**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,…, an/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,

**Proof by induction:**

Guess algorithm takes θ(n) time.

Now, T(n) = O(n) when, T(n) ≤ cn, for some constant c > 0 and n ≥ n0 for some n0 > 0.

Assume T(k) ≤ ck for k<n. -------------------------------------------------------------------------------(i)

Now using the recurrence relation, T(n) = 2 \* T(n/2) + 1 ---------------------------------------(ii)

Substituting this in (i) we get, T(n) = 2 \* T(n/2) + 1 ≤ 2 \* c \* (n/2) + 1 = cn + 1

Excluding the constant parts, we get T(n) = O(n). ------------------------------------------------(iii)

Now, T(n) = Ω(n) when, T(n) ≥ cn for some constant c > 0 and n ≥ n0 for some n0 > 0.

Assume T(k) ≥ ck for k<n. -------------------------------------------------------------------------------(iv)

Now substituting (ii) in (i) we get, T(n) = 2 \* T(n/2) + 1 ≥ 2 \* c \* (n/2) + 1 = cn + 1

Excluding the constant parts, we get T(n) = Ω(n). ------------------------------------------------(v)

From (iii) and (v) it is proved that T(n) = θ(n).

Answer to the question No. 2.

1. T(n) = 25 T(n/5) + n2

n2

n2

n2

O(n2log5n)

At each level the cost is n2 and the recursion continues until the problem size becomes 1. So, we get:

Taking the logarithm base 5 of both sides:

Thus, the total number of levels in a recursion tree is .

Total cost = n2 \*

Proof by Induction:

The guess for the running time of the algorithm is θ( n2 .

Assume, T(k) ≤ c k2  for k < n.

The recurrence relation is, T(n) = 25 T(n/5) + n2.

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

T(5) = 25 T(1) + 25 = 50 ≤ c \* 25 \* = 25c

So, the recurrence relation is true for c ≥ 2.

Now, T(n) = 25 T(n/5) + n2 ≤ 25\* c \* (n/5)2  n2 = 25 \* c \* n2/25 \* + n2

= cn2 – (cn2 - n2)

So, T(n) ≤ cn2 for (cn2 - n2) ≥ 0 or c ≥ 1.

We get, T(n) = O(n2

Again assume, T(k) ≥ c k2  for k < n.

Now, T(n) = 25 T(n/5) + n2 ≥ 25\* c \* (n/5)2  + n2 = 25 \* c \* n2/25 \* + n2

= cn2 – (cn2 - n2)

So, T(n) ≥ cn2 for (cn2 - n2) ≤ 0 or c ≤ 1.

We get, T(n) = Ω(n2

From (1) and (2) we get, θ( n2 .

1. T(n) = 4 T(n/3) + n4.

n4

4/81\*n4

(4/81)2\*n4

O(n2log5n)

At each level the cost is n2 and the recursion continues until the problem size becomes 1. So, we get:

Taking the logarithm base 5 of both sides:

Thus, the total number of levels in a recursion tree is . So, 4k =

Total cost = n4 + (4/81) \* n4 + (4/81)2 \* n4 + (4/81)3 \* n4 + … + \* n4

= n4 (1 + (4/81) + (4/81)2 + (4/81)3 + … + ) = n4 \* 1/{1-(4/81)} = 81/77 \* n4

The guess for the running time of the algorithm is θ( n4.

Proof by Induction:

Assume, T(k) ≤ c k4 for k < n.

The recurrence relation is, T(n) = 4 T(n/3) + n4.

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

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

So, the recurrence relation is true for c ≥ 85/81.

Now, T(n) = 4 T(n/3) + n4 ≤ 4 \* c \* (n/3)4 + n4 = 4 \* c \* n4/81 + n4 = (4/81)cn4 + n4 = n4 {(4/81)c + 1}

So, T(n) ≤ cn4 for sufficiently large c.

We get, T(n) = O(n4

Again assume, T(k) ≥ c k4 for k < n.

Now, T(n) = 4 T(n/3) + n2 ≥ 4 \* c \* (n/3)4 + n4 = 4 \* c \* n4/81 + n4

= cn4 + n4

So, T(n) ≥ cn4 for sufficiently large n and c>0.

We get, T(n) = Ω(n4

From (1) and (2) we get, θ( n4.

Answer to the Question No. – 3.

From the given description we can write, T(n) = x T(n/3) + O(logn).

Here, a = x, b = 3 and f(n) = logn.

As, T(n) = o(n2) so f(n) = logn is not going to dominate T(n) and it’s bound is going to be determined by o(n2).

Using master method we get, .

⇒ < 2

⇒ x < 9

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