Assignment - 2

Answer to the question No. 1

1. Divide-and-Conquer algorithm can be divided into three parts. That are –
   Divide: Divide the array of n distinct numbers into two sub-arrays – array[a1, a2,…, a(n/2)] and array[a((n/2)+1), …, an] where each of them has size n/2.
   Conquer: Recursively solve each of the subproblems. The base case will be the subproblem with size 1 and index of that element will be returned.
   Combine: Now put the solution for each of the subproblems together to obtain a solution for the original problem. So, from the two subproblem solved individually, we get the index s1 and s2 of the smallest element of both. We then compare the elements at the indices s1 and s2. If the element at index s1 is smaller, return s1. Otherwise, return s2.

   Considering how Divide-and-Conquer algorithm works, the pseudocode for finding the index of smallest number from an unsorted array is written below:

```
findMinIndex (array, left, right) {
        if (left == right) return left;

        mid = (left + right) / 2;

        leftMinIndex = findMinIndex(arr, left, mid);
        rightMinIndex = findMinIndex(arr, mid+1, right);

        if (arr[leftMinIndex] < arr[rightMinIndex]) return leftMinIndex;

        else return rightMinIndex;
        }
```

   In the above pseudocode, findMinIndex is a recursive function that takes an array, start and end index value as left and right, respectively. The first if statement checks if there is only one element in the array. When the condition is false, it splits the array into two halves and find the minimum in each half. Finally, return the index of the smallest number which is found between the subarrays using divide-and-conquer algorithm.

   Recurrence relation for the running time of the above algorithm is,

$$T(n) = \begin{cases} 1, & if\ number\ of\ element, n = 1 \\ 2T(n/2) + 1, & otherwise \end{cases}$$

   Now, Assume $\theta(1) = \text{p}$, some constant and using substitution method we can write,

$$\begin{aligned} T(n) &= 2\,T(n/2) + 1 \\ &= 2[\,2\,T(n/4) + 1] + 1 = 4\,T(n/4) + 2 + 1 \\ &= 4[\,2\,T(n/8) + 1] + 2 + 1 = 8\,T(n/8) + 4 + 2 + 1 \\ &\dots \end{aligned}$$

$= 2^k T(n/2^k) + (2^k – 1) 1$

Here, $n/2^k = 1$. So, $k = \log_2 n$.

So, T(n) = n + (n-1) = 2n – 1. For large value of n, T(n) = θ(n).

Proof by Induction:

Guess: The algorithm takes θ(n) time. To prove this we need to find O(n) and Ω(n) from the recurrence relation, T(n) = 2 T(n/2) + 1.

We know, T(n) = O(n) when, T(n) $\leq$ cn, for some constant c > 0 and n $\geq$ n_0 for some $n_0 > 0$ and T(n) = Ω(n) when, T(n) $\geq$ cn for some constant c > 0 and n $\geq$ n_0 for some $n_0 > 0$.

Base Case: When n = 2, the run-time is a constant, which is 3. And 3 $\leq$ 2c, for any c $\geq$ 3/2; that is T(n) = O(n).

Again, when n = 2, the run-time is a constant, which is 3. And 3 $\geq$ 2c, for any c $\leq$ 3/2; that is T(n) = Ω (n).

Hence, the base case holds.

Proof: To prove T(n) = O(n), Let's assume T(k) $\leq$ ck for any k < n and c is a constant and c > 0.

Now, T(n) = 2 T(n/2) + 1 $\leq$ 2 * c * n/2 + 1 = cn + 1 = cn + c + 1 – c = c(n+1) + (1-c)

T(n) $\leq$ c (n+1) for 1-c < 0 or c > 1.

So T(n) = O(n) where c is a constant and c > 1. ------------------ (i)

Again, To prove T(n) = Ω(n), Let's assume, T(k) $\geq$ ck for any k < n and c is a constant and c > 0.

Now, T(n) = 2 T(n/2) + 1 $\geq$ 2 * c * n/2 + 1 = cn + 1.

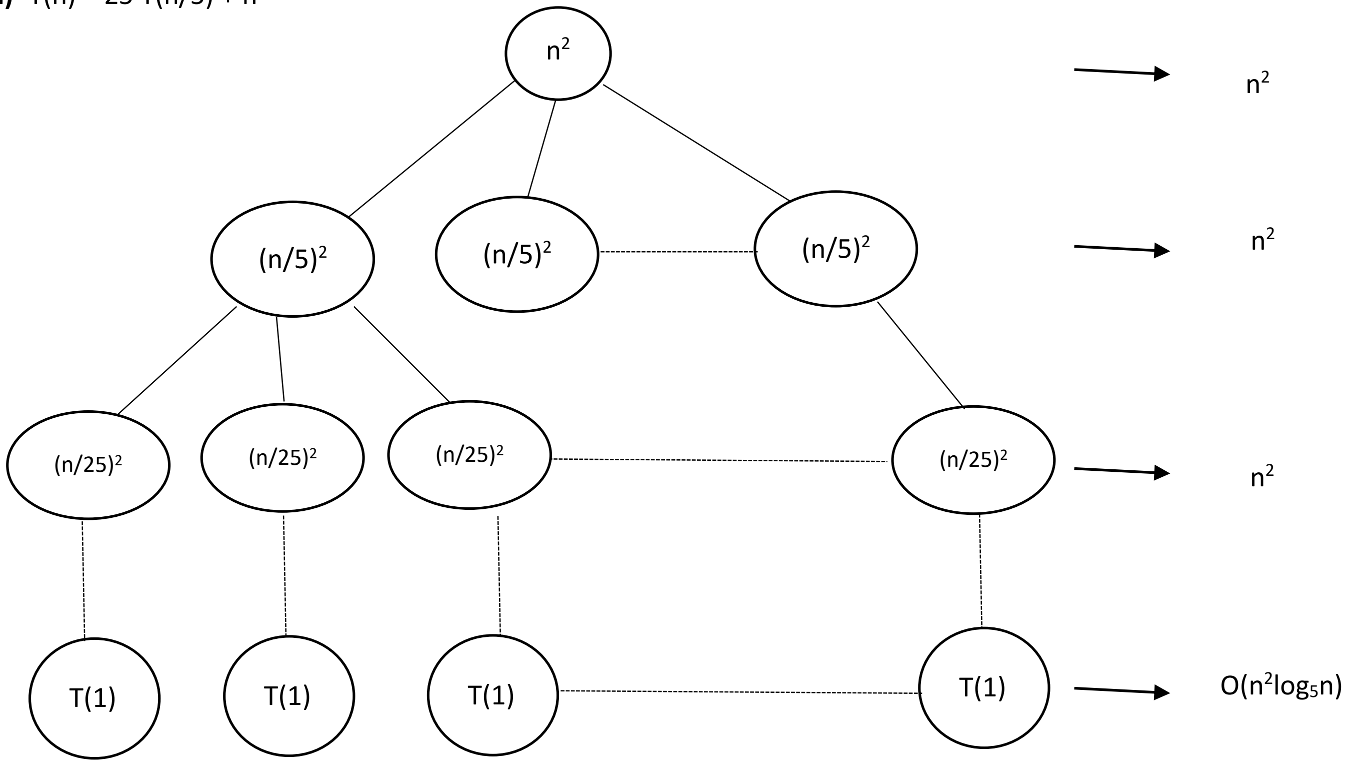
T(n) $\geq$ cn for c > 0

So T(n) = Ω(n) where c is a constant and c > 0. ---------------- (ii)

From i and ii we get T(n) = θ(n). (Proved)

Answer to the question No. 2

(a) $T(n) = 25\,T(n/5) + n^2$



At each level the cost is n^2 and the recursion continues until the problem size becomes 1. So, we get:

$$\frac{n}{5^k} = 1$$

$$5^k = n$$

Taking the logarithm base 5 of both sides:

$$k = \log_5 n$$

Thus, the total number of levels in a recursion tree is $\log_5 n$.

Cost of internal nodes = $n^2 + 25.\ n^2/25 + 25^2.\ n^2/25^2 + 25^3.\ n^2/25^3 + \ldots.$

$= n^2 + n^2 + n^2 + n^2 + \ldots.$

$= n^2 .\ k = n^2 \log_5 n$

Cost of leaf node = $5^k = 5^{\log_5 n} = n^{\log_5 5} = n$

Total cost = Cost of internal nodes + Cost of leaf node = $n^2 * \log_5 n + n$

So, runtime complexity is $O(n^2 \log_5 n)$.

Proof by Induction:

Guess: The guess for the running time of the algorithm is O($n^2 \log_5 n$). So, $T(n) \leq c\, n^2 \log_5 n$ when some constant $c > 0$ and for all $n \geq n_0$ and $n_0 > 0$.

The recurrence relation is, $T(n) = 25\, T(n/5) + n^2$.

Base Case: For simplicity let's prove for n = 5

$T(5) = 25\, T(1) + 25 = 50 \leq c * 25 * \log_5 5 = 25c$

So, the recurrence relation is true for $c \geq 2$.

Proof: Assume, $T(k) \leq c\, k^2 \log_5 k$ for $k < n$.

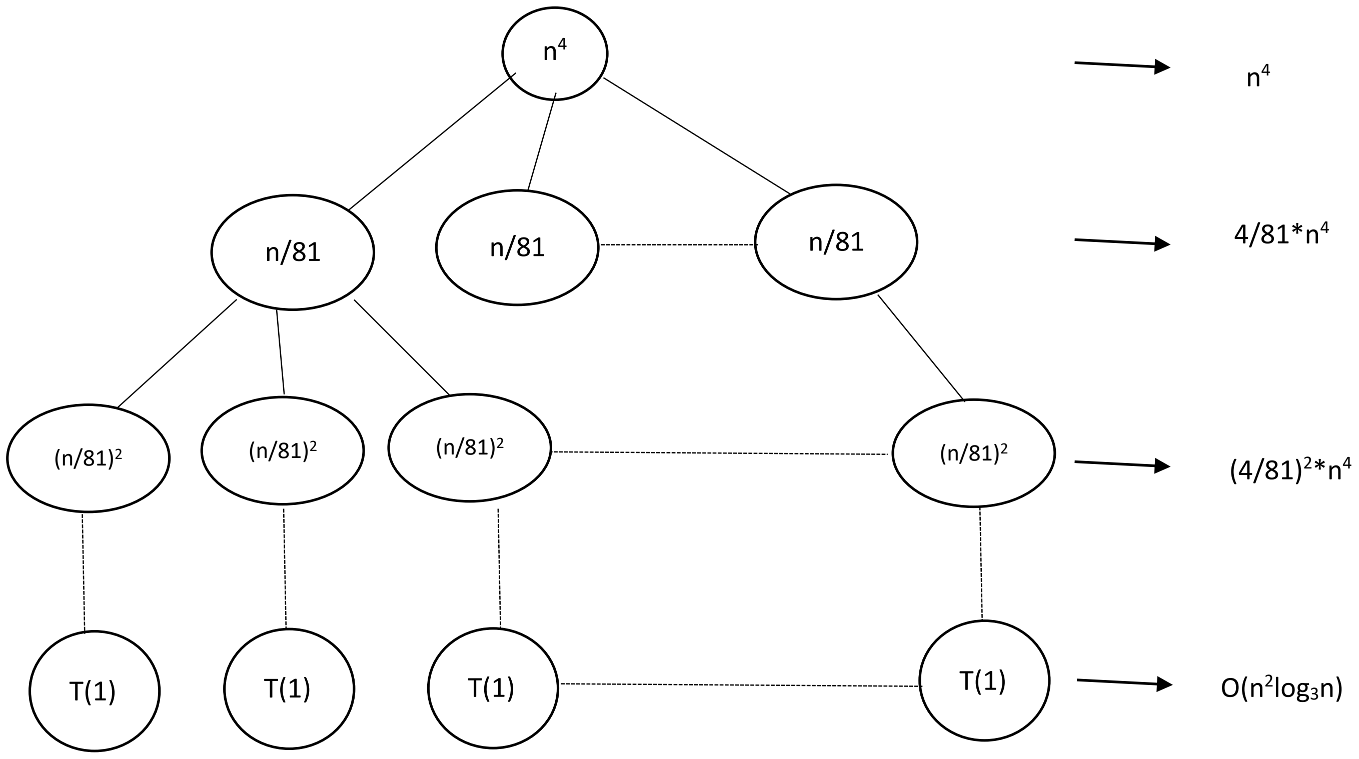
Now, $T(n) = 25\, T(n/5) + n^2 \leq 25 * c * (n/5)^2 \log_5(\frac{n}{5})\ n^2 = 25 * c * n^2/25 * (\log_5 n - \log_5 5) + n^2$

$= cn^2 \log_5 n - (cn^2 - n^2)$

So, $T(n) \leq cn^2 \log_5 n$ for $(cn^2 - n^2) \geq 0$ or $c \geq 1$.

We get, $T(n) = O(n^2 \log_5 n)$ when $c \geq 2$. (Proved)

(b) $T(n) = 4\, T(n/3) + n^4$.



At each level the cost is n^2 and the recursion continues until the problem size becomes 1. So, we get:

$$\frac{n}{3^k} = 1$$

$$3^k = n$$

Taking the logarithm base 3 of both sides:

$$k = \log_3 n$$

Thus, the total number of levels in a recursion tree is $\log_3 n$.

Cost of internal nodes = $n^4 + (4/81) * n^4 + (4/81)^2 * n^4 + (4/81)^3 * n^4 + \ldots$

$= (4/81)^0.n^4 + (4/81)^1. n^4 + (4/81)^2. n^4 + (4/81)^3. n^4 + \ldots.$

$= n^4 . \{(4/81)^0 + 4/81)^1 + (4/81)^2 + (4/81)^3 +\ldots. \}$

$= n^4 . 1/(1-4/81) = (81/77)\ n^4$

Cost of leaf node = $4^k = 4^{\log_3 n} = n^{\log_3 4}$

Total cost = Cost of internal nodes + Cost of leaf node = $n^{\log_3 4} + (81/77)\ n^4$, $\log_3 4$ is less than 2. So, $n^{\log_3 4}$ is less than n^4

So, runtime complexity is $O(n^4)$.

Proof by Induction:

Guess: The guess for the running time of the algorithm is $O(n^4)$. So, $T(n) \leq c\ n^4$ when some constant $c > 0$ and for all $n \geq n_0$ and $n_0 > 0$.

The recurrence relation is, $T(n) = 4\ T(n/3) + n^4$.

Base Case: For simplicity let's prove for n = 3

$T(3) = 4\ T(1) + 81 = 85 \leq c * 81 = 81c$

So, the recurrence relation is true for $c \geq 85/81$.

Proof: Assume, $T(k) \leq c\ k^4$ for $k < n$.

Now, $T(n) = 4\ T(n/3) + n^4 \leq 4 * c * (n/3)^4 + n^4 = 4 * c * n^4/81 + n^4$

$= (4/81)cn^4 + n^4 = cn^4 + n^4 + (4/81)\ cn^4 - cn^4$

$= cn^4 + n^4 \{1- (77/81)\ c\}$

So, $T(n) \leq cn^4$ if $c \geq 81/77$.

We get, $T(n) = O(n^4)$ when $c \geq 81/77$. (Proved)

Answer to the Question No. – 3

From the given description we can write, $T(n) = x\,T(n/3) + O(\log n)$ using Master Theorem.

Here, $a = x$, $b = 3$ and $f(n) = \log n$. We need to determine the maximum value of a such that $T(n) = o(n^2)$.

As $f(n) = \log n$, we need to compare it with $n^{\log_3 x}$.

To ensure $T(n) = o(n^2)$, $\log_3 x < 2$

$\Rightarrow x < 9$

So maximum 8 subproblems of size n/3 can be taken.