Question 1

In order to apply the Masters theorem the given equation should be similar to the form T(n) = a*T(n/b) + f(n)
Where a and b are constants such that
a>=1 and b > 1
f(n) should be a function that is asymptotically positive growing with the values of n

(a)
$T(n)=8T(3n/2)+n^3$
a=8

b=2/3

f(n)=n^3

Since the value of b<1 we cannot apply the Masters theorem on this recurrence equation also we can see that the size of the sub problems is increasing with the increase in n in this equation.

(b)

$T(n)=4T(n/2)+n^2\text{sqrt}(n)$.

a=4

b=2

$f(n)=n^2\text{sqrt}(n)$

$n^{\log_b a} = n^{\log_2 4} = n^2$

$f(n) = n^2 * n^{1/2} = n^{5/2}, f(n) \in \Omega(n^{2+e})$ where e can be taken as 0.1, 0.2 0.5 etc

$f(n) \in \Omega(n^{log_b a+e}), e > 0$

We can see that this is the third case of the Masters Theorem

Hence we will need to check the regularity condition as well that is :

There should be a, c <1 and n0>=0

Such that af(n/b)<=c.f(n), for n > = n0

$4 * (n/2)^{5/2} <= c . n^{5/2} for c < 1$

1/sqrt(2) * n^(5/2) <= c.n^(5/2)

1/sqrt(2) = c

Hence we can see that case 3 should be applicable

Hence,

$T(n) = \theta f(n)$

So

T(n) = $\theta(n^{5/2})$

(c)

$T(n)=7T(n/3)+n^{11/5}$

a=7

b=3

f(n)=n^(11/5)

$n^{\log_b a} = n^{\log_3 7} < n^{\log_3 9}$

Hence ,

f(n)=n ^(11/5) > n^2

Hence we can say that

$f(n) \in \Omega(n^{log_b a+e}), e > 0$

$(n^{11/5}) \in \Omega(n^{log_3 7+e}), e > 0$

$n^{2.2} \in \Omega(n^{log_3 7+e}), where e is 0.43$

Since this is the case 3 of masters theorem we will check the regularity condition

There should be a, c <1 and n0>=0

Such that af(n/b)<=c.f(n), for n > = n0

7.(n/3)^(2.2) <= c. n^(2.2)

c=7/(3^2.2) which is less that 1

Hence the regularity condition applies well

So we can say that the complexity of the above recurrence equation is:

$T(n) = \theta f(n) = \theta(n^{11/2})$

(d)

T(n)=4T(n/2)+100-sqrt(n)

a=4
b=2
f(n)=100-sqrt(n)
When we substitute the value of n > 10000 in f(n) we can see that the value of f(n) decreases hence the function is not increasing and is not asymptotically positive.

This is why we cannot apply masters theorem to the above equation.

(e)
T(n)=½T(n/3)+sqrt(n)
a=1/2
b=3
Here we can see that value of a < 1 so we cannot apply the Masters Theorem.

Question 2

(a1)

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

Solving the question by the forward iteration method and let n=2^k

k=lg(2)

$T(1) = 1$

T(2) = 2T(1)+2lg(2) = 2+2lg(2)

T(4) = 2T(2)+4lg(4) = 2(2+2lg(2)) + 4lg(4) = 4+4lg(2)+4lg(4)

T(8)=2T(4)+8lg(8) = 2(4+4lg(2)+4lg(4)) + 8lg(8) = 8+8lg(2)+8lg(4)+8lg(8)

T(16)=2T(8)+16lg(16) = 2(8+8lg(2)+8lg(4)+8lg(8)) + 16lg(16) = 16 + 16lg(2) + 16lg(4) + 16lg(8) + 16lg(16)

.

.

.

$$T(2^k) = 2^k + 2^k lg(2) + 2^k lg(4) - - - - 2^k lg(2^k)$$

$$T(2^k) = 2^k(1 + lg(2) + lg(4) - - - - lg(2^k))$$

$$T(2^k) = 2^k(1 + lg2^1 + lg2^2 + lg2^3 - - - lg2^k)$$

$$T(2^k) = 2^k(1 + 1lg2 + 2lg2 + 3lg2 - - - klg2)$$

$$T(2^k) = 2^k * [1 + lg2(1 + 2 + 3 - - - k)]$$

$$T(2^k) = \frac{2^k * lg2 * k(k+1)}{2} + 2^k$$

$$T(2^k) = \frac{n * lg2 * lgn(lgn+1)}{2} + n$$

$$T(n) = O(n * lgn(lgn+1))$$

$$T(2^k) = O(n * (lg^2 n))$$

(a2)

$$T(n) = 2T(n/4) + \frac{\sqrt{n}}{logn}$$

Let us assume that n=4^k and (k=lgn—-> with base 4)

T(1) =1

$$T(4^{!}) = 2T(1) + \frac{\sqrt{4}}{log4} = 2 + \frac{\sqrt{4}}{1}$$

$$T(4^2) = 2T(4) + \frac{\sqrt{4^2}}{log4^2} = 2[2 + \frac{\sqrt{4}}{2^0}] + \frac{\sqrt{4^2}}{2^1} = 2^2 + \frac{2\sqrt{4}}{1} + \frac{\sqrt{4^2}}{2}$$

$$T(4^3) = 2T(4^2) + \frac{\sqrt{4^3}}{log4^3} = 2^3 + \frac{2^2\sqrt{4}}{1} + \frac{2^1\sqrt{4^2}}{2} + \frac{2^0\sqrt{4^3}}{3}$$

.
.
.
.
.
.

$$T(4^k) = 2^k + \frac{2^{k-1}\sqrt{4^1}}{1} + \frac{2^{k-2}\sqrt{4^2}}{2} + \frac{2^{k-3}\sqrt{4^3}}{3} - - - - - - \frac{2^{k-k}\sqrt{4^k}}{k}$$

$$2^k + \Sigma_{i=1}^{k} \frac{2^{k-i}\sqrt{(4^i)}}{i}$$

$$= 2^k + \Sigma_{i=1}^{k} \frac{2^k\sqrt{(4^i)}}{2^i * i}$$

$$= 2^k[1 + \Sigma_{i=1}^{k} \frac{\sqrt{(4^i)}}{2^i * i}]$$

$$= 2^k + \Sigma_{i=1}^{k} \frac{2^k\sqrt{(2^{2i})}}{2^i * i}$$

$=2^k + \Sigma_{i=1}^{k} \frac{2^k * 1}{i}$

$=2^k[1 + \Sigma_{i=1}^{k} \frac{1}{i}]$

Substituting, k = lg n where the base of lg is 4

$\sqrt{n}[1 + \Sigma_{i=1}^{lgn} \frac{1}{i}]$

$=O(\sqrt{n}[1 + lg(lg(n)])$ where lg n has a base of 4

(b) The recurrence in lecture 6, page 7. Use T(1)=0

$$T(n) = \frac{n+1}{n} T(n-1) + c\frac{2n-1}{n}$$

T(1)=0

$$\text{T(2)} = \frac{2+1}{2} T(1) + c\frac{3}{2} = c\frac{3}{2}$$

$$\text{T(3)}=\frac{4}{3} T(2) + c\frac{5}{3} = \frac{4}{3} * \frac{3}{2} c + c\frac{5}{3}$$

$$\text{T(4)}= \frac{5}{4}[c\frac{4}{3} * \frac{3}{2} + c\frac{5}{3}] + c\frac{7}{4}$$

$$= c * \frac{5}{2} + c * \frac{25}{12} * c * \frac{7}{4}$$

Taking 5 common from the entire LHS

$=c * 5[\frac{1}{2} + \frac{5}{12} * \frac{7}{20}]$

$=c * 5[\frac{3}{2*3} + \frac{5}{3*4} * \frac{7}{4*5}]$

$T(n) = c * (n+1)\Sigma_{i=2}^{n} \frac{2i-1}{i(i+1)}]$

$T(n) <= c * (n+1)\Sigma_{i=2}^{n} \frac{2i}{i(i+1)}]$ ——-> removing the -1 and change to <=

$T(n) <= c * (n+1)\Sigma_{i=3}^{n+1} \frac{2}{i)}]$

$2 * c * (n+1)\Sigma_{i=3}^{n+1} \frac{1}{i}]$

$=2 * c * (n+1)[lg(n+1) - lg(3)]$

$= \theta(nlgn)$

(c) Assume c>1. T(8)=…=T(0)=c, T(n) = 3T(n-2)+2n-1 otherwise. Give the exact solution and the asymptotic big-theta bound for T(n).

T(0)=c
T(1)=c
T(2)=c
.
.
.
.
T(8)=c———————> given in question
T(9)=3.T(7)=3.c

T(10)=3.T(8)=3.c
T(11)=3.T(9)=3.3.c
T(12)=3.T(10)=3.3.c
T(13)=3.T(11)=3.3.3.c
T(14)=3.T(12)=3.3.3.c
.
.
.
.
So now if we see that for n=0 to 8 the value is 3^0.c

For n=9 and n=10 the value is 3^1.c

9-8=1 —-> 1/2 on dividing by 2 = 0.5 =$ceil(0.5)$=1
10-8=2 —-> 2/2 on dividing by 2 =1 =$ceil(1)$=1

For n=11 and n=12 the value is 3^2.c

11-8=3— —> 3/2 on dividing by 2 = 1.5=$ceil(1.5)$= 2
12-8=4— —-> 4/2 on dividing by 2 = 2=$ceil(2)$=2

For n=13 and n=14 the value is 3^3.c

13-8=5— —>5/2 on dividing by 2 = 2.5 =$ceil(2.5)$=3
14-8=6— —>6/2 on dividing by 2 = 3=$ceil(3)$=3

Hence we can say that
T(n) = c. $3^{\lceil \frac{n-8}{2} \rceil}$ for all the n > 8
T(n) = c for all n = 1 to 8

3. Purpose: Often, recursive function calls use up precious stack space and might lead to stack overflow errors. Tail call optimization is one method to avoid this problem by replacing certain recursive calls with an iterative control structure. Learn how this technique can be applied to QUICKSORT. (2 points each) Solve Problem 7-4, a-c on page 188 of our textbook.

a.)
Argue that TAIL-RECURSIVE-QUICKSORT(A,1,A.length) correctly sorts the array A.

We can say that TAIL-RECURSIVE-QUICSORT(A, A.length) correctly sorts the sorts the array A because of the following factors:

———-

Actual Algorithm

```
QUICKSORT(A,p,r)
 1 if p<r
2       q=PARTITION(A,p,r)
3       QUICKSORT(A; p; q )
4       QUICKSORT(A; q+1, r)
```

———

Modified Algorithm

```
TAIL-RECURSIVE-QUICKSORT(A,p,r)
1 while (p<r)
        //Partition and sort left subarray.
2       q=PARTITION(A,p,r)
3       TAIL-RECURSIVE-QUICKSORT(A,p,q)
4       p=q+1
```

We can see that the difference in the 2 algorithms is that in actual Quick Sort there is an if condition on line 1 however, there is a while loop in the Modified Algorithm.

In the actual quick sort algorithm the array is partitioned in 2 parts by finding out a partition element and then there are recursive quick sort calls on the 2 sub arrays.

Further, in the modified quick sort when the partition is done the quick sort function is recursively called for left sub array.

However, when all the recursive calls for the left sub array are over there will be another iteration using the while loop where the quick sort will be done for the right part as this time the value of the p will become as q+1 that will be making recursive calls for the right sub array.

Hence the , difference in line 4 and line 2 justifies that the TAIL-RECURSIVE-QUICKSORT algorithm should work perfectly.

b)

Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\theta(n)$ on an n-element input array.

A scenario when the TAIL-RECURSIVE-QUICKSORT's stack depth will be $\theta(n)$ is when we give this algorithm a completely sorted array. In that case the always the pivot element will be the last array of the element after partition algorithm. Further we will not have to make any recursive calls for the right sub arrays as they will be only one element sub arrays.

Hence this way the calls will be like :

TAIL-RECURSIVE-QUICKSORT(A,p,n)
TAIL-RECURSIVE-QUICKSORT(A,p,n-1)
TAIL-RECURSIVE-QUICKSORT(A,p,n-2)
TAIL-RECURSIVE-QUICKSORT(A,p,n-3)
TAIL-RECURSIVE-QUICKSORT(A,p,n-4)
.
.

.
.
.
TAIL-RECURSIVE-QUICKSORT(A,p,1)

C)

Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is, $\theta(logn)$ Maintain the $\theta(nlogn)$ expected running time of the algorithm.

The depth of the TAIL-RECURSIVE-QUICKSORT will be $\theta(logn)$ when the partition element always comes out to be the middle element and the array is divided into 2 arrays when with almost equal halves.

Hence there are 2 ways that I could think of in which the Algorithm can be modified:

We can use Randomized Quick Sort Algorithm which will always ensure that the partitions are being made in such a way that the depth of the stack is log(n).

Using this Randomized quick sort algorithm will also ensure average case complexity for the sorting to be of (n log n)

In case we use the Randomized Quick Sort the only difference that will be made is that the partition will be randomly done.

Same has been explained on Page 179 of our text book.

“In exploring the average-case behavior of quicksort, we have made an assumption that all permutations of the input numbers are equally likely. In an engineering situation, however, we cannot always expect this assumption to hold. We could do so for Quicksort also, but a different randomization technique, called *random sampling*, yields a simpler analysis”

———————————————————————————————

Further another way to ensure that the stack size is maintained to log n is by always ensuring that the recursive call is done on the smaller sub-array. This way we can ensure that the height of the stack while recursive calls grows unto only log(n)

We can modify the TAIL-RECURSIVE-QUICKSORT as follows to achieve the same:

```
MODIFIED-TAIL-RECURSIVE-QUICKSORT(A,p,r)
1 while (p<r)
2       q=PARTITION(A,p,r) //Partition
3       if you q<(r-p)/2   //if the left subarray is small sort that first
4               MODIFIED-TAIL-RECURSIVE-QUICKSORT(A,p,q-1)
5               p=q+1
6       else //sort the right sub array
7               MODIFIED-TAIL-RECURSIVE-QUICKSORT(A,p,q)
8               r=q-1
```

4.

Consider a situation where your data is almost sorted—for example you are receiving time-stamped stock quotes and earlier quotes may arrive after later quotes because of differences in server loads and network traffic routes. Focus only on the time-stamps. Assume that each time-stamp is an integer, all time-stamps are different, and for any two time-stamps, the earlier time-stamp corresponds to a smaller integer than the later time-stamp. The time-stamps arrive in a stream that is too large to be kept in memory completely. The time-stamps in the stream are not in their correct order, but you know that every time-stamp (integer) in the stream is at most hundred positions away from its correctly sorted position. Design an algorithm that outputs the time-stamps in the correct order and uses only a constant amount of storage, i.e., the memory used should be independent of the number of time-stamps processed. Solve the problem using a heap.

Answer:

In this situation using a heap data structure would be very helpful. We can use a min heap as we want the data to be coming in ascending order that are time Stamps.
Hence, we can implement a Priority Queue using a MIN HEAP data structure.

We will make a min heap that can have a maximum size of 101 because, it is stated in the question that an unsorted time stamp will be there at most hundred positions away from the correctly ordered element. So the time stamp if it comes late will have its correct place approximately 100 elements at the back.

And in the Priority Queue we can use the extract min function to take out the minimum value.

The Algorithm should be as follows:

-> when time stamps are coming
->insert the incoming time stamp value on the heap
->If the size of heap is more than 100 then we extract the minimum
->if the time stamps stop arriving then we just empty the min heap using extract element

The pseudo code for the algorithm should be like below:

```
1 number_of_elements = 0
2 While True:

3       if time_stamp comes:
4          Insert_in_heap (Heap, time_stamp).    //function to insert element in heap
5          number_of_elements = number_of_elements + 1 //count elements in heap
6          if (number_of_elements > 100)
7                remove_element_from_heap(Heap). //prints the min element in heap

8       if time_stamp not coming:
9         while (number_of_elements > 0)
10              remove_element_from_heap(Heap)
11              number_of_elements = number_of_elements - 1
```

```
12        break  // leave the final while loop as all the task has ended

13 Insert_in_heap (Heap, time_stamp)
14      add the element to the last index of the heap
15      size_of_heap+=1
16      heapify_from_down(size_of_heap)

17 heapify_from_down(size_of_heap)
18      while(i%2 >)
19      if heap(i)>heap(i%2)
20              swap heap(i), heap(i%2)
21      i=i%2

22 remove_element_from_heap()
23      remove the topmost element of the heap
24      bring the last element of the heap to the top
25      call the min-heapify function to make it a heap again

16 min_heapify(i)
17 l=2*i+1
18 r=2*i+2
19 If l <=size_of_heap and heap[l]<heap[i]
20      small=l
21 Else
22      small=i
23 If r<=size_of_heap and heap[smallest]>heap[r]
24      smallest=r
25 If smallest!=i
26      exchange heap[i] and heap[smallest]
27      min_heapify(smallest)
```

How the Algorithm is correct :
Using Loop Invariants:

Initialization:

When there are no elements in the heap there is no element being outputted.
Further, no element is outputted until the heap size becomes more than 100.

Maintenance:

We can say that when an element a is being in output using the remove_element_from_heap(Heap) function, there can not be an element b that comes which is less than a.

As it is a condition that they correct place of an element can be only approximately hundred places back.

So lets say if a time 2 comes and the 99 other time stamps come that are greater than 2 like 3, 4 , 5 , 6 , 7 etc.

All these numbers will be places in the heap till there are 100 elements.

Now lets say on the 101th position again 2 comes again 2 this time when 2 comes one 2 will be another 2 will be inserted in the heap.

Next it will be checked that the heap is bigger than hindered and the old 2 will be removed.

Also ideally according to the questions statement the time stamp 2 should not have come later than 100 elements to maintain the correctness of the Algorithm.

This is how the algorithm will perform perfectly.

Termination:

At the termination of the loop all the elements will maintain a correct order.

Complexity of the Algorithm

The heap that is being maintained is of constant size that is 100 elements.

Further the height of the Heap will also be log(100) hence all the operations :

Insertion, extracting of the element etc will take a constant time that is log(100)