Question 1.

(a) Write a linear time divide and conquer algorithm (i.e., $\theta(n)$) to calculate x^n (x is raised to the power n). Assume x and n are integers >=0. Note: your algorithm here should be of $\theta(n)$.

(b) Analyze the time complexity of your algorithm in the worst-case by first writing its recurrence relation. Show why it is $\theta(n)$.

(c) Can you improve your algorithm to accomplish the end in O(log n) time complexity (we still look for a divide and conquer algorithm). If yes, write the corresponding algorithm, write the recurrence relation for its time complexity and analyze it. If no, justify your answer.

Answer:

a) Algorithm to find x^n in linear time is

```
public static int linearXpowN(int x, int n){
        if(n == 1) return x;
        int product = 1;
        if(n % 2 != 0){
                product = x;
                n -= 1;
        }
        return product * linearXpowN(x, n / 2) * linearXpowN(x, n / 2);
}
```

b) Let T(n) is the time complexity of the above function. Number of basic operations are:

Code	Cost	Number of times it runs
if(n == 1) return x;	C1	1
int product = 1;	C2	1
if(n % 2 != 0)	C3	1
product = x;	C4	1
n -= 1;	C5	1
return product * linearXpowN(x, n / 2) * linearXpowN(x, n / 2);	T(n / 2)	2

So, T(n) = 2T(n / 2) + (C1 + C2 + C3 + C4 + C5)
Let, C = (C1 + C2 + C3 + C4 + C5)
Then, T(n) = 2T(n / 2) + C
We can apply Master's theorem only if the recurrence relation is of form:
T(n) = a * T(n / b) + f(n)
Where a >= 1, b >= 2 and f(n) is an asymptotically positive function
Given function is of the same form where a = 2, b = 2 and f(n) = C

f(n) = Theta(1)

=== Theta(n^0)

So, d = 0 in Master's Theorem

$\log_b a = \log_2 2 = 1$

Since, $d = 0 < \log_b a = 1$, the given function follows Case-I pattern of Master's Theorem

According to Case-I of Master's Theorem,

If $f(n) = Theta(n^d)$ where $d < log_b a$

Then f(n) grows asymptotically slower than $log_b a$ Therefore,

$T(n) = Theta(n^{log_b a})$

Therefore, $T(n) = Theta(n^{log_b a})$

$=== Theta(n)$

Therefore, $T(n) = Theta(n)$

c) Algorithm to find x^n in logarithmic time is

```
public static int logarithmicXpowN(int x, int n){
        if(n == 1) return x;
        int product = 1;
        if(n % 2 != 0){
                product = x;
                n -= 1;
        }
        int subProduct = logarithmicXpowN(x, n / 2);
        return product * subProduct * subProduct;
}
```

Let T(n) is the time complexity of the above function. Number of basic operations are:

Code	Cost	Number of times it runs
if(n == 1) return x;	C1	1
int product = 1;	C2	1
if(n % 2 != 0)	C3	1
product = x;	C4	1
n -= 1;	C5	1
int subProduct = logarithmicXpowN(x, n / 2);	T(n / 2)	1
return product * subProduct * subProduct	C6	1

So, T(n) = T(n / 2) + (C1 + C2 + C3 + C4 + C5 + C6)

Let, C = (C1 + C2 + C3 + C4 + C5 + C6)

Then, T(n) = T(n / 2) + C

We can apply Master's theorem only if the recurrence relation is of form:

T(n) = a * T(n / b) + f(n)

Where a >= 1, b >= 2 and f(n) is an asymptotically positive function

Given function is of the same form where a = 1, b = 2 and f(n) = C

$f(n) = Theta(1)$

$=== Theta(n^0)$

So, d = 0 in Master's Theorem

$log_b a = log_2 1 = 0$

Since, d = 0 == $log_b a$ = 0, the given function follows Case-III pattern of Master's Theorem

According to Case-III of Master's Theorem,

If $f(n) = Theta(n^d log^k n)$ where $d = log_b a$

Then, $T(n) = Theta(n^d log^{(k+1)} n)$

We can rewrite f(n) as

$f(n) = Theta(1)$

$Theta(n^0 log^0 n)$

So, d = 0 and k = 0 in Master's Theorem

Therefore, $T(n) = Theta(n^d log^{(k+1)} n)$

$=== Theta(n^0 log^{(0+1)} n)$

$=== Theta(logn)$

Therefore, T(n) = Theta(logn)

Question 2.

a) Explain the divide and conquer algorithmic technique in general. How do we usually analyze the time complexity of algorithms written using this technique. Consider an example and explain.

b) Quick sort algorithm uses a divide and conquer strategy to sort the given list. Assume the pivot is always defined such that it divides up the list into two subarrays of equal size. Analyze your algorithm by writing the recurrence relation. Apply Master Theorem to bound the relation asymptotically. Show all your work.

Answer:

a)

A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This technique can be divided into the following three parts:

1. **Divide:** This involves dividing the problem into smaller sub-problems.
2. **Conquer:** Solve sub-problems by calling recursively until solved.
3. **Combine:** Combine the sub-problems to get the final solution of the whole problem.

Total time taken by a Divide and Conquer algorithm will have contribution from each of the three components

- Time to divide the problem into sub-problem; Let's say D1(n)
- Time to solve sub-problems
  - How many sub-problems? Let's say 'a' sub-problems
  - What is the size of sub-problems? Let's say 'n/b'
- Time to combine solutions of sub-problems; Let's say D2(n)

If T(n) is the time taken by the algorithm to solve a problem of size n then

$T(n) = a * T(n / b) + D1(n) + D2(n)$

$=== T(n) = a * T(n / b) + D(n)$

We analyse the time complexity of divide and conquer algorithms by writing a recurrence relation for that algorithm and solving that recurrence relation using iterative method or Master's Theorem to find the time complexity.

Example: Consider Algorithm to find maximum and minimum in an array.

```
Algorithm FindMaxMin(S){
    if | S | == 1 then return (S);
    else{
        Split S into two sets S1, S2 of equal size.
        FindMaxMin(S1); Suppose Max1, Min1 are the maximum, minimum of S1
        FindMaxMin(S2); Suppose Max2, Min2 are the maximum, minimum of S2
```

```
                return (MAX = max(Max1, Max2), MIN = min(Min1, Min2));
            }
        }
```

Analysis:

Suppose, that the number of basic operations needed to find the maximum and the minimum of the set S is T(n). Time taken to divide the problem into sub-problems is O(1). Since we are dividing the given list into two halves of size n/2 each and store them in S1, S2. FindMaxMin(S1) and FindMaxMin(S2) each take T(n/2) time. Once maximum and minimum of S1, S2 are found we perform 2 comparisons to combine solutions of sub-problems which take O(1) time.

So, the recurrence relation will be:

$T(n) = 2 * T(n / 2) + O(1)$

We can apply Master's theorem only if the recurrence relation is of form:

$T(n) = a * T(n / b) + f(n)$

Where a >= 1, b >= 2 and f(n) is an asymptotically positive function

Given function is of the same form where a = 2, b = 2 and f(n) = O(1)

f(n) = Theta(1)

=== Theta(n^0)

So, d = 0 in Master's Theorem

$\log_b a = \log_2 2 = 1$

Since, $d = 0 < \log_b a = 1$, the given function follows Case-I pattern of Master's Theorem

According to Case-I of Master's Theorem,

If $f(n) = \text{Theta}(n^d)$ where $d < \log_b a$

Then f(n) grows asymptotically slower than $\log_b a$ Therefore,

$T(n) = \text{Theta}(n^{\log_b a})$

Therefore, $T(n) = \text{Theta}(n^{\log_b a})$

=== Theta(n)

Therefore, T(n) = Theta(n)

b)

Algorithm of Quick Sort is

```
QuickSort(S){
    If | S | == 2 then compare those 2 numbers and return (minValue, maxValue)
    else{
        Pick an arbitrary element Sk (pivot) from S
        Divide S into parts S1, S2, S3 such that elements of S1 < Sk, elements of S2 = Sk
        and elements of S3 >Sk
        Return (QuickSort(S1), S2, QuickSort(S3))
    }
}
```

Partitioning takes O(n) since we go through entire array to partition them. Let S1 has size k and S2 has size 1 and S3 has size n - k - 1.

Time taken to solve S1 is T(k). Time taken to solve S2 is T(n - k - 1).

So, $T(n) = T(k) + T(n - k - 1) + O(n)$

When pivot divides array into two equal halves for every call that is the best case of Quick Sort.
At best case, k will be n/2.

So, $T(n) = T(n / 2) + T(n / 2) + O(n)$

$T(n) = 2 * T(n / 2) + O(n)$

We can apply Master's theorem only if the recurrence relation is of form:

$T(n) = a * T(n / b) + f(n)$

Where a >= 1, b >= 2 and f(n) is an asymptotically positive function

Given function is of the same form where a = 2, b = 2 and f(n) = O(n)

$f(n) = Theta(n)$

$=== Theta(n^1)$

So, d = 1 in Master's Theorem

$\log_b a = \log_2 2 = 1$

Since, d = 1 == $\log_b a$ = 1, the given function follows Case-III pattern of Master's Theorem

According to Case-III of Master's Theorem,

If $f(n) = Theta(n^d \log^k n)$ where $d = \log_b a$

Then, $T(n) = Theta(n^d \log^{(k+1)} n)$

We can rewrite f(n) as

$f(n) = Theta(n)$

$Theta(n^1 \log^0 n)$

So, d = 1 and k = 0 in Master's Theorem

Therefore, $T(n) = Theta(n^d \log^{(k+1)} n)$

$=== Theta(n^1 \log^{(0+1)} n)$

$=== Theta(nlogn)$

Therefore, T(n) = Theta(nlogn)

Question 3.

(a). Let's consider a long, quiet country road with houses scattered very sparsely along it. (Picture the road as a long line segment with an eastern endpoint and a western endpoint.) Further let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations. Give an efficient algorithm that achieves this goal using as few base stations as possible. Prove its correctness and explain its time complexity.

(b). Implement your algorithm in part a in C++/Java. Given a list of n house positions, output the positions of base stations for optimal outcome. Paste your code in the solution file. Comment as needed. Analyze its time complexity providing enough details and justification. Show all your work.

Answer:

a)

Suppose there is a long straight country road, with n houses sparsely scattered along the road. Positions along the road are specified by distance in kilometers from one end, say in terms of distance d in an array form. The position of the k^{th} house along the road is given by d = distance[k]. Assume a cell tower has a range of 4 km. That is, if the j^{th} tower is placed at kilometer t_j along the road, then any house k with $|distance[k] - t_j| \leq 4$ would have service from that tower. The problem is to place the minimum number of cell phone towers along the road, so that each of the n houses can get service from at least one of the towers.

We can use greedy algorithm to solve this problem efficiently. The approach to follow is

"When there is a house not covered by any base station, we place one 4 miles after the house."

Algorithm:

```
PlaceTowers(distance){
        Sort the distance array in ascending order.
        if n == 0 then return {}

        towerPositions = []
        Add distance[0] + 4 to towerPositions (First tower is as far down the road as possible)
        lastTowerIndex = 0 (Index of Latest Tower which is 0 that we added above)

        for index = 1 to n - 1{
                if abs(distance[index] − towerPositions[lastTowerIndex]) > 4 (House at index has
                                                                                no service)
                {
                        Add distance[index] + 4 to towerPositions (Build next tower as far down the
                                                                        road as possible.)
                        lastTowerIndex += 1
                }
        }

        return towerPositions
}
```

Correctness of the algorithm:

We get minimum number of base stations when each house is serviced by only 1 base station.

The algorithm above provides the optimal solution because we are making sure that no house gets served by more than 1 base station.

Let's prove by contradiction. Let's assume that a house is getting serviced by 2 base stations. But, according to our logic we are creating a new base station only if abs(distance[index] − towerPositions[lastTowerIndex]) which means the latest tower we placed is not in range of the house at index. If the previous base station is in the range we use that base station only to service this house without creating a new base station. Also, we are placing new tower 4kms away from this house so this new base station can't service any house before index. So, no house can be serviced by 2 adjacent base stations at a time. This contradicts the claim we assumed at the beginning. Therefore, each house gets serviced by only one base station which as a result gives us minimum number of base stations. Therefore, this algorithm is the optimal algorithm.

Time Complexity Analysis:

Sorting takes O(nlogn) time. We are looping through the sorted array one time which takes O(n) time and performing O(1) time taking actions inside the loop. So the time complexity will be

$f(n) = O(n) * O(1) + O(nlogn)$

$f(n) = O(n) + O(nlogn)$

If the problem specified explicitly that the house positions are sorted before calling this function then we won't have to do that latter part i.e; O(nlogn). So the time complexity of algorithm if the distances array is already sorted will be O(n).

If the distances array is not already sorted we need to perform O(nlogn) work to sort the array and O(n) work to find the positions of base stations. Since O(nlogn) grows faster than O(n), the time complexity will be O(nlogn).

b)
Java Code to implement this algorithm:

```
public static ArrayList<Integer> placeTowers(int[] distances){
        Arrays.sort(distances);
        ArrayList<Integer> towerPositions = new ArrayList<>();
        if(distances.length == 0) return towerPositions;

        towerPositions.add(distances[0] + 4);
        int lastTowerIndex = 0;

        for(int index = 1;index < distances.length;index++){
                if(Math.abs(distances[index] - towerPositions.get(lastTowerIndex)) > 4){
                        towerPositions.add(distances[index] + 4);
                        lastTowerIndex += 1;
                }
        }

        return towerPositions;
}
```

Time Complexity Analysis:

Code	Cost	Number of times it runs
Arrays.sort(distances);	O(nlogn)	1
ArrayList<Integer> towerPositions = new ArrayList<>();	C1	1
if(distances.length == 0) return towerPositions;	C2	1
towerPositions.add(distances[0] + 4);	C3	1
int index = 1	C4	1
index < distances.length	C5	n + 1
index++	C6	n
if(Math.abs(distances[index] - towerPositions.get(lastTowerIndex)) > 4){	C7	n
towerPositions.add(distances[index] + 4);	C8	n
lastTowerIndex += 1;	C9	n
return towerPositions;	C10	1

The function is
f(n) = C1 + C2 + C3 + C4 + C5 * (n + 1) + C6 * n + C7 * n + C8 * n + C9 * n + C10 + O(nlogn)
Let f'(n) = C1 + C2 + C3 + C4 + C5 * (n + 1) + C6 * n + C7 * n + C8 * n + C9 * n + C10
f'(n) = (C5 + C6 + C7 + C8 + C9) * n + (C1 + C2 + C3 + C4 + C5 + C10)
Let C > 10 * (C1 + C2 + C3 + C4 + C5 + C6 + C7 + C8 + C9 + C10)
Then, f'(n) will always be <= C * n for all n >= 1
According to Upper Bound definition in Asymptotic Notations, f(n) = O(g(n)) if for some constants C & n0 f(n) <= C * g(n) for all n >= n0
Here g(n) = n, n0 = 1 and C = 10
Therefore f'(n) = O(n) for all n >= 1
Now, f(n) = f'(n) + O(nlogn)
f(n) = O(n) + O(nlogn)

If the problem specified explicitly that the house positions are sorted before calling this function then we won't have to do that latter part i.e; O(nlogn). So the time complexity of algorithm if the distances array is already sorted will be O(n).
If the distances array is not already sorted we need to perform O(nlogn) work to sort the array and O(n) work to find the positions of base stations. Since O(nlogn) grows faster than O(n), the time complexity will be O(nlogn).