Algorithms

Assignment 2

Ashwin Sathiyanarayanan
002227609

Question 1

(a) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$
$Guess$: $T(n) = O(n)$

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$
$= cn/2 + cn/4 + cn/8 + n$
$= (4cn + 2cn + cn)/8 + n$
$= 7cn/8 + n$
$= n[7c/8 + 1]$

$Therefore,\ for\ T(n)\ to\ be \leq cn$
$n[7c/8 + 1]\ should\ be \leq cn$

$n[7c/8 + 1] \leq cn$
$dividing\ both\ sides\ by\ n,$
$7c/8 + 1 \leq c$
$1 \leq c - 7c/8$
$1 \leq (8c - 7c)/8$
$1 \leq c/8$
$c \geq 8$

$Therefore,\ we\ have\ proved\ that$
$T(n) = O(n)\ when\ c \geq 8$

(b) $T(n) = 4T(n/2) + n^2$

Guess: $T(n) = O(n^2)$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n^2 \\
&= 4.c.(n/2)^2 + n^2 \\
&= 4.c.(n^2/4) + n^2 \\
&= cn^2 + n^2 \\
&= n^2[c + 1]
\end{aligned}
$$

We have an equal order term as a suppliment to our required proof.
Also, when $c \geq 0$, $n^2[c + 1]$ *can never be less than* cn^2.
Therefore, we have to improve our guess.

Improved guess: $T(n) = O(n^2 logn)$

$$
\begin{aligned}
T(n) &= 4T(n/2) + n^2 \\
&= 4.c.[(n/2)^2 log(n/2)] + n^2 \\
&= 4.c.[(n^2/4)log(n/2)] + n^2 \\
&= c.[n^2(logn - log2)] + n^2 \\
&= c.[n^2(logn - 1)] + n^2 \\
&= c.[n^2 logn - n^2] + n^2 \\
&= cn^2 logn - cn^2 + n^2 \\
&= cn^2 logn + n^2[1 - c]
\end{aligned}
$$

Therefore, for $T(n)$ *to be* $\leq cn^2 logn$

$n^2[1 - c]$ *should be* ≤ 0

$n^2[1 - c] \leq 0$

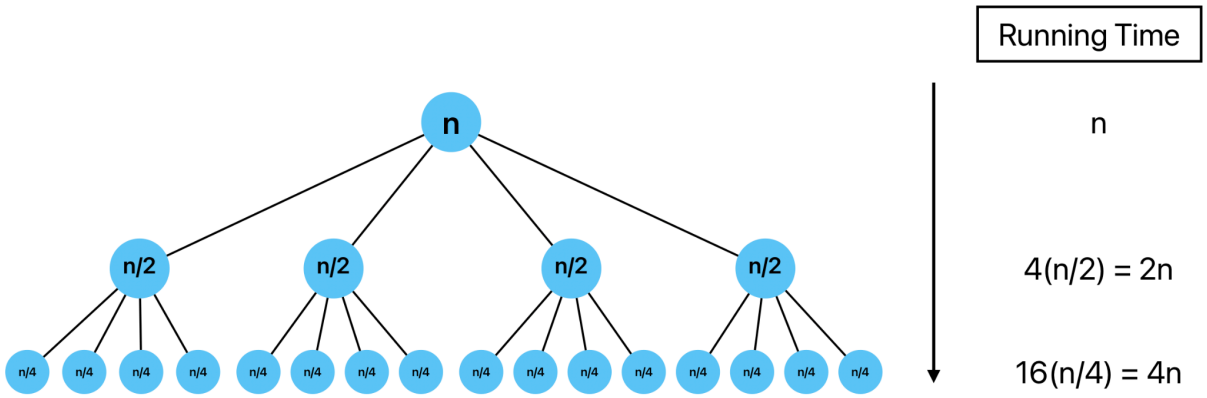
dividing both sides by n^2

$1 - c \leq 0$

$c \geq 1$

Therefore, we have proved that $T(n) = O(n^2 logn)$ *when* $c \geq 1$

Question 2:



$$Depth\ of\ the\ tree\ =\ log_2 n$$

$$Number\ of\ leaves\ =\ 4^{log_2 n} = n^2$$

$$Summation\ of\ running\ time\ of\ each\ level\ =$$
$$n\ +\ 2n\ +\ 4n\ +\ 8n\ +\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .$$

$$=\ n[1\ +\ 2\ +\ 4\ +\ 8\ +\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .]$$

$$=\ n[2^0 + 2^1 + 2^2 + 2^3 + .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .\ .]$$

$$summation\ of\ geometric\ series\ =\ \sum_{0}^{k} r^i$$

$$=\ n \sum_{i=0}^{lgn-1} (2^i)$$

$$Upper\ limit\ of\ the\ series\ =\ height\ of\ the\ tree\ except\ the\ last\ level\ =\ logn\ -\ 1$$

$$=\ n[(2^{lgn} - 1)/(2 - 1)]$$

$$=\ n[(2^{lgn} - 1)/(1)]$$

$$=\ n[2^{lgn} - 1]$$

$(2^{logn} = n^{log2})$

$= n[n^{lg2} - 1]$
$= n[n^{1} - 1]$
$= n[n - 1]$
$= n^{2} - n$

Therefore,

$$T(n) = O(n^{2})$$

Verification using Substitution method:

$T(n) = 4T(n/2) + n$

Let our guess be $T(n) = O(n^{2})$

Now, we should prove $T(n) \leq cn^{2}$

$T(n) = 4T(n/2) + n$
$\leq c.4.(n/2)^{2} + n$
$= c.4.(n^{2}/4) + n$
$= cn^{2} + n$

Since n^{2} *is an higher order term than n, we can modify our guess accordingly.*

Let our new guess be $T(n) = O(n^{2}) - dn$

Now, we should prove $T(n) \leq cn^{2} - dn$

$$T(n) = 4T(n/2) + n$$
$$\leq 4c(n/2)^2 - 4d(n/2) + n$$
$$= 4cn^2/4 - 4dn/2 + n$$
$$= cn^2 - 2dn + n$$

Now, we have to prove $n[1 - 2d] \leq -d$, *in order to prove* $T(n) = O(n^2)$

$n[1 - 2d] \leq -d$
divide both sides by n

$1 - 2d \leq -d$
$1 \leq d$
$d \geq 1$

Also, our new guess was $T(n) \leq cn^2 - dn$, *therefore* $cn^2 - 2dn + n$ *will always be less than* $cn^2 - dn$ *for any positive values of* d

Therefore, from substitution method, we have verified the result of recursion tree method.

Therefore,

$T(n) = O(n^2)$ *for all values of* $d \geq 1$ *and* $c \geq 0$.

Question 3:

(a)

$T(n) = 2T(n/4) + 1$

$a = 2,\ b = 4,\ f(n) = 1$

$n^{log_b a} = n^{log_4 2} = n^{1/2}$

$Master\ Theorem$:

$Case\ 1$:

$Check\ if\ f(n) = O(n^{log_b a - \epsilon})$

$Let\ \epsilon\ be\ 0.5,$

$$f(n) = O(n^{log_b a - \epsilon})$$
$$= O(n^{log_4 2 - 0.5})$$
$$= O(n^{1/2 - 0.5})$$
$$= O(n^0)$$
$$= O(1)$$

$1 = O(1)$
$This\ is\ true.\ Therefore,\ case\ 1\ is\ satisfied.$

$Therefore,$

$T(n) = \Theta(n^{1/2})$

(b)

$T(n) = 2T(n/4) + 1$

$a = 2,\ b = 4,\ f(n) = 1$

$n^{log_b a} = n^{log_4 2} = n^{1/2}$

Master Theorem:

Case 1:

Check if $f(n) = O(n^{log_b a - \epsilon})$

Let ϵ *be* 0.5,

$$f(n) = O(n^{log_4 2 - 0.5})$$
$$= O(n^{1/2 - 0.5})$$
$$= O(n^0)$$
$$= O(1)$$

$n \neq O(1)$

This is false. So let's move to the next case.

Case 2:

Check if $f(n) = \Theta(n^{log_b a} . lg^k n)$

Let $k = 0$,

$$f(n) = \Theta(n^{log_b a} . lg^k n)$$
$$= \Theta(n^{log_4 2} . lg^0 n)$$
$$= \Theta(n^{1/2} . 1)$$
$$= \Theta(n^{1/2})$$

$Check\ if\ f(n) = \Theta(n^{1/2})$

$n \neq \Theta(n^{1/2})$

$This\ is\ false.\ So\ let's\ move\ to\ the\ next\ case.$

$case\ 3$:

$(i)\ Check\ if\ f(n) = \Omega(n^{log_b a + \epsilon})$

$Let\ \epsilon = 0.5$

$$f(n) = \Omega(n^{log_b a + \epsilon})$$
$$= \Omega(n^{1/2 + 1/2})$$
$$= \Omega(n^1) = \Omega(n)$$

$Check\ if\ f(n) = \Omega(n)$

$n = \Omega(n)$

$This\ is\ true.$

$(ii)\ Check\ if\ a.f(n/b) \leq c.f(n)$

$2.f(n/4) \leq c.f(n)$

$2(n/4) \leq c.n$

$n/2 \leq c.n$

$let\ c = 1,$

$n/2 \leq 1.n$

$n/2 \leq n$

$This\ is\ true.$

$Therefore,$

$T(n) = \Theta(n)$

(c)

$T(n) = 2T(n/4) + n^2$

$a = 2,\ b = 4,\ f(n) = n^2$

$n^{log_b a} = n^{log_4 2} = n^{1/2}$

$Case\ 1$:

$Check\ if\ f(n) = O(n^{log_b a - \epsilon})$
$Let\ \epsilon = 0.5$

$$f(n) = O(n^{1/2 - \epsilon})$$
$$= O(n^{1/2 - 0.5})$$
$$= O(n^0)$$
$$= O(1)$$

$n^2 \neq O(1)$

$This\ is\ false.\ Therefore,\ case\ 1\ fails.\ Lets\ move\ to\ the\ next\ case.$

$Case\ 2$:

$Check\ if\ f(n) = \Theta(n^{log_b a}.lg^k n)$
$Let\ k = 0,$

$$f(n) = \Theta(n^{1/2}.lg^0 n)$$
$$= \Theta(n^{1/2}.1)$$
$$= \Theta(n^{1/2})$$

$n^2 \neq \Theta(n^{1/2})$

$This\ is\ false.\ Therefore,\ case\ 2\ fails.\ Lets\ move\ to\ the\ next\ case.$

$case\ 3$:

$(i)\ Check\ if\ f(n) = \Omega(n^{log_b a + \epsilon})$

$Let\ \epsilon = 0.5$

$f(n) = \Omega(n^{1/2 + 0.5})$
$= \Omega(n^1)$
$= \Omega(n)$

$n^2 = \Omega(n)$

$This\ is\ true.$

$(ii)\ Check\ if\ a.f(n/b) \leq c.f(n)$

$2.f(n/4) \leq c.f(n)$
$2(n^2/16) \leq c.n^2$
$n^2/8 \leq c.n^2$

$Let\ c = 1,$

$n^2/8 \leq 1.n^2$
$n^2/8 \leq n^2$

$This\ is\ true.$

$Therefore,$

$T(n) = \Theta(n^2)$

(d)

$T(n) = 2T(n/4) + n^{1/2}$

$a = 2,\ b = 4,\ f(n) = n^{1/2}$

$n^{log_b a} = n^{log_4 2} = n^{1/2}$

$Case\ 1$:

$Check\ if\ f(n) = O(n^{log_b a - \epsilon})$
$Let\ \epsilon = 0.5$

$$\begin{aligned} f(n) &= O(n^{1/2 - \epsilon}) \\ &= O(n^{1/2 - 0.5}) \\ &= O(n^0) \\ &= O(1) \end{aligned}$$

$n^{1/2} \neq O(1)$

$This\ is\ false.\ Therefore,\ case\ 1\ fails.\ Lets\ move\ to\ the\ next\ case.$

$Case\ 2$:

$Check\ if\ f(n) = \Theta(n^{log_b a}.lg^k n)$
$Let\ k = 0,$

$$\begin{aligned} f(n) &= \Theta(n^{log_b a}.lg^0 n) \\ &= \Theta(n^{1/2}.1) \\ &= \Theta(n^{1/2}) \end{aligned}$$

$n^{1/2} = \Theta(n^{1/2})$

$This\ is\ true.\ Case\ 2\ is\ satisfied.$

$Therefore,$

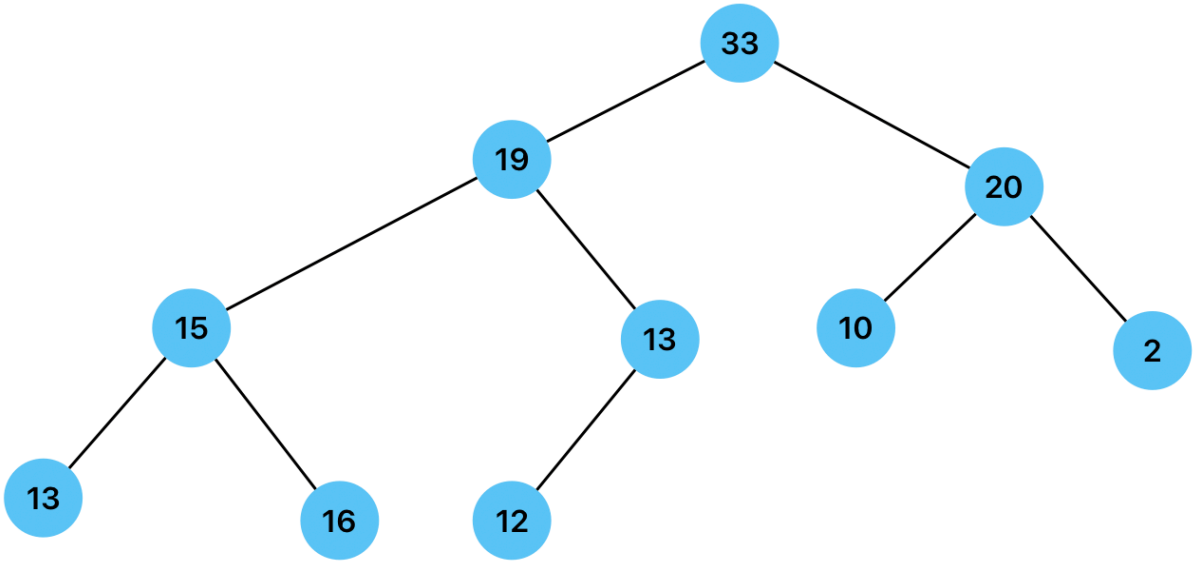
$$T(n) = \Theta(n^{1/2} logn)$$

Question 4:

(a) The lowest element will be at any one of the leaf nodes and will not be a parent to any subtree of the heap. This is because, in a max-heap, the parent node of any subtree will be greater than its children nodes. This explains why the smallest element in the heap will always be at any of the leaves and not at any of the parent nodes.

(b)

[33, 19, 20, 15, 13, 10, 2, 13, 16, 12]



This is not a max-heap because the subtree with value of 15 as the parent has a right child with a value of 16, which is greater than the value of 15. This violates the max-heap property as according to max-heap-property the value of the parent node must be greater than that of its children nodes.

Question 5:

```
def maxHeapify(A, i):

    largest = i
    isSubtreeSorted = False

    while(isSubtreeSorted is False):

        l = ( 2 * i )
        r = ( 2 * i ) + 1

        if l <= A.heapSize and A[l] > A[i]:
            largest = l
        else:
            largest = i

        if r <= A.heapSize and A[r] > A[largest]:
            largest = r

        if largest != i:
            exchange A[i] with A[largest]
            i = largest
        else:
            isSubtreeSorted = True

def buildMaxHeap(A):
    A.heapSize = len(A)
    for i = n // 2 downto 1:
        maxHeapify(A, i)
```

Question 6:

```
MergeKSortedLists(KLists):
    result = []
    minHeap = []

    if KLists.length() == 0:
        return result

    for i in range(KLists.length()):
        if KLists[i]:
            Min-Heap-Insert(minHeap, (KLists[i][0], i, 0))

    while minHeap.isempty() != true:
        ListId, ListValue, IndexofValue = Heap-Extract-Min(minHeap)
        result.append(ListValue)

        if IndexofValue+1 < KLists[ListId].length():
            Min-Heap-Insert(
              minHeap,
              (KLists[ListId][IndexofValue+1], ListId, IndexofValue+1)
            )

    return result
```

Explanation:

1. Initially, we declare the variables required (i.e.) the result merged list and the minHeap we use to get the minimum of the lists.
2. First, we check whether the given List of Lists (i.e.) KLists is empty, and if it is empty we just return the empty result list.
3. Now we start our merging algorithm. This merging algorithm works in a way that first, we loop through the list of lists and add the following values as one element to our heap: (i) the value of the element (ii) the index of that list in the list of lists and (iii) the index of that element in that particular list. We do this for all the first

elements in all the lists. We use the Min-Heap-Insert function for this because this function maintains minHeap's property of having the least element at the top after we insert each element into the heap.

4. Then we initiate a while loop with a terminating condition that the heap should not be empty. In each iteration, we get the minimum element from the heap which comprises the values that we added in the previous step. The extraction of the minimum element is done using the Min-Extract-Heap method which maintains the minHeap's property of having the least element at the top after we remove each element.

5. After this, we append the extracted minimum element to our result list. Then we check whether the list, to which the element that we just inserted belongs, is empty or not. If it is not empty, it means that the list still has elements in it. So, in that case, we insert the next element from that particular list into our heap using the Min-Insert-Heap insert function but now we insert the following values: (i) the value of the next element from that particular list (KLists[ListId][IndexofValue+1]), the index of that list in the list of lists (ListId) and the new index of the new element that we just inserted using the Min-Heap-Insert function.

6. The while loop runs until all the lists in the KLists are empty and all the elements in those lists are inserted into our result list in sorted order.

7. This algorithm runs in O(nlogk) as the time taken by the Min-Heap-Insert and Min-Extract-Heap is O(logk) time each as there are a total of k elements in the heap at any given time and the while loop runs for each element in all the lists put together which is n. Therefore the running time will be O(nlogk).