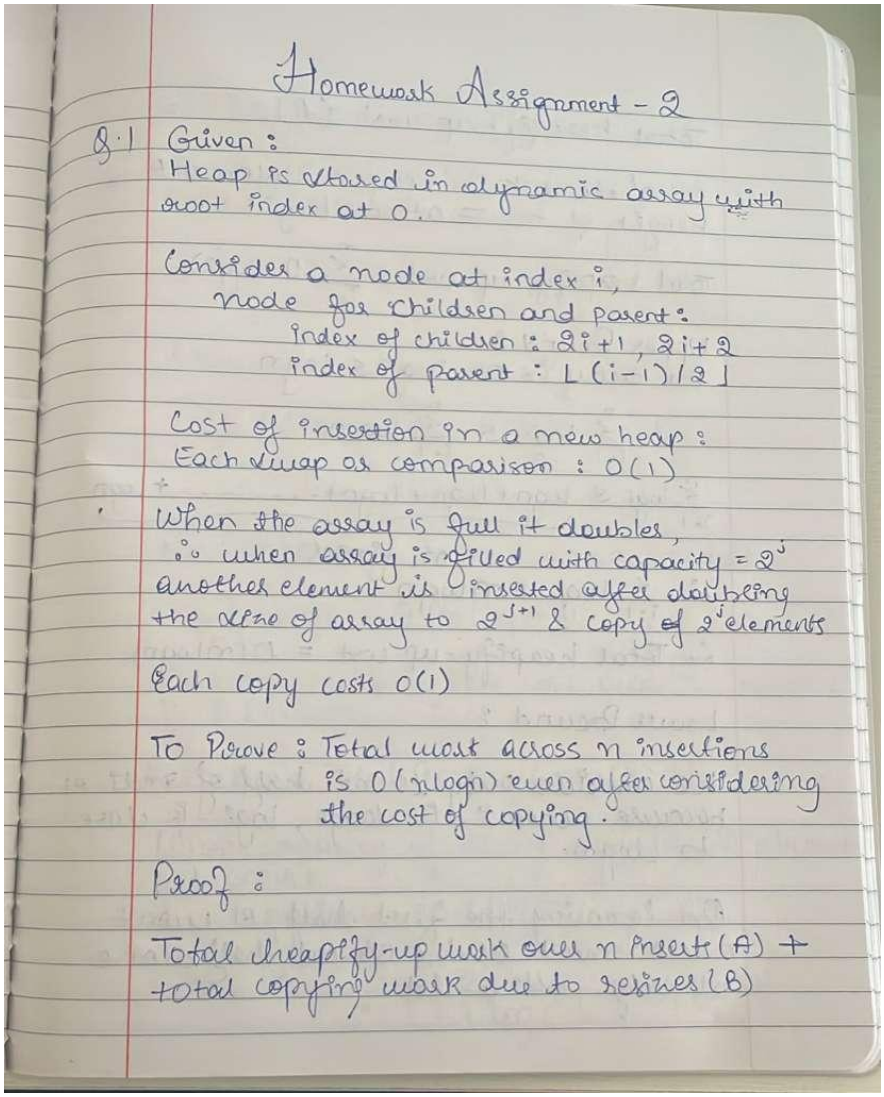
NAME: SAKSHI KULDEEP DHARIWAL

USC ID: 1851490301

Problem 1.
When discussing the implementation of binary heaps we said that the data is stored in an array. Since we did not impose any limits on the size of the heap, it means that we have to use an unbounded array. However, when discussing the runtime of the operations, we did not consider the cost of re-sizing (copying the entries) the array. Show that the cost of `build` in the online case (i.e. inserting n entries into an empty heap) is $O(n \cdot log\ n)$ even with the cost of copying taken into account. For the sake of simplicity you may assume that we store the root of the heap at index 0, instead of leaving it empty. Provide detailed calculations for the runtime.

[Hint: revisit the cost analysis of unbounded arrays.]

[4 points]



Homework Assignment - 2

Q.1 Given :
Heap is stored in dynamic array with root index at 0.

Consider a node at index i,
node for children and parent :
index of children : $2i+1$, $2i+2$
index of parent : $\lfloor (i-1)/2 \rfloor$

Cost of insertion in a new heap :
Each swap or comparison : $O(1)$

When the array is full it doubles,
∴ when array is filled with capacity = 2^j
another element is inserted after doubling
the size of array to 2^{j+1} & copy of 2^j elements

Each copy costs $O(1)$

To Prove : Total work across n insertions
is $O(n \log n)$ even after considering
the cost of copying.

Proof :

Total heapify-up work over n insert (A) +
total copying work due to resizes (B)

Total heapify-up work (A):

No. of elements in heap = i elements
Height of tree = atmost $\lfloor \log i \rfloor$

Total heapify-up cost = $\sum_{i=1}^{n} O(\log i)$

Upper Bound:
For every $i \leq n$, $\log i \leq \log n$

$\therefore \sum_{i=1}^{n} \log i \leq \sum_{i=1}^{n} \log n$

$\sum_{i=1}^{n} \log i \leq \underbrace{\log n + \log n + \log n + \cdots + \log n}_{n \text{ times}}$

$\therefore \sum_{i=1}^{n} \log i \leq n \log n$

$\therefore$ Total heapify-up cost = $O(n \log n)$

Lower Bound:

We consider only second half of indices because when i is large, $\log i$ is close to $\log n$.

On ignoring the first half, it won't matter much as the second half alone already costs a lot

$\therefore i = \lfloor \frac{n}{2} \rfloor + 1, \ldots, n$

There are atleast $n/2$ indices and for each of them:

$$\log i \geq \log\left(\frac{n}{2}\right) = \log n - \log 2 = \log n - 1$$

Now since these are $\frac{n}{2}$ elements:

$$\frac{n}{2}(\log n - 1) = \frac{n \log n}{2} - \frac{n}{2} = \Omega(n \log n)$$

$\therefore$ Combining upper & lower bounds:

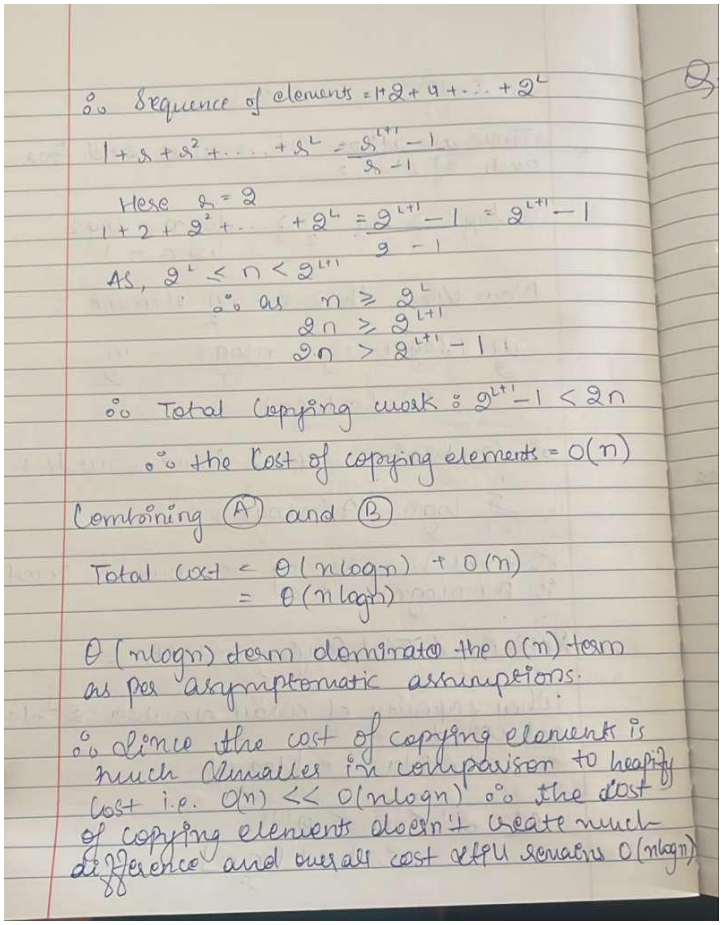
$$\sum_{i=1}^{n} \log i = \Theta(n \log n)$$

$\therefore$ total heapify-up cost over n insertions is $\Theta(n \log n)$

Resizing / Total Copying Cost (B):

When capacity of array doubles, 2^j elements get copied
consider $l = \lfloor \log_2 n \rfloor$

as $2^l \leq n \leq 2^{l+1}$
$\therefore l = \lfloor \log_2 n \rfloor$



Problem 2.
Suppose you have two binary min-heaps, A and B, with a total of n elements between them. You want to know whether A and B have a key in common. Devise an algorithm that solves this problem in $O(n \cdot log\ n)$ time and besides the two binary heaps it only uses an array of length 2.

[Hint: first come up with a $O(n \cdot log\ n)$ algorithm without the restriction on the size of extra storage.]

[4 points]

Q.2 Case 1 : Without memory restriction.

Step 1 : Extract all elements from both heaps.

- Extraction the min element from heap of size m costs $O(\log m)$
- Doing the extraction m times $\Rightarrow O(m \log m)$
- Say heap B has n_1 elements
- Extracting n_1 elements cost $\Rightarrow O(n_1 \log n_1)$
- Total cost : $O(m \log m + n_1 \log n_1)$
- $m + n_1 = n$

∴ Total cost : $O(n \log n)$

Step 2 : Store the elements in 2 arrays in sorted order.

Step 3 : Check for common elements using a linear scan through both sorted arrays in $O(n)$

Eg : Heap A : {4, 8, 12}
Heap B : {3, 8, 15, 20}

$m = 3$ $n_1 = 4$ total $n = 7$

From A : ∴ $A_{sorted} = [4, 8, 12]$
extract-min → 4
extract-min → 8
extract-min → 12

From B :
extract-min → 3
extract-min → 8
extract-min → 15
extract-min → 20

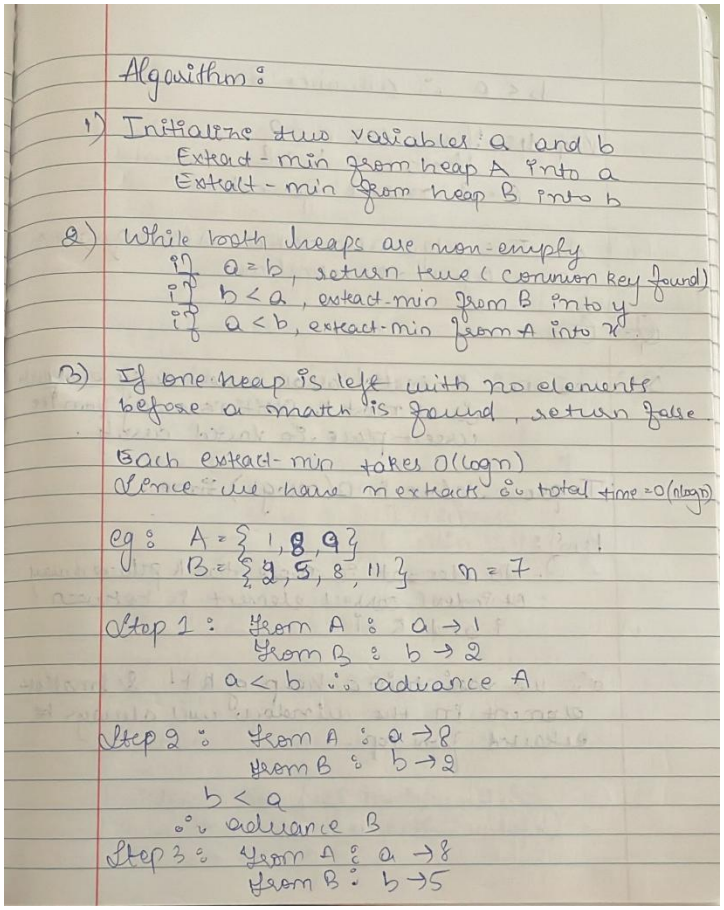
∴ $B_{sorted} = [3, 8, 15, 20]$

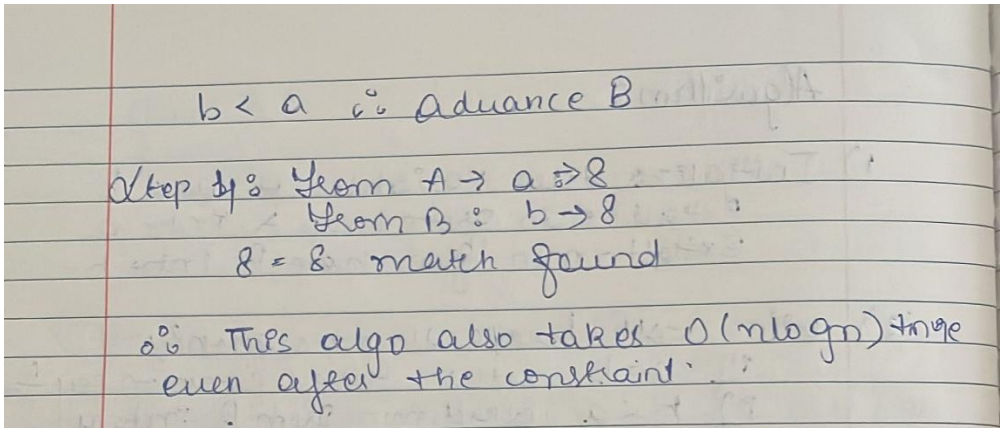
Now merge-like linear scan i.e. we scan through both sorted arrays.

Compare 4 & 3, since $3 < 4$, advance B
Compare 4 & 8 ; since $4 < 8$, advance A
Compare 8 & 8. They are equal.

Case 2 : With a restriction of only 2 extra array slots allowed.

Here, instead of storing the elements in a seperate array, we can directly extract elements from heap A & heap B and traverse using 2 variables only & compare.





Problem 3.
You are given a "k-sorted array": this is an almost-sorted array, where each of the elements are misplaced by less than k positions from their correct location. For example, A = [1,2,3,6,4,5,7] is a 3-sorted array, because the elements 4, 5, and 6 are misplaced by 1, 1, and 2 positions respectively from their correct locations. Design an $O(n \cdot log\ k)$ algorithm to sort the array and analyze the runtime.

[Hint: using a small heap will be helpful.]

[5 points]

Q.3 Given : K-sorted array where every element is at most K positions away from its correct place in sorted array.

To prove : Sort in $O(n \log k)$ time.

Proof :
- No element is more than K places away.
- At index i, correct element is between i and K+i.

∴ we maintain a heap of K+1, & smallest element in the window will always be extracted from top.

Algorithm :

1) Initialize a min-heap and insert first K+1 elements of array.
2) At every index i from 0 to n-1 :
   - Extract the minimum element from heap which gives the correct element for position i
   - insert the next element from the array at index i+K+1 into heap until its not empty.
3) Repeat steps ① and ② until heap is empty.

Eg : Array : [1, 2, 3, 6, 4, 5, 8] K=3

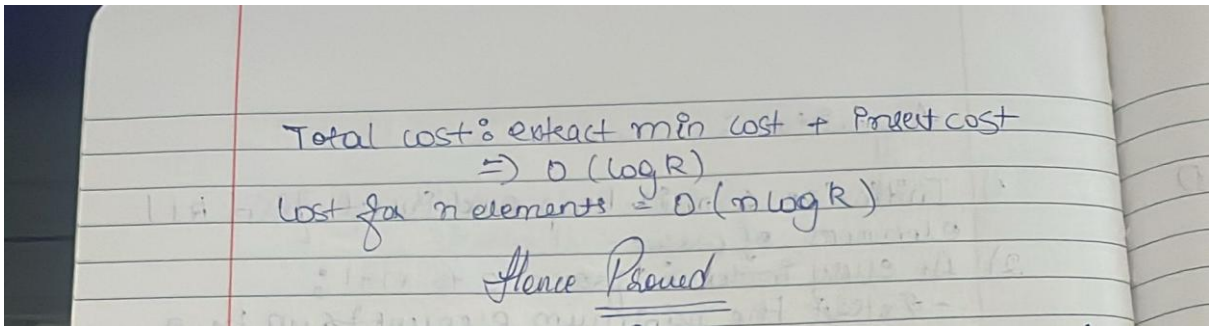
K+1 = 4 ∴ 4 elements initially

∴ [1 2 3 6] min : 1
[2 3 6 4] min : 2
[3 6 4 5] min : 3
[6 4 5 8] min : 4
[6 5 8] min : 5
[6 8] min : 6
[8] min : 8

∴ final sorted array = [1 2 3 4 5 6 8]

Runtime Analysis :

Building the heap initially : $O(k)$
Extracting-min elements : $O(\log k)$
Insert : $O(\log k)$

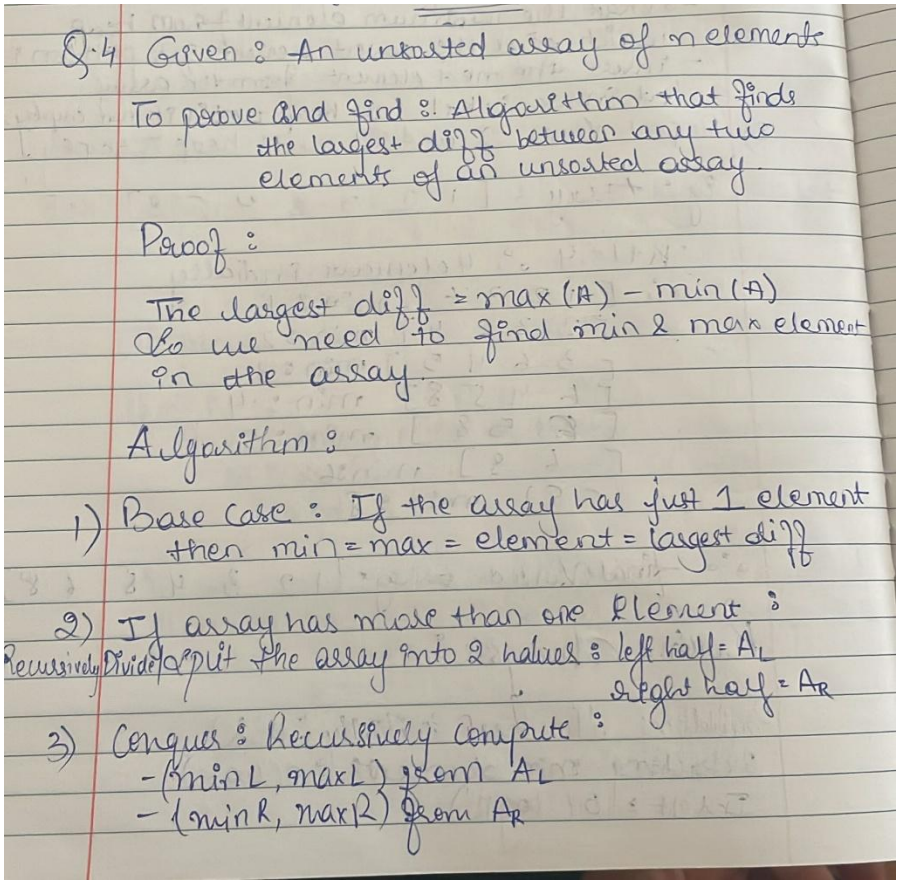


Problem 4.
Write a divide and conquer algorithm that finds the largest difference between any two elements of an unsorted array of n numbers.

Example: in case of $A = [3, 7.65, -2, 3, -0.85, 9.15, -1.5]$
the algorithm should return 11.15.

Your algorithm should run in $O(n)$ time. To achieve this, the algorithm needs $T(n) = 2 \cdot T(\frac{n}{2}) + O(1)$ as its runtime recurrence relation. After providing the algorithm, justify that its runtime is indeed $O(n)$ by unrolling its recursion tree.

[5 points]



4) Combine the 2 arrays and find min & max
Overall min = min(minL, minR)
Overall max = max(maxL, maxR)

5) Return max(A) − min(A)

eg: A = [3, 7.65, -2, 3, -0.85, 9.15, -1.5]
(indices 0, 1, 2, 3, 4, 5, 6)

① [3, 7.65, -2, 3] [-0.85, 9.15, -1.5]

② [3, 7.65] [-2, 3] [-0.85, 9.15] [-1.5]

③ [3] [7.65] [-2] [3] [-0.85] [9.15] [-1.5]

Left sub tree:

At ③ : left most subtree : [3] [7.65]
right most subtree : [-2] [3]

For left most subtree = min(3, 7.65) = 3
max(3, 7.65) = 7.65

For right most subtree : min(-2, 3) = -2
max(-2, 3) = 3

Now for left subtree overall:
min(3, -2) = -2
max(3, 7.65) = 7.65

Right sub tree:

At ③ : left most subtree : [-0.85] [9.15]
right most subtree : [-1.5]

left : min(-0.85, 9.15) = -0.85
max(-0.85, 9.15) = 9.15

right : min(-1.5) = -1.5
max(-1.5) = -1.5

Overall = min(-0.85, -1.5) = -1.5
max(9.15, -1.5) = 9.15

Combining results from left & right subtrees:

min(-2, -1.5) = -2
max(7.65, 9.15) = 9.15

Result = max − min
= 9.15 − (-2)
= 11.15

Base case : min[3, 3] = 3 min[7.65] = max[7.65]
max[3, 3] = 3 = 7.65
min[-2, -2] = max[-2, -2] = -2
min[3, 3] = max[3, 3] = 3

$\min[-0.85] = \max[-0.85) = -0.85$
$\min[9.15] = \max[9.15] = 9.15$
$\min[-1.5] = \max[-1.5] = -1.5$

Complexity Proof :

At the root level : 1 array of size n
$\therefore$ cost = c

At level 1, it splits into 2 arrays of size n/2
For each split cost = c $\therefore$ total cost = 2c

At level 2, it splits into 4 arrays of size n/4
For each split each costs c, total = 4c

At level k, we have 2^k splits each of size $n/2^k$ and each costs c
$\therefore$ total cost at level k = $2^k \cdot c$

When the length of array becomes 1, we stop.

$\frac{n}{2^k} = 1 \quad n = 2^k \quad \log_2 n = k$

So, the recursion tree has $\log_2 n + 1$ levels (0 through $\log_2 n$)

Total Cost = $c(1 + 2 + 4 + 8 + \dots + 2^{\log_2 n})$

This is a G.P.

$1 + r + r^2 + \dots + r^k = \frac{r^{k+1} - 1}{r - 1}, \quad r \neq 1$

$S = 1 + 2 + 2^2 + \dots + 2^{\log_2 n}$

$= \frac{2^{\log_2 n + 1} - 1}{2 - 1}$

$= 2^{\log_2 n + 1} - 1$

$= 2^{\log_2 n}(2) - 1$

$= n \cdot 2 - 1$

$= 2n - 1$

$\therefore$ Total Cost : $O(2n - 1)$

$\Rightarrow O(n)$

At leaves, we do base case work (reading single element, cost = d)

There are $2^{\log_2 n} = n$ leaves
$\therefore$ total cost = $d \cdot n = O(n)$

Total Cost : $T(n) = O(n) + O(n) = O(n)$

To be precise : $T(n) = c(2n - 1) + dn = \Theta(n)$

$\therefore$ cost of algorithm = $\Theta(n)$

Hence Proved