WRITTEN ASSIGNMENT - 2.

① The pseudo-code to find out k small elements in an array of length 'n' ~~can be~~ is as follows:

```
K_small_elements (arr, K):
    K_small_heap = arr [0:K]
    Max Heapify ( arr, 0)

    for i in range ( K, len (arr)):
        if (arr [K] < small_elements[0]):
                small_elements [0] = arr [K]
                Max Heapify (arr, 0)
```

Pseudo-code for max Heapify :

Inputs: Array A (or binary tree)
index i in array.

Precondition: The binary trees rooted at Left(i) and Right(i) are heaps. Note: A[i] may be smaller than its children.

Post condition: The subtree rooted at index i is a heap.

MaxHeapify (A, i):

l = LEFT(i)

r = RIGHT(i)

if l ≤ len(A) and A[l] > A[i]:

largest = l.

else: largest = i

if r ≤ len(A) and A[r] > A[largest].

largest = r.

if largest != i:

exchange A[i] with A[largest].

~~max chec~~

MaxHeapify (A, largest).

Analysis of The Algorithm:

① The algorithm uses an additional array of size 'K' to store the 'K' small elements: So, The space complexity is O(K) [additional].

② Run-time complexity:

The algorithm has a run-time complexity of $O((n-K)\log(n)) \approx O(n\log(K))$ for sufficiently large values of 'n'.

Proof:

→ It takes log(K) times to heapify an array of size 'K'.

→ The algorithm heapifies the array (n-K) times.

So, run-time complexity is $O((n-K)\log(K)) \approx O(n\log(K))$.

2) ① Given recurrence relation, $T(n) = 2T(n/4) + 1$.

Analytically, we can break-down the above recurrence as follows:

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

$T(n/4) = 2T(n/16) + 1$

$T(n/16) = 2T(n/64) + 1$

$\vdots$

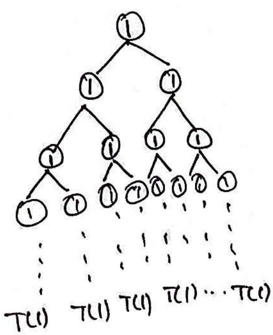
K^{th} Term $= T(n/4^{K-1}) = 2T(n/4^K) + 1$. ⟶ Generalization.

The above recurrence would converge when $\boxed{n/4^K = 1}$.

So, when, $K = \log_4 n$, the above recurrence would converge.

Therefore, the algorithm described by the given recurrence relation would run for $\boxed{K = \log_4 n}$ times.

We can visualize the above recurrence as follows:

	Input	Tree	Work
$K=0$	n		1
$K=1$	$n/4$		2
$K=2$	$n/16$		4
$K=3$	$n/64$		8
$\vdots$	$\vdots$		$\vdots$
$K = \log_4 n$	$n/4^{\log_4 n} = 1$		$2^{\log_4(n)} = \underline{\underline{2^K}}$

Sum of work,

$T(n) = 2^K T(1) + 2^{K-1} + 2^{K-2} + \cdots + 1$

$= \dfrac{2^{K+1} - 1}{2 - 1} = 2^{K+1} - 1$

$= 2(2^K) - 1 = 2\sqrt{n} - 1$, $\therefore \boxed{T(n) = \Theta(\sqrt{n})}$

② ② Given recurrence relation, $T(n) = 2T(n/2) + n^2$.

The above recurrence can be broken down as follows:

$$T(n) = 2T(n/2) + n^2$$
$$T(n/2) = 2T(n/4) + (n/2)^2$$
$$\vdots$$
$$K^{th}\text{ term} = T(n/2^{K-1}) = 2T(n/2^K) + (n/2^{K-1})^2$$

The above recurrence would converge when, $\frac{n}{2^K} = 1, (or)$ when $K = \log_2(n)$. $[\ T(n/2^K = n/2^{\log_2 n} = n/n = 1\ ]$.

we can visualize the above recurrence as follows:

K	Input/Nodes	Tree	Work
			n^2
1	1	n^2	$n^2/2$
2	2	$(n/2)^2 \quad (n/2)^2$	$n^2/4$
3	4	$(n/4)^2 \quad (n/4)^2 \quad (n/4)^2 \quad (n/4)^2$	$\vdots$
4	8	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$T(1) \cdots T(1)$	$(n/2^{K-1})^2 \times 2^{K-1}$
	2^{K-1}		
K			

$\therefore$ Height of the tree $= \log_2(n)$.

when, $K = \log_2(n)$, work $= \frac{n}{2^{K-1}} = \frac{2n}{2^K} = 2n = \Theta(n)$

~~So we can approximate the run time complexity as~~

The total work can be written as:

$$T(n) \approx \theta(n) + c\left(n^2 + \frac{n^2}{4} + \frac{n^2}{4} + \cdots\right)$$

$$\approx \theta(n) + n^2\left(1 + 1/2 + 1/4 + \cdots\right).$$

$$\approx \theta(n) + n^2(c).$$

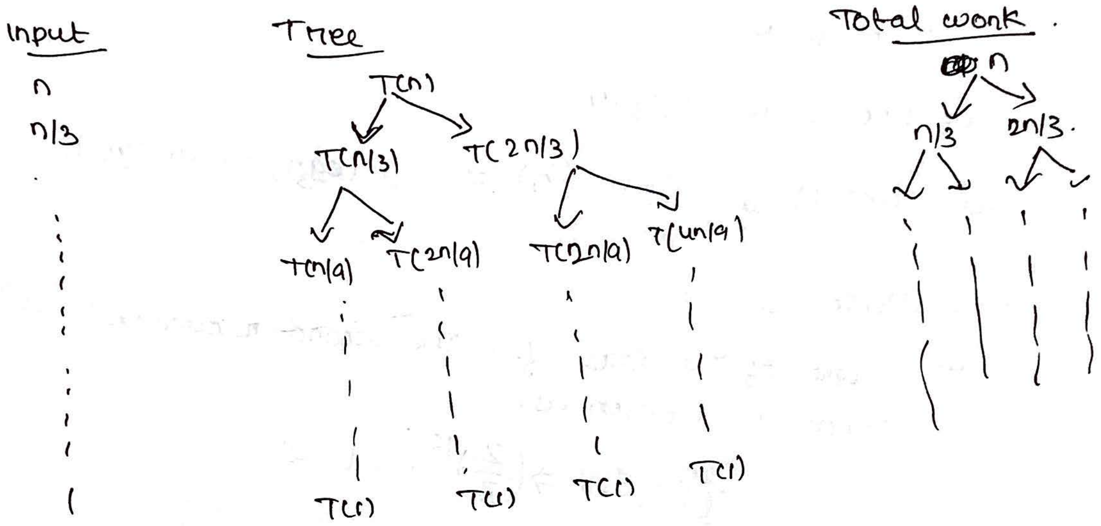
where, c = constant.

$$\Rightarrow \boxed{T(n) = \theta(n^2)}$$

② ③ Given recurrence relation, $T(n) = T(n/3) + T(2n/3) + n$.

(a) There are two sub-recurrences in this recurrence relation.

Let's visualize the recurrence as a Binary-tree.



(b) There could be two cases for termination.

1st case: (Lower Bound)

Height of the tree $n/3^k = 1 \Rightarrow \boxed{k = \log_3(n)}$.

Total work for each node:

$k = 1$, work $= n$.

$k = 2$, work $= n/3 + 2n/3 = 1$.

⋮

when $K = k$, work $= \sum_{j=0}^{k} \left({}^{k}C_j\right) \left(\frac{1}{3}\right)^{k-i} (2/3)^{i} = \left(\frac{1}{3} + \frac{2}{3}\right)^{k} n$
$= \underline{n}$

~~we see that total~~

The work done for each value of 'k' is (n). So, the total work for 'k' steps is $\boxed{kn}$.

$$T(n) \approx kn.$$

we know, $K = \log_3(n)$.

So, $T(n) \approx \log_3(n)\,(n) = \Theta(n \log_3 n) = \Theta(n \log n)$.

Case-2: (Upper Bound)

The height of the tree for the second recurrence $T(2n/3)$ can be written as,

$$\left(\frac{2}{3}\right)^K n = 1$$

$$\Rightarrow \frac{n}{(3/2)^K} = 1.$$

$$\Rightarrow$$

∴ when, $\boxed{K = \log_{3/2} n}$, the recurrence would terminate.

∴ The total running time can be written as –

$$T(n) \approx Kn.$$

$$\approx n \log_{3/2}(n)$$

$$\approx \Theta(n \log(n))$$

∴ The average time complexity of $T(n) = T(n/3) + T(2n/3) + n$ is $\Theta(n \log(n))$.

3) In this sorting algorithm we shuffle the elements in an array 'a' and run deterministic quicksort on the randomly shuffled version of 'a'.

For example,

if a = [1,2,3], after random shuffling we get one of - [1,2,3], [3,2,1], [1,3,2], [2,1,3], [2,3,1], [3,1,2].

(*) Each of the random arrangements are equally probable. So, when we run deterministic quick sort on any arrangement of 'a', it's equally probable that we choose any of the elements in 'a' as the pivot. This is similar to randomized quick sort where we choose a random element as the pivot

(*) The partition function of randomized Quicksort is as follows:

Pseudo-code:

```
Randomized-partition (A,p,r).
i = random(p,r).
exchange A[r] and A[i].
return partition (A, p,r).
```

Referred from CLRS.

Here we randomly choose an index between 'p' and 'r', and swap the last element A[r] with the random element of array, A[i].

⊛ In our implemention of deterministic Quick sort, we select the first element (or) last element as the pivot. But, the input to this quick sort algorithm is a random arrangement of elements in input array, (a). So, any element of original a could be the pivot in our implementation of Quick sort.

⊛ Worst case:

When we choose the smallest (or) largest element as the pivot, (or) when the ~~random arrangement generates an arrangement in case in~~ elements are arranged in sorted order.

So, our implementation of deterministic Quicksort is similar to randomized quick sort where, any element could be the pivot. Even the selection of partition index is similar. Additionally, the worst-case scenario for randomized quick sort and the current quick sort is also similar.

So, we can relate the run time complexity of randomized Quick sort with the average runtime of our sorting algorithm.

so, runtime (average) = $O(n\log(n))$.