ANALYSIS AND DESIGN OF ALGORITHM(E2UC403B)

NOTES

1.Explain analysis of algorithms is important?(2)

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms is important?

More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Algorithm Analysis:

- Best case
- Worst case
- Average case

2.Describe the Algorithm Analysis of Binary Search.(2)

Analysis of Binary Search

In the base case, the algorithm will end up either finding the element or just failing and returning false. In both cases, the algorithm is going to take a constant time because only comparison and return statements are going to be executed.

Otherwise, comparison and calculation of the middle element will take constant time and then the problem is divided into another problem of size $\frac{n}{2}$ (either `BINARY-SEARCH(A, start, middle-1, x)` or `BINARY-SEARCH(A, middle+1, end, x)`). So, the overall running time ($T(n)$) can be written as:

$$T(n) = T\left(\frac{n}{2}\right) + c$$

Basically, we are reducing our problem to half the size in each iteration.

n	$n/2^0$
↓	
n/2	$n/2^1$
↓	
n/4	$n/2^2$
↓	
n/8	$n/2^3$
.	
.	
.	
$n/2^i$	

In general, we can write the size of the problem as $\frac{n}{2^i}$ after doing i comparisons. Thus, for the base case, $\frac{n}{2^i} = 1 => i = \log_2 n$.

Also, the comparison will reach the base case only in the worst case and thus, binary search is a $O(\lg n)$ algorithm.

3. Explain Divide – and – Conquer approach?(2)

A **divide and conquer algorithm** is a strategy of solving a large problem by

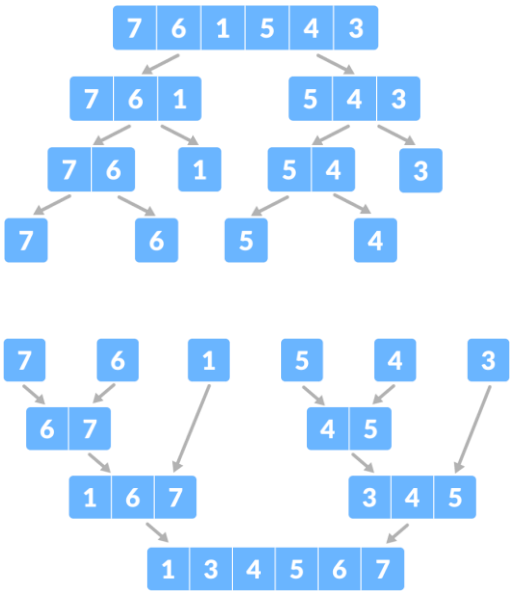
1. breaking the problem into smaller sub-problems
2. solving the sub-problems, and
3. combining them to get the desired output.

To use the divide and conquer algorithm, **recursion** is used

How Divide and Conquer Algorithms Work?

Here are the steps involved:

1. **Divide**: Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.



For a merge sort, the equation can be written as:

```
T(n) = aT(n/b) + f(n)
     = 2T(n/2) + O(n)
Where,
a = 2 (each time, a problem is divided into 2 subproblems)
n/b = n/2 (size of each sub problem is half of the input)
f(n) = time taken to divide the problem and merging the subproblems
```

```
T(n/2) = O(n log n) (To understand this, please refer to the master theorem.)

Now, T(n) = 2T(n log n) + O(n)

      ≈ O(n log n)
```

4. Explain space complexity (2)

Space complexity *is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

Space Complexity = **Auxiliary Space + Input space**

5. Explain Asymptotic notations in algorithm analysis(5)

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

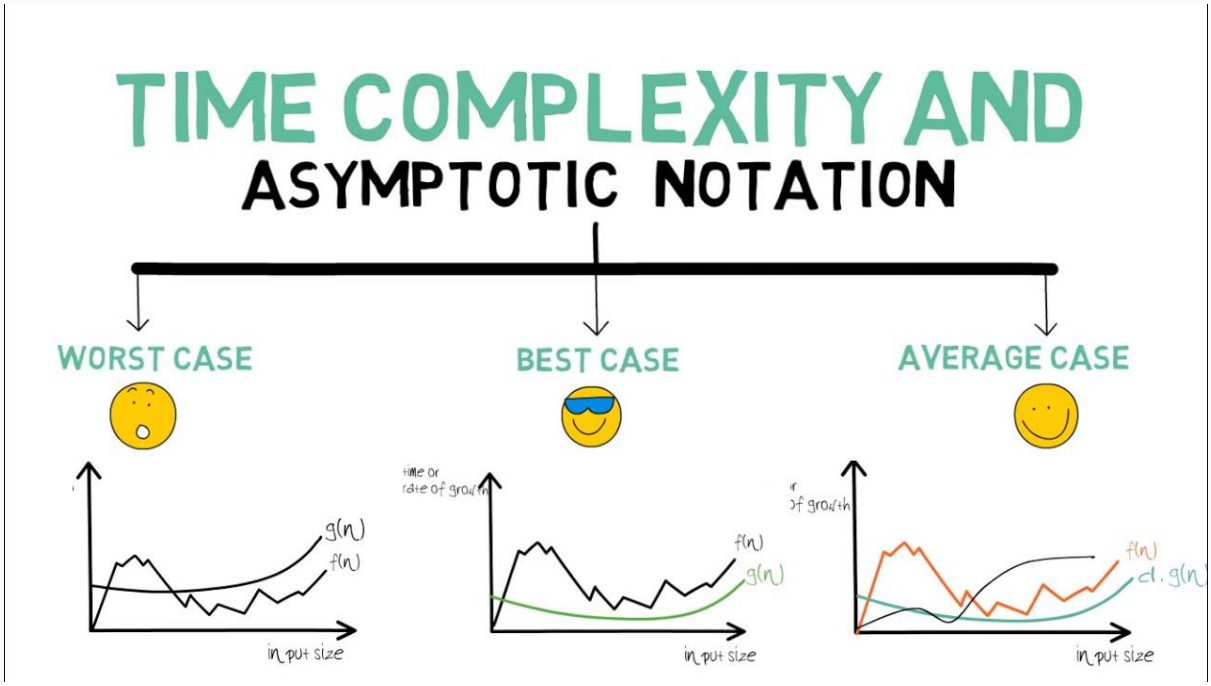
Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



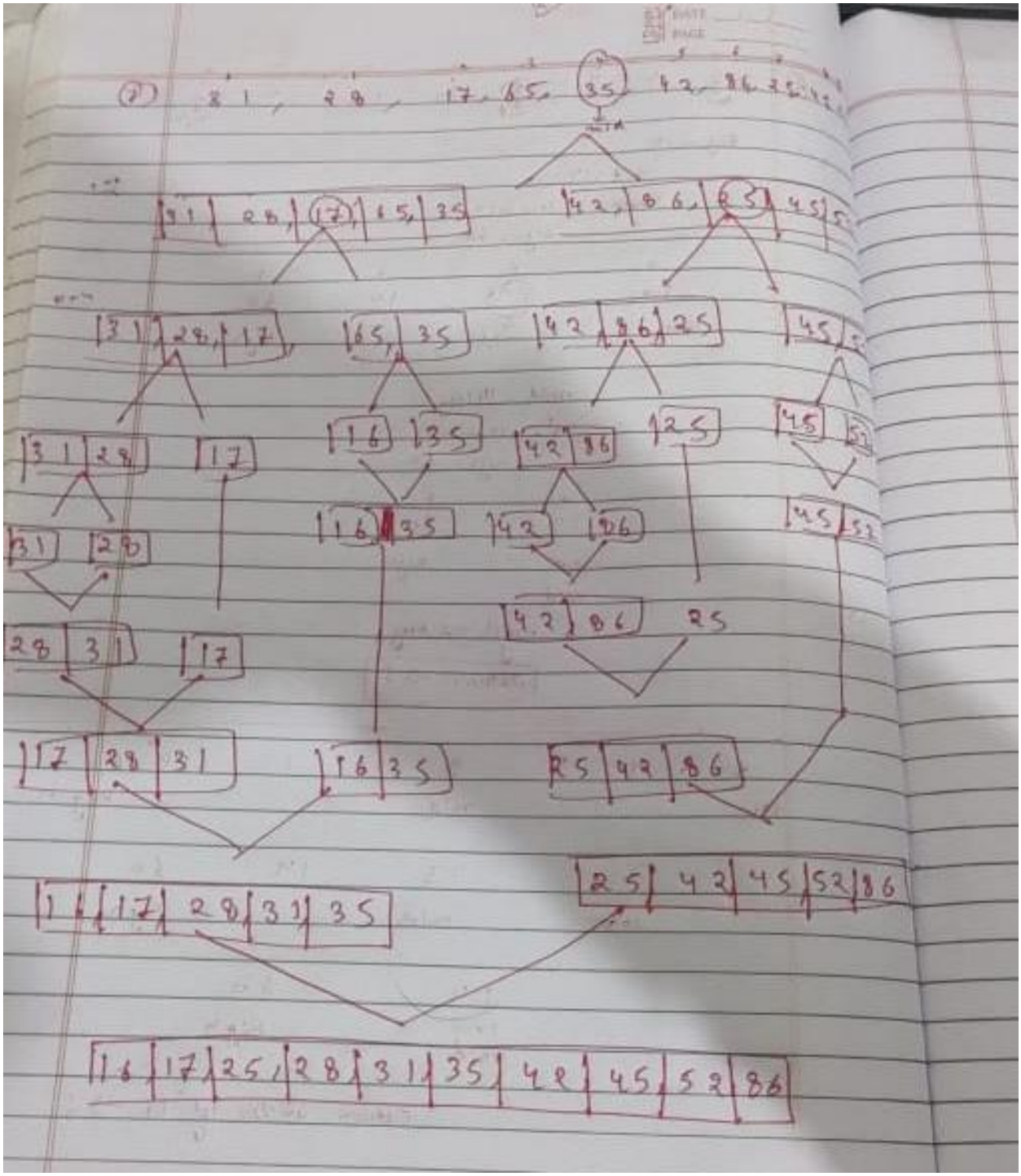
6. Explain time complexity(2)

Time complexity is a programming term that quantifies the amount of time it takes a sequence of code or an algorithm to process or execute in proportion to the size and cost of input.

It will not look at an algorithm's overall execution time. Rather, it will provide data on the variation (increase or reduction) in execution time when the number of operations in an algorithm increases or decreases.

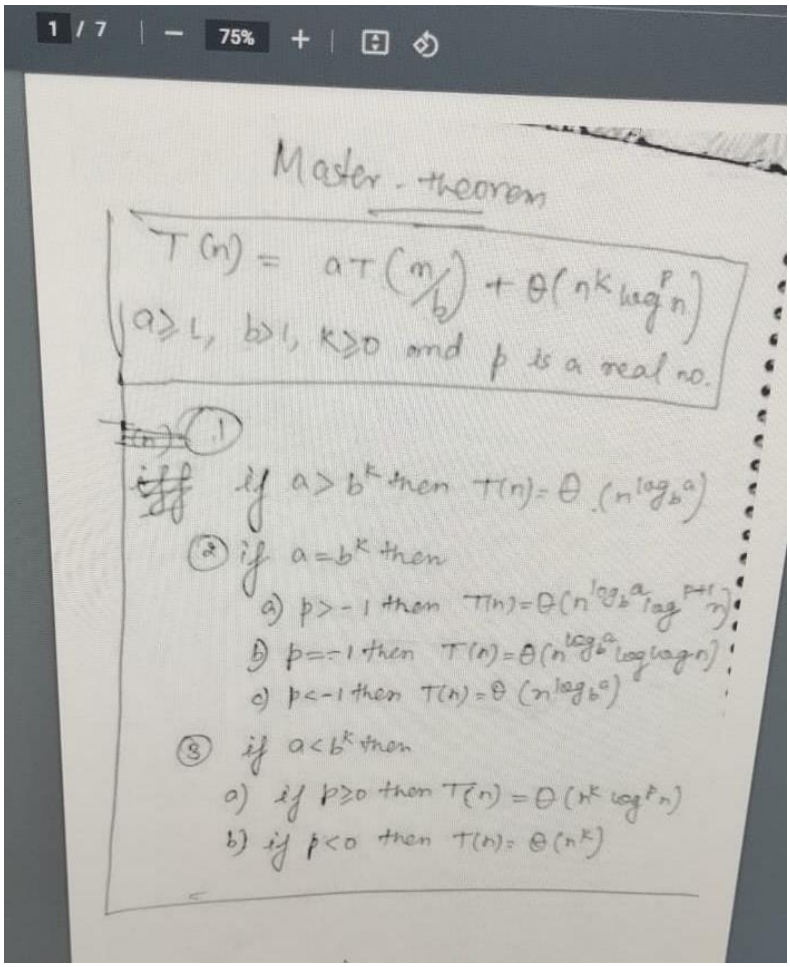
Yes, as the definition suggests, the amount of time it takes is solely determined by the number of iterations of one-line statements inside the code.

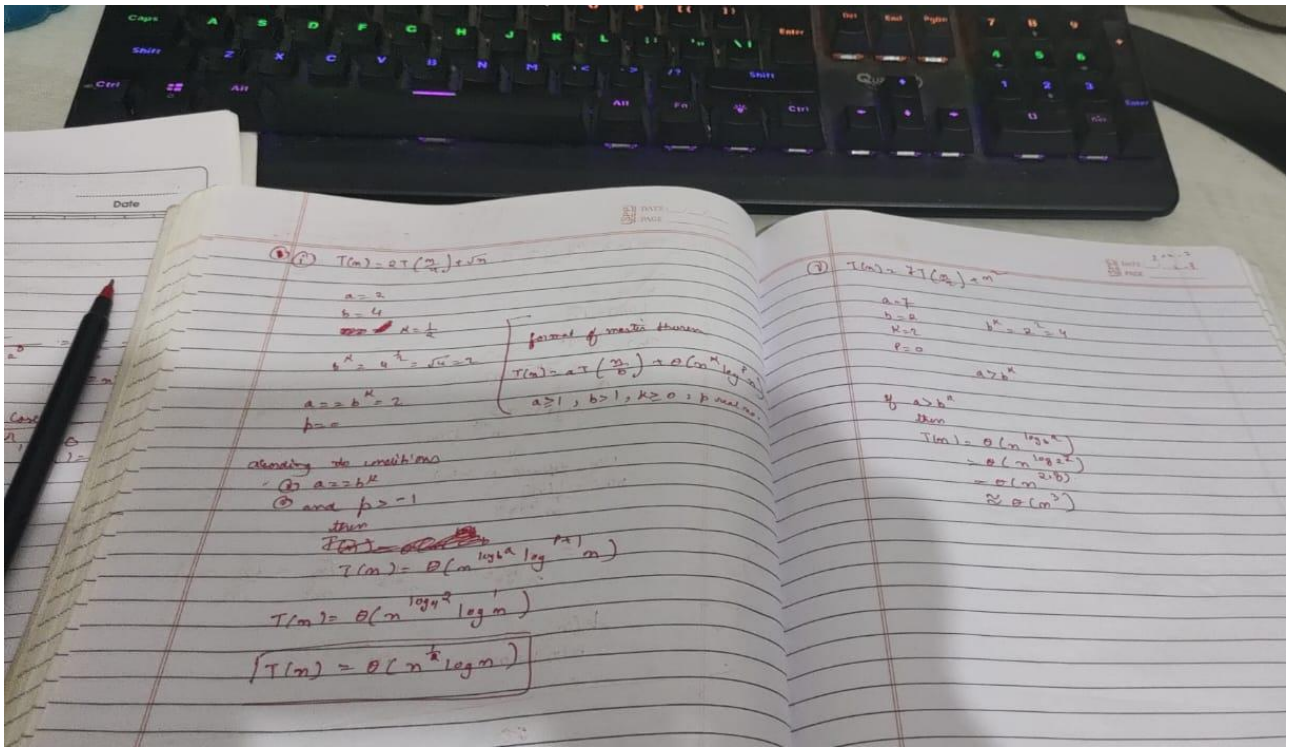
7.Apply Merge Sort to sort the list a[1:10]=(31,28,17,65,35,42.,86,25,45,52). Draw the tree of recursive calls of merge sort, merge functions (5 MARKS)



8. Solve using Masters theorem i) T(n)=2T(n/4)+√n ii) T(n)=7T(n/2)+ n2 (5)

formula





9. Write and explain recursive binary search algorithm. (5)

2. Recursive Method (The recursive method follows the divide and conquer approach)

```
binarySearch(arr, x, low, high)
        if low > high
            return False

        else
            mid = (low + high) / 2
            if x == arr[mid]
                return mid

            else if x > arr[mid]         // x is on the right side
                return binarySearch(arr, x, mid + 1, high)

            else                         // x is on the left side
                return binarySearch(arr, x, low, mid - 1)
```

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

time complexity analysis in ans-2

10. Write Divide – And – Conquer recursive Merge sort algorithm and derive the time complexity of this algorithm.(8)

ans-3

Merge Sort Algorithm-

Merge Sort Algorithm works in the following steps-

- It divides the given unsorted array into two halves- left and right sub arrays.
- The sub arrays are divided recursively.
- This division continues until the size of each sub array becomes 1.
- After each sub array contains only a single element, each sub array is sorted trivially.
- Then, the above discussed merge procedure is called.

- The merge procedure combines these trivially sorted arrays to produce a final sorted array.

Time Complexity Analysis-

In merge sort, we divide the array into two (nearly) equal halves and solve them recursively using merge sort only.

So, we have-

$$T\left(\frac{n_L}{2}\right) + T\left(\frac{n_R}{2}\right) = 2T\left(\frac{n}{2}\right)$$

n_L = Left Half

n_R = Right Half

$$n_L \approx n_R$$

Finally, we merge these two sub arrays using merge procedure which takes Θ(n) time as explained above.

If T(n) is the time required by merge sort for sorting an array of size n, then the recurrence relation for time complexity of merge sort is-

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Recurrence Relation

On solving this recurrence relation, we get T(n) = Θ(nlogn).

Thus, time complexity of merge sort algorithm is T(n) = Θ(nlogn).

```
// L : Left Sub Array , R : Right Sub Array , A : Array

merge(L, R, A)
{
    nL = length(L)    // Size of Left Sub Array
    nR = length(R)    // Size of Right Sub Array

    i = j = k = 0

    while(i<nL && j<nR)
    {
      /* When both i and j are valid i.e. when both the sub arrays have elements to insert in A */

        if(L[i] <= R[j])
        {
           A[k] = L[i]
           k = k+1
           i = i+1
        }
        else
        {
           A[k] = R[j]
           k = k+1
           j = j+1
        }
    }

    // Adding Remaining elements from left sub array to array A
    while(i<nL)
    {
       A[k] = L[i]
       i = i+1
       k = k+1
    }

    // Adding Remaining elements from right sub array to array A
    while(j<nR)
    {
       A[k] = R[j]
       j = j+1
       k = k+1
    }
}
```

```
// A : Array that needs to be sorted

MergeSort(A)
{
   n = length(A)
   if n<2 return
   mid = n/2
   left = new_array_of_size(mid)        // Creating temporary array for left
   right = new_array_of_size(n-mid)     // and right sub arrays

   for(int i=0 ; i<=mid-1 ; ++i)
   {
      left[i] = A[i]                    // Copying elements from A to left
   }

   for(int i=mid ; i<=n-1 ; ++i)
   {
      right[i-mid] = A[i]               // Copying elements from A to right
   }

   MergeSort(left)                      // Recursively solving for left sub array
   MergeSort(right)                     // Recursively solving for right sub array

   merge(left, right, A)                // Merging two sorted left/right sub array to final
array
}
```

11.Write the algorithm for Strassen's matrix multiplication and find the time complexity of the algorithm.(8)

Let us consider two matrices ***X*** and ***Y***. We want to calculate the resultant matrix ***Z*** by multiplying ***X*** and ***Y***.

Naïve Method

First, we will discuss naïve method and its complexity. Here, we are calculating ***Z = X × Y***. Using Naïve method, two matrices (***X*** and ***Y***) can be multiplied if the order of these matrices are ***p × q*** and ***q × r***. Following is the algorithm.

```
Algorithm: Matrix-Multiplication (X, Y, Z)
for i = 1 to p do
   for j = 1 to r do
      Z[i,j] := 0
      for k = 1 to q do
         Z[i,j] := Z[i,j] + X[i,k] × Y[k,j]
```

Complexity

Here, we assume that integer operations take ***O(1)*** time. There are three **for** loops in this algorithm and one is nested in other. Hence, the algorithm takes $O(n^3)$ time to execute.

Strassen's Matrix Multiplication Algorithm

In this context, using Strassen's Matrix multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix multiplication can be performed only on **square matrices** where **n** is a **power of 2**. Order of both of the matrices are **n × n**.

Divide **X**, **Y** and **Z** into four (n/2)×(n/2) matrices as represented below −

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \qquad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using Strassen's Algorithm compute the following −

$$M_1 := (A + C) \times (E + F)$$

$$M_2 := (B + D) \times (G + H)$$

$$M_3 := (A - D) \times (E + H)$$

$$M_4 := A \times (F - H)$$

$$M_5 := (C + D) \times (E)$$

$$M_6 := (A + B) \times (H)$$

$$M_7 := D \times (G - E)$$

Then,

$$I := M_2 + M_3 - M_6 - M_7$$

$$J := M_4 + M_6$$

$$K := M_5 + M_7$$

$$L := M_1 - M_3 - M_4 - M_5$$

Analysis

$$T(n) = \begin{cases} c & if\ n = 1 \\ 7\ x\ T(\frac{n}{2}) + d\ x\ n^2 & otherwise \end{cases}$$ where *c* and *d* are constants

Using this recurrence relation, we get $T(n) = O(n^{log7})$

Hence, the complexity of Strassen's matrix multiplication algorithm is $O(n^{log7})$.

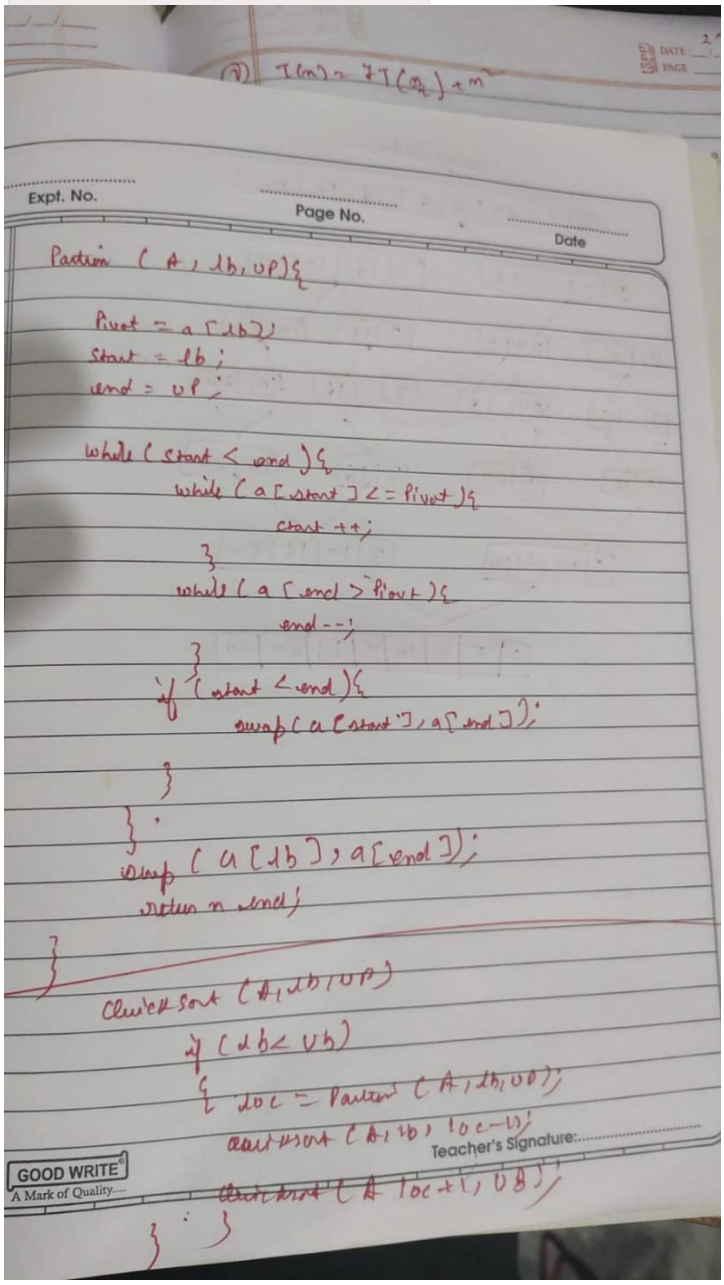
12. Write the quick sort algorithm. Trace the same on data set 4,3,1,9,8,2,4,7. (5)

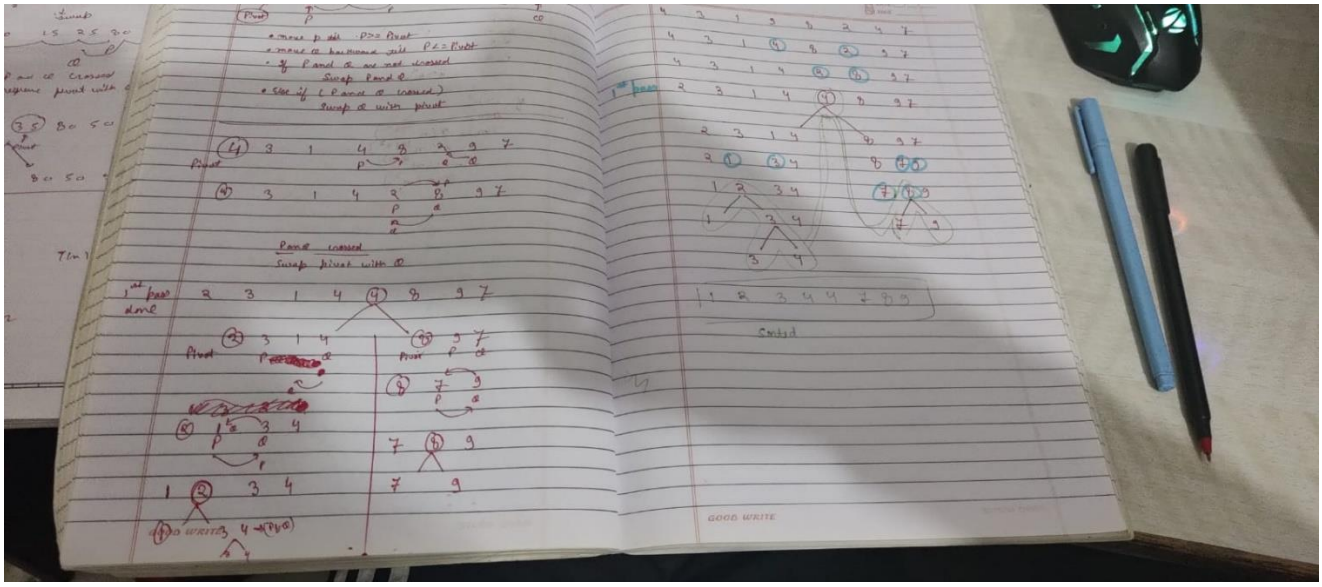
Algorithm:

```
QUICKSORT (array A, start, end)
{
 1 if (start < end)
 2 {
3 p = partition(A, start, end)
4 QUICKSORT (A, start, p - 1)
5 QUICKSORT (A, p + 1, end)
6 }
}
```

1. Time Complexity

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	$O(n^2)$





13. Define an algorithm?(2)

In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output. For example,

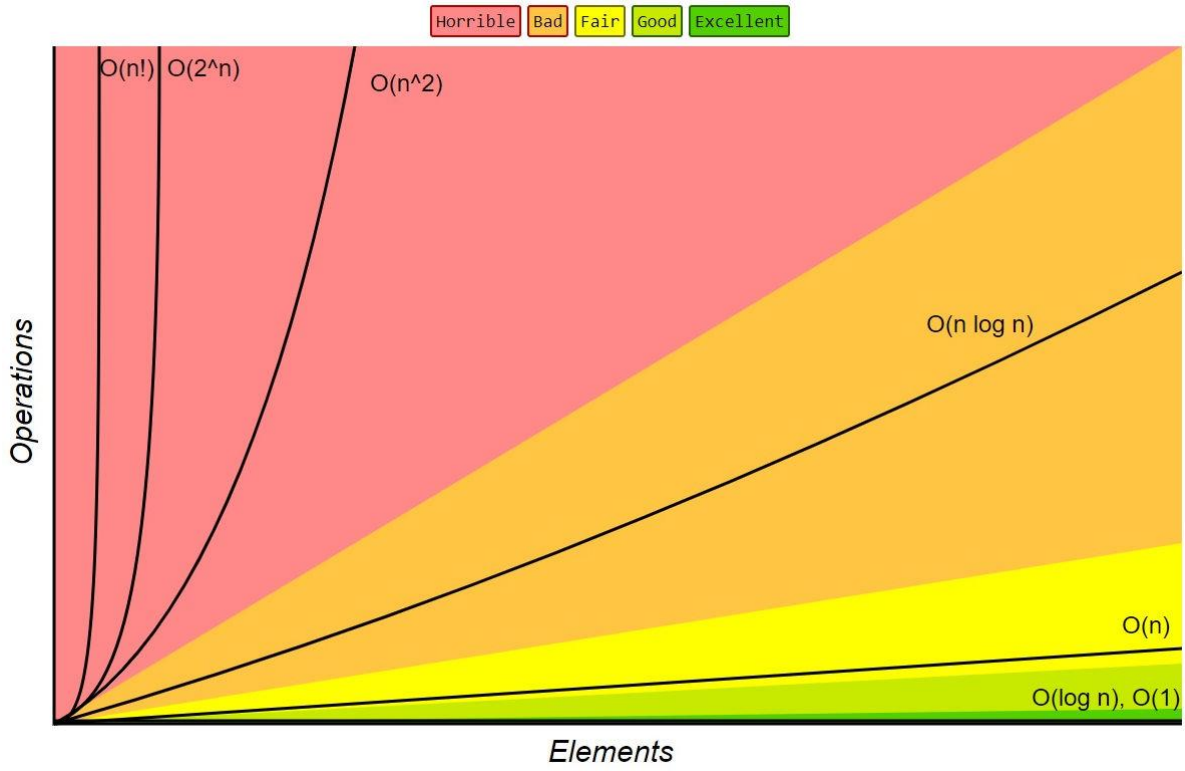
An algorithm to add two numbers:

1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

Qualities of a Good Algorithm

- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

14. Define the Complexity of an Algorithm?(2)



Algorithmic complexity is a measure of how long an algorithm would take to complete given an input of size n. If an algorithm has to scale, it should compute the result within a finite and practical time bound even for large values of n. For this reason, complexity is calculated asymptotically as n approaches infinity.

15. What are the Asymptotic Notations?(2) ans-5

16. Explain the recursive algorithms? State the important rules which every recursive algorithm must follow.(5)

A **recursive algorithm** is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

all recursive algorithms must obey three important laws:

1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

17. Apply merge sort algorithm on input 31, 22, 45, 67, 99, 21, 17, 13(5)

18. How can we compare between two algorithms written for the same problem? (5)

Approach 1: Implement and Test

Approach 2: Graph and Extrapolate ,Approach 2: Create a formula

Approach 3: Approximate, Practice with Big-O

we want to define some parameters upon which we can compare two algorithms. The two most important factors are:

1. **Time**: Ever since the invention of computers, programs are written to reduce the time taken to achieve a job. Hence, an algorithm is considered better if it runs faster than the other one.
2. **Space**: Memory is cheap now days, but everyone loves an application that takes the least amount of RAM. Don't we all hate when you have 70 tabs open, and a web browser slows down your entire PC. Also, when writing system level code, every extra byte of memory matters. Thus, an algorithm which takes up less RAM would be considered to be a better one.

Experimental:

- A fool proof way to compare 2 different algorithms would be to **actually run them and observe the results**. The one which gives you the output in less time would said to be the better one.
- But, when running these algorithms, you need to **ensure that we are using the same hardware for both**. A faster processor can give better results for a poorly written algorithm. Also, you cannot rely on a single run. Your computer might be running other workloads in the background and that can affect the performance. Hence, it is recommended that you several iterations of both the techniques and then get an average value before comparing.
- We also need to **make sure that the input workload for the algorithms remain the same**. Using random input data cannot give us reliable results. It could be possible that certain input loads work unreasonably fast. Example: Trying to sort an already sorted list.

Even with all these conditions in mind, you cannot certainly reach an exact result, as it is scientifically impossible to run 2 algorithms in the exact same conditions. You wouldn't be able to compare two algorithms which work nearly the same. Thus, you need to do a little deep dive in the actual flow of the algorithm.

Analytical:

Your algorithm may contain a lot of conditional blocks, and control blocks that determine the runtime of your algorithm. Suppose you are given with 2 code blocks. Forget about the implementation details, and assume that both of them give the correct answer.

Algorithm 1:

```
for(i = 0; i < 10; i++) {
  for(j = 0; j< 5; j++) {
    // do something...
  }
}
```

Code language: Java (java)

Algorithm 2:

```
for(i = 0; i < 10; i++) {
  // do something...
}

for(j = 0; j < 5; j++) {
  // do something...
```

At the first glance it may seem that both the codes have 2 loops. One runs for 10 times and the other runs for 5 times. So, it is natural to feel that both will take the same time to run. But when we compare both of them analytically, we see that algorithm 1 is doing something for 50 times, while algorithm 2 is doing something for just 15 times. This means that algorithm 1 would be a better choice.

This phenomenon is also known as **Runtime Complexity**.

19. Devise an algorithm to insert a node in a Binary Search Tree.(5) not in cat1
20. write an algorithm to implement Binary Search.Algorithm(8)

Binary Search Algorithm

```
do until the pointers low and high meet each other.
   mid = (low + high)/2
   if (x == arr[mid])
      return mid
   else if (x > arr[mid]) // x is on the right side
      low = mid + 1
   else                   // x is on the left side
      high = mid - 1
```

```
int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;

    if (array[mid] == x)
      return mid;

    if (array[mid] < x)
      low = mid + 1;

    else
      high = mid - 1;
  }

  return -1;
}
```

Binary Search Complexity Time Complexities

- **Best case complexity**: O(1)
- **Average case complexity**: O(log n)
- **Worst case complexity**: O(log n)

The space complexity of the binary search is O(1).

21. Define the Time complexity and space complexity of Algorithms. (5)

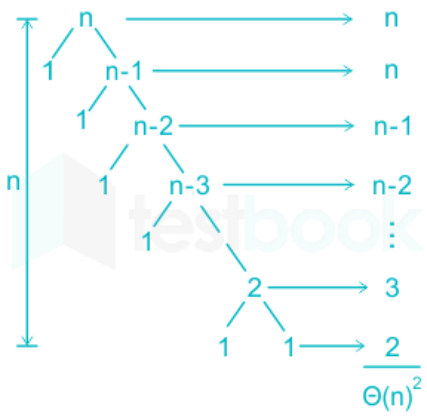
Time Complexity	Space Complexity
Calculates time needed	Calculates memory space needed
Counts time for all statements.	Counts memory space for all variables even inputs and outputs
Depends mostly on input data size	Depends mostly on auxiliary variables size
More important for solution optimization	Less important with modern hardwares
Time complexity of merge sort is $\mathcal{O}(nlogn)$	Space complexity of merge sort is $\mathcal{O}(n)$

22. Explain Asymptotic Notations used in Algorithms. (8) ans=5

23. What is the recurrence for the worst case of Quick Sort and what is the time complexity in the Worst case? (2)

In Quicksort, the worst-case takes Θ (n^2) time. The worst case of quicksort is when the first or the last element is chosen as the pivot element.

Diagram



$$\sum_{p=1}^{N-1} p = 1+2+3+....+N-1 = \frac{N(N-1)}{2} - 1$$

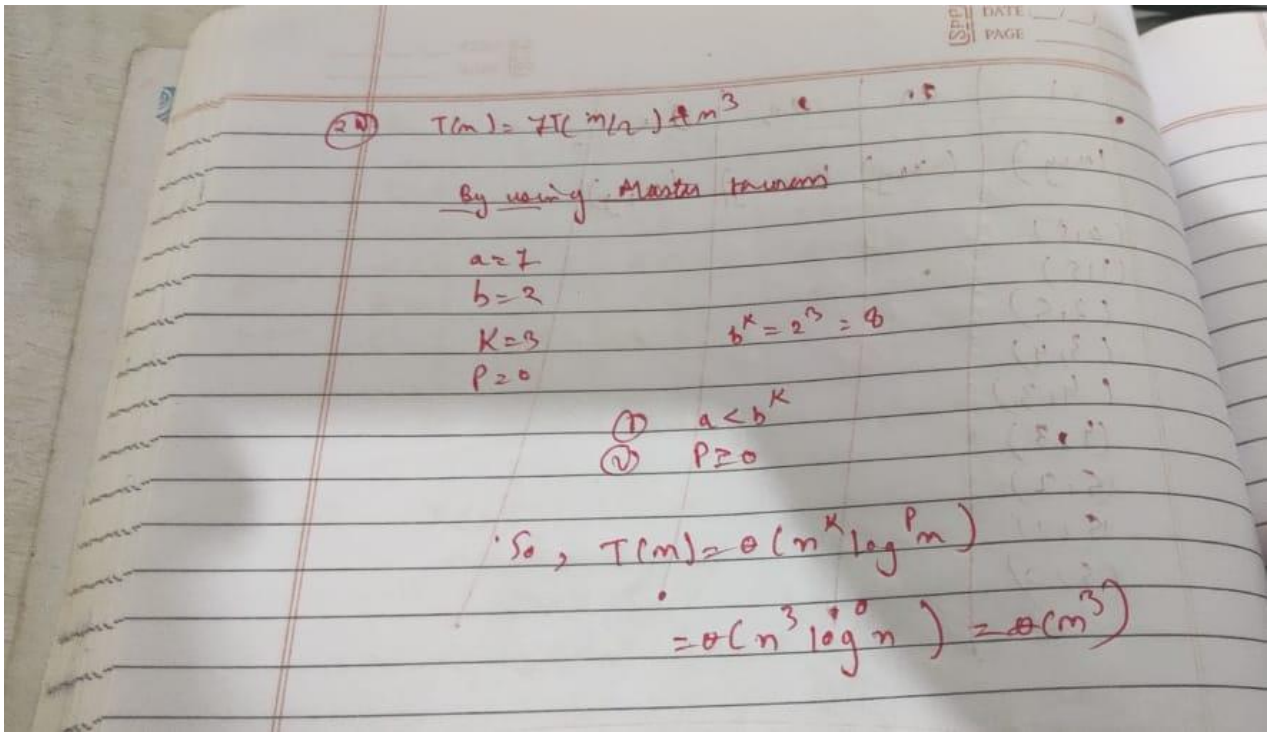
This will give Θ (n^2) time complexity.

Recurrence relation for quick sort algorithm will be,

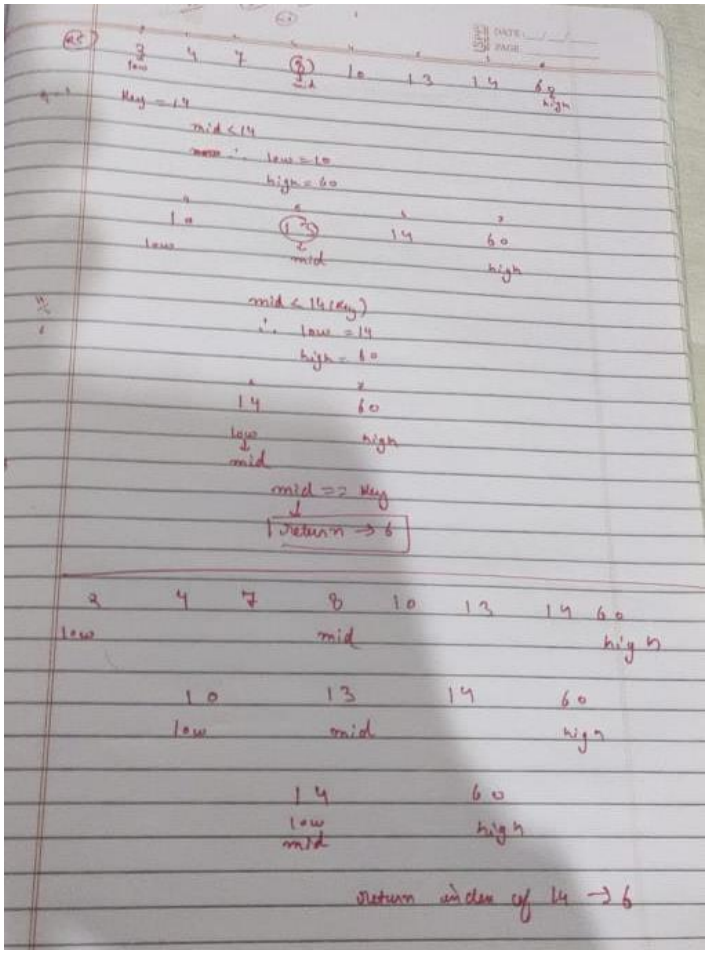
T (n) = T (n-1) + Θ (n)

This will give the worst-case time complexity as Θ (n^2).

24. Solve the recurrence $T(n) = 7T(n/2) + n^3$ (5)



25. Demonstrate Binary Search method to search Key = 14, form the array A=<2,4,7,8,10,13,14,60>.(2)



26. Explain Strassen's algorithm for matrix multiplication.(5) ans-11

27. How time complexity of an algorithm differs from space complexity? (5) ans-21

28. What are the advantages of Merge sort over the quick sort algorithm? (2)

Parameter	Merge Sort	Quicksort
Principle	Works on the divide-and-conquer technique	Works on the divide-and-conquer technique
Partition of Elements	Divides a list into two (almost) equal halves	Can partition a list in any ratio
Stability	Stable	Not Stable
Best-Case Time Complexity	O(n log n)	O(n log n)
Worst-Case Time Complexity	O(n log n)	$O(n^2)$
Average-Case Time Complexity	O(n log n)	O(n log n)
Space Complexity	O(n)	O(1)
Sorting Method	External	Internal
Preferred for	Linked lists	Datasets where the elements are more or less evenly distributed over the range
Efficiency	Equally efficient for all types of inputs of the same size	Might vary depending on the selection of the pivot

29. Explain the Big-Oh() computation (2) ans-5

30. What are the different ways of expressing the complexity of an algorithm?(2) ans-5

31. List the factors which affects the running time of the algorithm(2)

Recursion – Recursion can cause a lot of overhead which increases the running time of an algorithm

size of input

1. No of nested loops
2. Recursive calls

3. Whether used resource heavy operations like %(modulus)
4. Using any predefined function which in turn has non constant running time complexity.
5. No of operations written inside each loop.

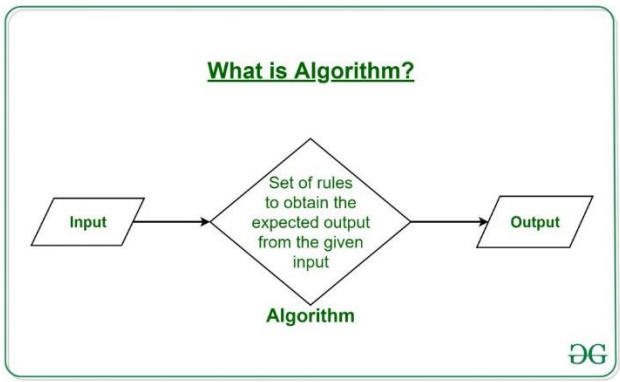
32 . Write the algorithm to perform Binary Search and compute its time complexity. Or Explain binarysearch algorithm with an example(8) ans-20 and 25

33. What are the applications of divide and conquer techniques?(2)

The applications of the Divide and Conquer algorithm are as follows:

- Binary Search
- Merge Sort
- Quick Sort
- Strassen's Matrix Multiplication

34. What is an algorithm?(2)



35. What are the types of algorithm efficiencies?(2)

Time efficiency - a measure of amount of time for an algorithm to execute.

Space efficiency - a measure of the amount of memory needed for an algorithm to execute.

Complexity theory - a study of algorithm performance

36. What is the use of asymptotic notation?(2)

Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n. There are three different notations: big O, big Theta (Θ), and big Omega (Ω). big-Θ is used when the running time is the same for all cases, big-O for the worst case running time, and big-Ω for the best case running time.

37.What is the substitution method?(2)

The substitution method is a condensed way of proving an asymptotic bound on a recurrence by induction. In the substitution method, instead of trying to find an exact closed-form solution, we only try to find a closed-form *bound* on the recurrence. This is often much easier than finding a full closed-form solution, as there is much greater leeway in dealing with constants.

The substitution method is a powerful approach that is able to prove upper bounds for almost all recurrences. However, its power is not always needed; for certain types of recurrences, the master method (see below) can be used to derive a tight bound with less work. In those cases, it is better to simply use the master method, and to save the substitution method for recurrences that actually need its full power.

38.Explain the various Asymptotic Notations used in algorithm design? Or Discuss the properties of asymptotic notations. Or Explain the basic efficiency classes with notations(8) ans-5

39. What are the fundamental steps to solve an algorithm? Explain. Or Describe in detail about the steps in analyzing and coding an algorithm.(8)

Step 1: Obtain a description of the problem.

Step 2: Analyze the problem.

When determining the starting point, we should start by seeking answers to the following questions:

- What data are available?
- Where is that data?
- What formulas pertain to the problem?
- What rules exist for working with the data?
- What relationships exist among the data values?

When determining the ending point, we need to describe the characteristics of a solution. In other words, how will we know when we're done? Asking the following questions often helps to determine the ending point.

- What new facts will we have?
- What items will have changed?
- What changes will have been made to those items?
- What things will no longer exist?

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail.

Step 5: Review the algorithm.

40. Explain recursive relation and typeswith example(8)

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

$$T(n) = \begin{cases} \theta(1) & \text{if } n=1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n>1 \end{cases}$$

Type 1: Divide and conquer recurrence relations –
Following are some of the examples of recurrence relations based on divide and conquer.

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

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

These types of recurrence relations can be easily solved using substitution method.
For example,

$T(n) = T(n-1) + n$

$= T(n-2) + (n-1) + n$

$= T(n-k) + (n-(k-1)) \ldots\ldots (n-1) + n$

Substituting k = n, we get

$T(n) = T(0) + 1 + 2+\ldots\ldots +n = n(n+1)/2 = O(n^2)$

Type 2: Linear recurrence relations –
Following are some of the examples of recurrence relations based on linear recurrence relation.

$T(n) = T(n-1) + n$ for $n>0$ and $T(0) = 1$

Type 3: Value substitution before solving –
Sometimes, recurrence relations can't be directly solved using techniques like substitution, recurrence tree or master method. Therefore, we need to convert the recurrence relation into appropriate form before solving. For example,

$T(n) = T(\sqrt{n}) + 1$

41.Explain the Master theorem with example (8) ans-8

42. What are the applications of divide and conquer techniques? (2) ans-33

43. Write the algorithm to perform Binary Search and compute its time complexity. Or Explain binarysearch algorithm with an example(8) ans-20 and25

44. Explain about Strassen's Matrix Multiplication with example.(8) ans-11

45. Write the algorithm to perform Quick sort algorithm and compute its time complexity. (8) ans-12

Analysis of QuickSort

Time taken by QuickSort, in general, can be written as follows.

$T(n) = T(k) + T(n-k-1) + (n)$

Worst Case:

$T(n) = T(0) + T(n-1) + (n)$ which is equivalent to $T(n) = T(n-1) + (n)$

Best Case:

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

Average Case:

$T(n) = T(n/9) + T(9n/10) + (n)$

The solution of above recurrence is also O(nLogn):

46 Write the algorithm to perform Merge sort algorithm and compute its time complexity (8)

ans-10

Time Complexity: O(N log(N)), Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$T(n) = 2T(n/2) + \theta(n)$

47. Write the algorithm to perform linear search algorithm and compute its time complexity (8)

Algorithm

Steps involved in this algorithm are:

- **Step 1**: Select the **first element** as the **current element**.
- **Step 2**: Compare the **current element** with the **target element**. If matches, then go to *step 5*.
- **Step 3**: If there is a **next element**, then set **current element** to **next element** and go to *Step 2*.
- **Step 4**: **Target element** not found. Go to *Step 6*.
- **Step 5**: **Target element** found and return **location**.
- **Step 6**: Exit process.

Complexity

- Worst case time complexity: **O(N)**
- Average case time complexity: **O(N)**
- Best case time complexity: **O(1)**
- Space complexity: **O(1)**

C

C Copy

```c
#include <stdio.h>
/*
 * Part of Cosmos by OpenGenus Foundation
 * Input: an integer array with size in index 0, the element to be searched
 * Output: if found, returns the index of the element else -1
*/
int search(int arr[], int size, int x)
{
    int i=0;
    for (i=0; i<size; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
int main()
{
    int arr[] = {2,3,1,5}; // Index 0 stores the size of the array (initially 0)
    int size = sizeof(arr)/sizeof(arr[0]);
    int find = 1;
    printf("Position of %d is %d\n", find, search(arr,size,find));
    return 0;
}
```