CSCI-GA.1170-001 Homework 5

Ankit Sati

TOTAL POINTS

41 / 41

QUESTION 1

1 Merging Lists 21.5 / 17

✓ **- 0 pts** *Correct*

✓ **- 0.5 pts** *Incorrect/Missing base case for k = 1 in part (a)*

✓ **- 1 pts** *Incorrect/Missing base case(s) in part (b)*

✓ **+ 1.5 pts** *Correct definition of B' in part (d)*

✓ **+ 1 pts** *Proving B' is still sorted in part (d)*

✓ **+ 1 pts** *Correctly stated that comparison gives results the same except for when i = i', j = j' in part (d).*

✓ **+ 0.5 pts** *Justification for why tree can be run in A, B' in part (d)*

✓ **+ 0.5 pts** *Proving that comparison needs to occur in part (d)*

✓ **+ 1.5 pts** *Conclusion in part (d)*

QUESTION 2

2 Running Median 10.5 / 14

Part-(a)

✓ **- 1 pts** *Incorrect/Missing Invariant*

Part-(b)

✓ **+ 1 pts** *Correct Running Time*

✓ **+ 2 pts** *Partially Correct Algorithm*

Part-(c)

✓ **- 4.5 pts** *Sub-Optimal or Incorrect Algorithm - O(n) heap insertions (but has two components from: Algorithm, Correctness or Running Time incorrect/missing/improper)*

Part-(d)

✓ **- 0 pts** *Correct*

Part-(e)

✓ **- 0.5 pts** *Incorrect/Missing Algorithm Correctness*

✓ **- 0.5 pts** *Partially Incorrect Algorithm*

QUESTION 3

3 Running Median 2 9 / 10

✓ **- 1 pts** *incorrect/missing running time and correctness for insert*

S-1

(a)

Multiway - Merge. $(A_i, \ldots, A_j, n_i, \ldots, n_j)$.

if $i < j$:

$$k = \lfloor (i+j)/2 \rfloor$$

X = Multiway_Merge $(A_i \cdots A_k, n_i \cdots n_k)$

Y = Multiway_Merge $(A_{k+1} \cdots A_j, n_{k+1} \cdots n_j)$.

$m_1 = k - i + 1$

$m_2 = j - k$.

Merge $= (X, m_1, Y, m_2)$.

MERGE (X, m_1, Y, m_2):

$X[m_1 + 1] = \infty$

$Y[m_2 + 1] = \infty$

$i = 1$

$j = 1$

for $k = 1$ to $m_1 + m_2$:

if $X[i] \leq Y[j]$:

$A[k] = X[i]$

$i = i + 1$.

else. A[k] = Y[j]

j = j+1.

The MERGE (X, m_1, Y, m_2) algorithm runs in $O(m_1 + m_2$ time as it is only traversing over the arrays X and Y once.

The first call from the main driver function will be:

MULTIWAY_MERGE (A_0, ... A_{k-1}, m_1 ... m_{k-1})

&

5-1.

(a) In each divide step we divide our problem of size K into 2 subproblems of size $K/2$.

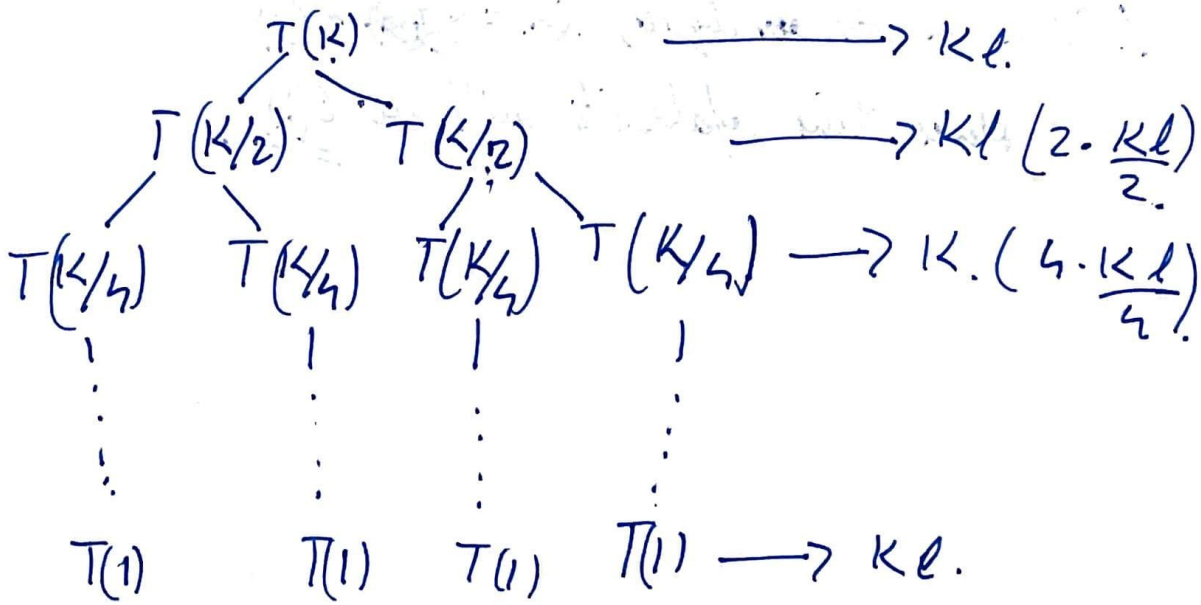
Also the time taken in the combine step (Merge (x_1, n_1, y_1, m_1) is

$$\theta(n) = \theta(Kl)$$

So, the recurrence relation will be.

$$\boxed{T(K) = 2T(K/2) + \theta(Kl)}$$

Recursion tree of this relation



$$T(K) = Kl + Kl + Kl \ldots \ldots Kl \ldots \text{(log } K \text{ times)}$$

$$T(K) = Kl \log K.$$

$$\boxed{T(K) = \theta\left(Kl \log(K)\right).}$$

(c) Total of $(n/2)$ ways of dividing a list of n numbers into two sorted arrays of equal lengths.

→ In a merging tree for the merging procedures of two subsets of A.

→ At every right level of the D.T, we will select a corresponding element for the sorted final array.

→ Let A_x & A_y be two sorted arrays.

→ There will be 1 unique branch for every process in the tree.

→ Number of leafs in a tree are. $\binom{n}{n/2}$

Proof → The order of comparisions in the merge procedure must be unique for every unique Combination of A_x & A_y.

Assumption - Order of comparision for two diffirent pairs is same.

$A_x : a_{x_1}, a_{x_2} \ldots\ldots a_{n/2}$.

$A_y : a_{y_1}, a_{y_2} \ldots\ldots a_{y/2}$.

$A_P = a_{P_1}, a_{P_2} \ldots\ldots a_{P/2}$

$A_Q = a_{Q_1}, a_{Q_2} \ldots\ldots a_{Q/2}$

For some arbitary k, i, j to select c_k. Comparing a_{xi}, a_{yi}

if $a_{xi} < a_{yi}$, $c_k = a_{xi}$

So $a_{P_i} < a_{Q_j}$ & $c_k = a_{P_i}$. (Acc to assumption)

In the next step., we would have $a_{x_{i+1}}$ & a_{y_j} since they are based on the same assumption:

$a_{x_{i+1}} = a_{p_{i+1}}$ &.

$a_{y_{i+1}} = a_{q_j}$ for selecting c_{k+1} to sort correctly.

Generalizing this we can say that A_x, A_y and A_p, A_q are same pairs to pressure sorting order.

⇒ Our Assumption is incorrect.

⇒ Order of Comparisions are unique for each A_x & A_y & every branch in D.T will be unique accordingly.

∴ The number of leafs would be equal number of ways we partition a sorted list of size $n/2 = \binom{n}{n/2}$

∴ Height of the tree would be atleast $\log_2 \binom{n}{n/2}$

$$= \log_2 \left(\frac{n!}{\frac{n}{2}! \frac{n}{2}!} \right) = \log \left(\frac{n!}{\left(\frac{n}{2}\right)^2} \right)$$

$$= \log \left(\frac{\sqrt{2\pi n} \cdot (n.e)^n}{\sqrt{2\pi \frac{n}{2}} \cdot \left(\frac{n}{2e}\right)^{n/2}} \right)$$

Sterling Approx.

$$= \log\left(\frac{\sqrt{2\pi n}\cdot(n/e)^n}{\pi n\cdot(n/2e)^n}\right)$$

$$= \log\left(\left(\frac{2}{\pi n}\right)^{1/2}\cdot 2^n\right)$$

$$= \log 2^n + 1/2\log\left(\frac{2}{\pi n}\right)$$

$$\doteq n - 1/2\log\left(\frac{\pi n}{2}\right)$$

$$= n - O(n) \quad \{\log\left(\frac{\pi}{2}n\right) \doteq O(n)\}$$

Tree height will have atleast $n - O(n)$ Comparisons.

(d) Path of execution must compare $(a_i : b_j)$.

If the path of execution in our D.T to the node representing C does not compare (a_i, b_j) then we cannot distinguish between the two arrays.

$$C_1 = [a_1/b_1 \cdots a_i, b_j \cdots a_{n/2}/b_{n/2}]$$

$$C_2 = [a_1/b_1 \cdots b_j, a_i \cdots a_{n/2}/b_{n/2}]$$

Here C is not sorted and would therefore represent the WRONG answer.

Since we assume that we have a valid D.T that performs the merge of paths in execution must pass from the node $(a_i : b_j)$.

⇒ Improved Condition for lower bound.

In order to achieve highest lower bound we can assume that we have an optimal decision tree design which only compares the elements in the array C that are consecutive.

This will contain exactly n-1 comparisions.

Therefore we can see that the merge is the optimal merging Algorithm here. as optimal lower bound is achieved.

(e) The problem of merging K- sorted lists

The total time take for multi Merge procedure = $O(n \log \log k)$.

⇒ Assume that we divide the array of n elements into n/k array with k elements each.

⇒ Now we sort all the K sorted arrays and merge lists.

For our sorting procedure we get

$$T(n) = \frac{n}{k} \cdot T(k) + O\left(n \cdot \log \log_{\frac{n}{k}}\right)$$

Smaller value of k, n becomes $T(n) = O(n \log \log n)$

However the time for Comparision sort is $= \Omega(n \log n)$

Since $n \log n \geq n \log \log n$; Our assumption is WRONG.

$T(n) \geq c_1 n \log n \cdot \forall n > n_0$

$T(n) \leq c_2 n \log n \cdot \forall n > n_0^*$

} Contradicting Statements

Hence the claim of merging with $O(n \log \log k)$ is false.

1 Merging Lists 21.5 / 17

✓ **- 0 pts** *Correct*

✓ **- 0.5 pts** *Incorrect/Missing base case for k = 1 in part (a)*

✓ **- 1 pts** *Incorrect/Missing base case(s) in part (b)*

✓ **+ 1.5 pts** *Correct definition of B' in part (d)*

✓ **+ 1 pts** *Proving B' is still sorted in part (d)*

✓ **+ 1 pts** *Correctly stated that comparison gives results the same except for when i = i', j = j' in part (d).*

✓ **+ 0.5 pts** *Justification for why tree can be run in A, B' in part (d)*

✓ **+ 0.5 pts** *Proving that comparison needs to occur in part (d)*

✓ **+ 1.5 pts** *Conclusion in part (d)*

Running Medians

5-2.

(a) We will use the heap data structure to achieve the required operation. We will use a [Max heap] to store the first half ($\frac{n}{2}$) of the elements of the array and and will use the [Min heap] to store the lower part of the array.

(b) 3 way Partition (A, P, q):

```
Pivot = A[q]
i = P-1
K = q.

for j = P to K-1.
        if A[j] == pivot
        swap. (A[j], A[K]).
        K--;
    if A[j] < pivot
            i = i+1.
            swap (A[i], A[j])

j = i+1.
while (K <= q);
        swap. (A[j], A[K])
        j++
        K++
swap. A[i+1], A[q])
return [i+1, K]
```

The above algorithm runs in $O(q-p)$

The algorithm is similar to the partition used in quick sort. Algorithm except for the case where we have also handled the elements same in value as in Pivot

$\Rightarrow \therefore$ The correctness of the algorithm is justified.

(c) Build $(A[1\ldots, n])$

i) Create a max heap H_1 out of the elements of array A.

ii) Take a min Heap H_2.

iii) for. $(i = 1$ to $n/2)$;

(iv) $\quad x = H1.\text{top}()$;

(v) $\quad H_1.\text{Pop}()$

(vi) $\quad H_2.\text{Push}(x)$.

The above algorithm creates a max heap H_1 & min Heap. H_2 to store $n/2$ elements each of the array A. Now since building a max heap / min heap takes $O(n)$ time. So, BUILD runs in $O(n)$.

(d) INSERT(x);

```
med = MEDIAN()
if (H1.size() > H2.size()):
    if x < med:
        H2.Push(H1.top())
        H1.Pop()
        H1.Push(x)
    else:
        H2.Push(x)
else if (H1.size() == H2.size()):
    if x < med:
        H1.Push(x)
    else:
        H2.Push(x)
else:
    if (x > med):
        H1.Push(H2.top())
        H2.Pop()
        H2.Push(x)
    else:
        H1.Push(x)
```

cont. →.

Since we are inserting the elements in either the max. or min heaps. Inserting the elements in max heap will call. MAX-HEAPIFY. Similarly, inserting the elements in min heap H_2 will call MIN-HEAPIFY which takes $O(\log n)$ time.

(e) Median ();

```
if H1.size() = H2.size():
    return (H1.top() + H2.top())/2.
else if H1.size() > H2.size();
        return H1.top()
    else:
        return H2.top.
```

In the above algorithm, since we are only accessing either the maximum elements in max heap or the minimum element in min heap. Therefore the complexity is $O(1)$.

2 Running Median 10.5 / 14

Part-(a)

✓ **- 1 pts** *Incorrect/Missing Invariant*

Part-(b)

✓ **+ 1 pts** *Correct Running Time*

✓ **+ 2 pts** *Partially Correct Algorithm*

Part-(c)

✓ **- 4.5 pts** *Sub-Optimal or Incorrect Algorithm - O(n) heap insertions (but has two components from: Algorithm, Correctness or Running Time incorrect/missing/improper)*

Part-(d)

✓ **- 0 pts** *Correct*

Part-(e)

✓ **- 0.5 pts** *Incorrect/Missing Algorithm Correctness*

✓ **- 0.5 pts** *Partially Incorrect Algorithm*

5-3

(a) In the previous question we were inserting the elements in a priority queue. (heap) due to which the time complexity was $O(\log n)$. In this question we will use a <u>dequeue</u> instead, due to which the time complexity will drop down to $O(1)$.

So we will use 2 dequeue

Dequeue D_1 will contain the first smallest $n/2$ elements in decreasing order from front of deque to the back.

Another deque D_2 will contain the largest $n/2$ elements in increasing order from front to back of deque.

(b) Build $(A[1, \ldots, n])$:

1. Create a min Heap H_1 out of the elements of array A.
2. Take a max heap H_2.
3. for $(i = 1$ to $n/2)$;
4. H_2. Push (H_1. top())
5. H_1. Pop()
6. while (! H_1. empty())
7. D_1. Push - front (H_1. top())

cont. →

8. H_1.Pop.

9. while (! H_2.empty()).

10. D_2.Push_front(H_2.top()).

11: H_2.Pop().

So our algorithm first stored in two heaps. H_1 & H_2 so as to build the heaps in $O(n)$ time as we did in 5-2(a).

Now for creating deque D_1, we inserted all the elements of H_2 in D_2.

"The Runtime of the Algorithm is $O(n)$ as all the operations involved here took time $O(1)$.

(c) Medrain():

```
if D1.Size() = D2.Size():
    return (D1.front() + D2.front())/2.
else if D1.size() > D2.size();
    return D1.front()
else
    return D2.front()
```

cont. →

If both the deque's have the same size, then the median will be the average of their front values.

⇒ Now if D_1 contain more elements than D_2, then front value of D_1 will be the median.

⇒ Similarly if D_2 contain more elements than D_1, then front value of D_2 will be the median.

Hence, only the front values of the deque are accessed.

∴ ∀ n in the deque, $T(n) = O(1)$

Hence time complexity will be $\underline{O(1)}$.

3 Running Median 2 **9 / 10**

✓ **- 1 pts** *incorrect/missing running time and correctness for insert*