1. Introduction:

1.1 Why we study algorithm??

Let us discuss various case studies to develop a deep understanding of the term Algorithm:

(i) **Dictionary**: - Find a word in a dictionary. We may apply the following approaches: -

Approach 1: - We open the dictionary which is given *in figure 1.1* from the very first page and start turning around pages one by one till we find the desired word. This approach is the linear searching approach. In general, for searching a word from a dictionary through this approach may take more time. As in the worst case, if he finds the word on the last page. So, he has to turn around n-1 pages to search the word, which is very high is approximately equal to a total number of pages.

Can there be any other approach?

Approach 2: - For searching the word. we may open the dictionary from the middle point, as the dictionary has words arranged in order. We can compare the desired word from the word on the opened page. If the first letter occurrence of the desired word is lesser in alphabetical order, then search in the first half else, search in the second half of the dictionary. So, the problem size is divided in half. Doing this recursively, he can easily search the word. In this approach, the number of comparisons will be less than approach 1.

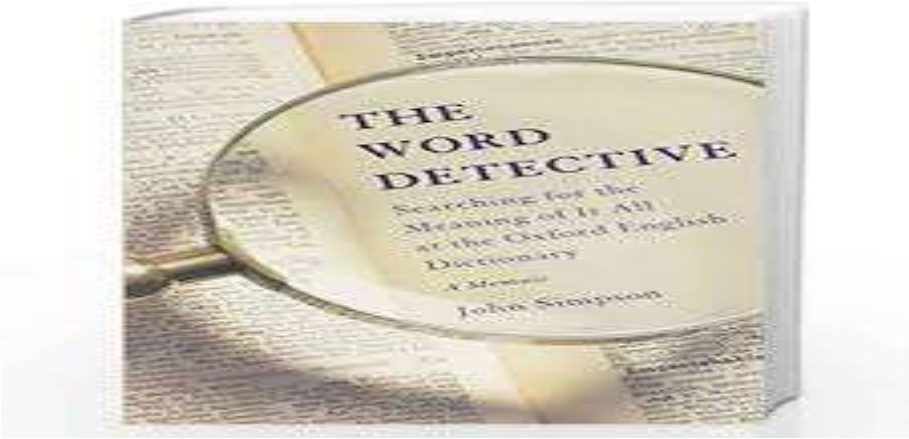


Figure 1.1: Representation of finding a word in a dictionary

(ii) **Google Maps**: - We use Google maps which is shown *in figure 1.2* to find the shortest path from place A to place B. Google maps also use the shortest route-finding algorithm. The Google map is based on a single-source shortest path. Dijkstra's, Algorithm is used to find the shortest path for a single source. ***(Detailed study will be done later)***

Figure 1.2: Google map for finding best route (via shortest path)

1.2. Definition of Algorithm:

An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

OR

An algorithm is thus a sequence of computational steps that transform the input into the output.

Let's take an example which is given in *figure 1.3* below of preparing pizza and try to build an algorithm to represent how it can be made**.**

Input: It takes inputs (ingredients).

Output: Pizza, an output (the completed dish).

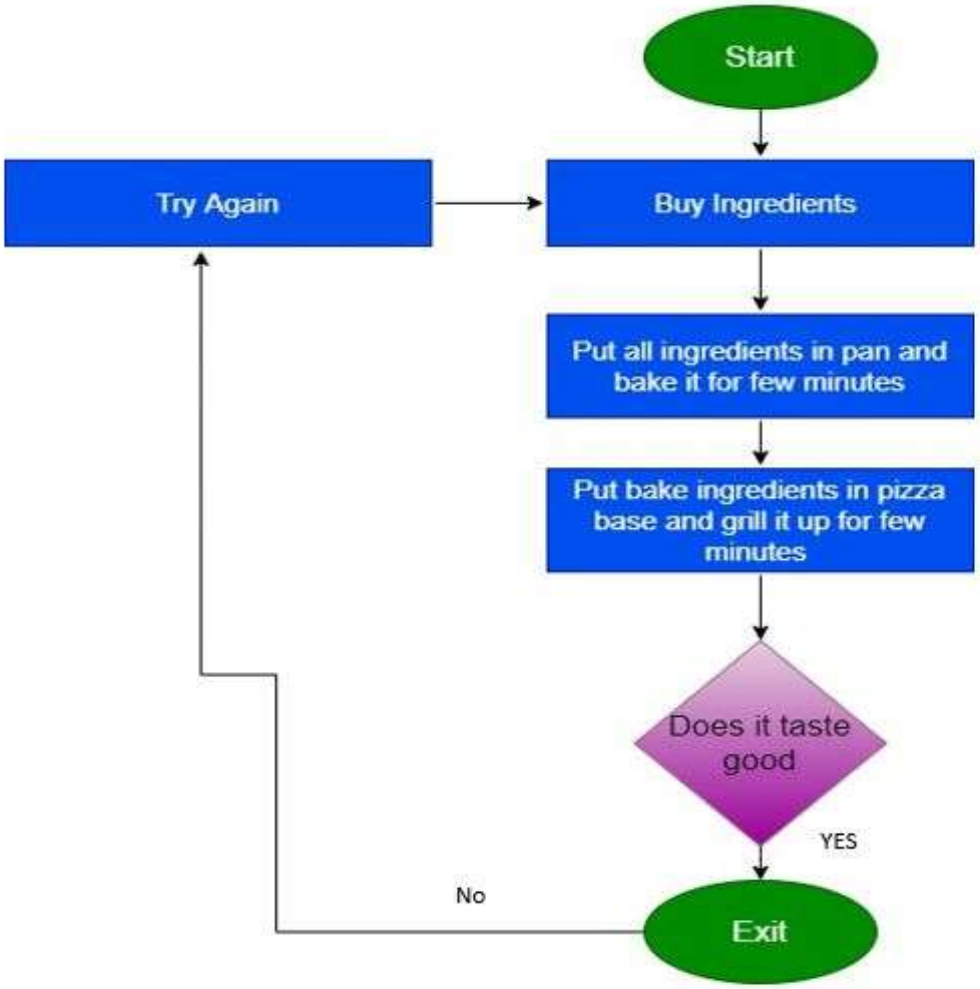


Figure 1.3 : A simple flowchart for Preparation of Pizza

NOTE

- **An algorithm is a step-by-step procedure to solve a problem. For example, in the above diagram, the preparation of pizza takes a finite set for completion. In the same manner, any real-world problem will take any required finite number of steps.**
- **In other words, a set of rules/instructions that specify how a work is to be executed step-by-step in order to achieve the desired results is referred to as an algorithm.**
- **For executing these steps, some cost is incurred, which we calculate in terms of Time and Space.**
- **A simple definition in respect of input and output relation is given below. Here inputs constraints are converted to final output using some steps known as an algorithm.**

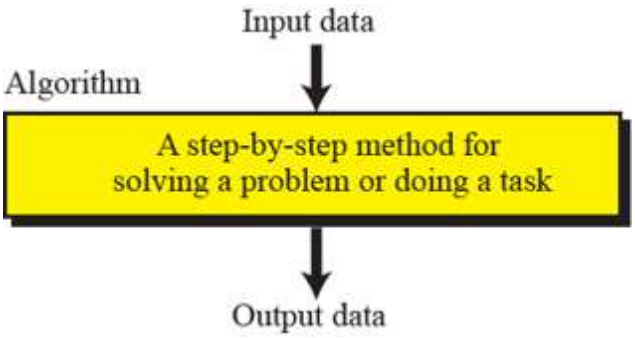


Figure 1.4 : Definition of Algorithm

Table 1: Different Symbols used for Flowchart:

Name	Symbol	Usage
Start or Stop	Start/Stop	The beginning and end points in the sequence.
Process	Process	An instruction or a command.
Decision	Decision	A decision, either yes or no. For example, a decision based on temperature that turns a central heating system on or off.
Input or output	Input/Output	An input is data received by a computer. An output is a signal or data sent from a computer.
Connector		A jump from one point in the sequence to another.
Direction of flow		Connects the symbols. The arrow indicates direction.

Example 2:

Let's understand an example of a person brushing his teeth in context of an algorithm.

Step 1: Take the brush.
Step 2: Apply paste on it
Step 3: Start Brushing
Step 4: Clean
Step 5: Wash
Step 6: Stop

1.3. How to write algorithm:

After understanding the meaning of algorithm, now we will discuss about various approaches of writing algorithm which are as follows:

1. **Natural Language:** We can write the algorithm in natural language but the problem in this is that there can be ambiguity *for e.g*. let's see the statement "Sarah gave a bath to her dog wearing a pink t-shirt". Ambiguity: Is the dog wearing the pink t-shirt?
2. **Flow chart:** we can also write an algorithm using flowcharts but the problem is that whenever there is a need of change in algorithm, for modification in flowcharts we have to learn specialized tools for changing flowchart.
3. **Programming Language:** We can write algorithm in programming language but the problem is that we need to have all the details and internal language about programming language software, operating system.
4. **Pseudo code:** This method is the largest applicable one. It's very simple as its syntax less.

Let's take an example of writing an algorithm for addition of two numbers with the help of above stated methods.

Method 1. Algorithm in Natural language

Step 1: Start

Step 2: Declare variables a, b and sum.

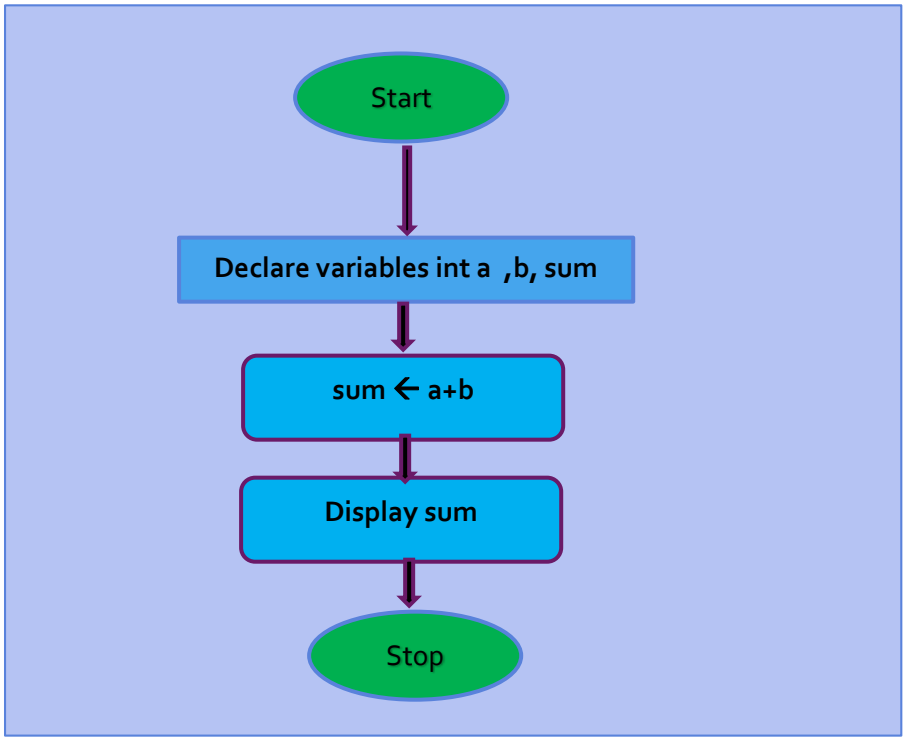
Step 3: Read values for a and b.

Step 4: Add a and b and assign the result to a variable sum.

Step 5: Display sum

Step 6: Stop

Method 2. Algorithm in terms of flowchart.



Method 3. Algorithm in terms of program.

```
int main()
{
  int a, b, sum;
  printf("Enter two numbers to add\n");
  scanf("%d%d", &a, &b);
  sum = a + b;
  printf("Sum of the numbers = %d\n", sum);
  return 0;
}
```

Method 4. Algorithm in terms of pseudo code

```
BEGIN
        Read a, b
        Set sum to a + b
END:
```

1.4. Characteristics of an Algorithm:

After studying the definition of an algorithm as well as its methods of writing, let's focus on the important characteristics of an algorithm.

- **Unambiguous** − The Algorithm should be written in such a manner that its intent should be clear.
- **Input** – There should be an initial condition that determines the starting position.
- **Output** − There should exist a point where we can reach the end state that is the desired goal.
- **Finiteness** – A good algorithm takes a definite number of steps before finishing.
- **Feasibility** – By feasibility, we mean that it should be solvable within a given set of resources.
- **Independent** − an algorithm should only describe the step-by-step procedure, and it does not need to discuss programming techniques.

Table 2: Representation of characteristics of algorithm using Pizza example

S.No.	Characteristic	Results
1.	**Unambiguous**	If chef is not cleared about the type of cheese (e.g. goat cheese, provolone, Aged Harvarti etc.) he has to use to prepare pizza, then he is at unambiguous stage.

2.	**Input**	Ingredients to make pizza will be considered as input. E.g. pizza base, cheese, pepperoni, Flour, Mozzarella etc.
3.	**Output**	The prepared pizza will be the output.
4.	**Finiteness**	As we have seen above in the flowchart for procedure to prepare pizza, the number of steps is finite.
5.	**Feasibility**	If the cook is able to prepare pizza with available set of resources using above stated algorithm, then that algorithm is said to be feasible. For e.g., the algorithm is demanding a particular brand cheese which is not available in India, then that algorithm is not feasible for pizza making.
6.	**Independent**	The algorithm written for pizza making is using LG microwave oven functionalities but the user is having Samsung brand's microwave oven. So, the algorithm steps are not technology independent here.

Example 2: Let's look at a very simple algorithm to find maximum element in a given n-element array (Integer).

Problem: Given an array of positive numbers, return the largest number of the array.

Inputs: An array A of positive numbers. This array must contain at least one number.

Outputs: A number n, which will be the largest number of the array.

Algorithm:

- **Set Max to 1.**
- **For each number x in the array A, compare it to Max. If x is larger, set Max to x.**
- **Max is now set to the largest number in the list.**

Pseudo Code:

```
Find_Max (intA[])
{
        Int i, n, Max;
        n=length(A);
        Curr_Max = A[1];
        For(i=2;i<=n;i++)
        {
                If(A[i]>=Curr_Max)
                        Max = A[i];
        }
return (Max);
}
```

Representation of the above example in terms of algorithm characteristics:

- Algorithm defined to find the largest element in an array is unambiguous as all steps lead to a unique interpretation.
- Algorithm contains defined inputs (input is array) and output (Max element is the output).
- Array is of finite length, so after looking at every element of the array the algorithm will stop/terminate. This implies that algorithm is finite.
- Algorithm will provide the feasible solution after verifying its correctness. This shows that the algorithm complies to feasibility too.

1.5. Applications of Algorithm:

- **Solving Everyday Problems**: We apply various algorithms in our daily life for example brushing our teeth, making sandwiches, etc.
- **Recommendation System**: Recommendation algorithm for Facebook and Google search, searching large aadhar based data set to come in this category.
- **Finding Route / Shortest Path**: To search for any information, Google recommends using a recommendation-based algorithm. To move from one place to another in minimum time, we use the shortest path algorithm.
- **Recognizing Genetic Matching**: To recognize the DNA structure of humans, we need to identify 100000 genes of human DNA for determining the sequences of the 3 billion chemical base pairs that make up the human DNA.
- **E-Commerce Categories**: The day-to-day electronic commerce activities are hugely dependent on algorithm, we use to do online shopping, and for this, we provide our details like bank details, electronic card details, and for this, we need a good Algorithm to predict day to day changes in users' choice and provide them product/services based on their choice.

1.6. Qualities of a Good Algorithm:

The factors that we need to consider while determining the quality of a good Algorithm are:

- **Time**: The amount of time it takes an algorithm to run as a function of the length of the input is known as time requirement. The number of operations to be done by the algorithm is indicated by the length of input.
- **Memory:** The amount of memory utilised by the algorithm (including the inputs) to execute and create the output is referred to its memory requirement.
- **Accuracy:** There can be more than one solution to the given problem, but the one which is the most optimal is termed as accurate.

Again, in the following manner, the quality of the procedure for preparing pizza can be assessed:-

- The time it takes to prepare pizza for one person should be between that of a first-time pizza maker (worst case time) and that of a highly skilled chef (best case time).
- The amount of space necessary to pack a pizza in a box should be optimised such that it fits perfectly in the box without minimum wastage of space.
- Pizza can be prepared in a variety of ways. However, the method that optimises the cook's motions in the kitchen, using fewer ingredients to produce the appropriate level of quality pizza with the least amount of money and time, will be the most accurate.

2.1. Performance Analysis:

The main goal to study this subject is to judge an Algorithm, as we already discussed that we can judge an algorithm on the basis of computing time and storage requirement. Analysing the algorithm depend on the space (storage) complexity (space acquired by the algorithm) and time taken for algorithm execution (time complexity).

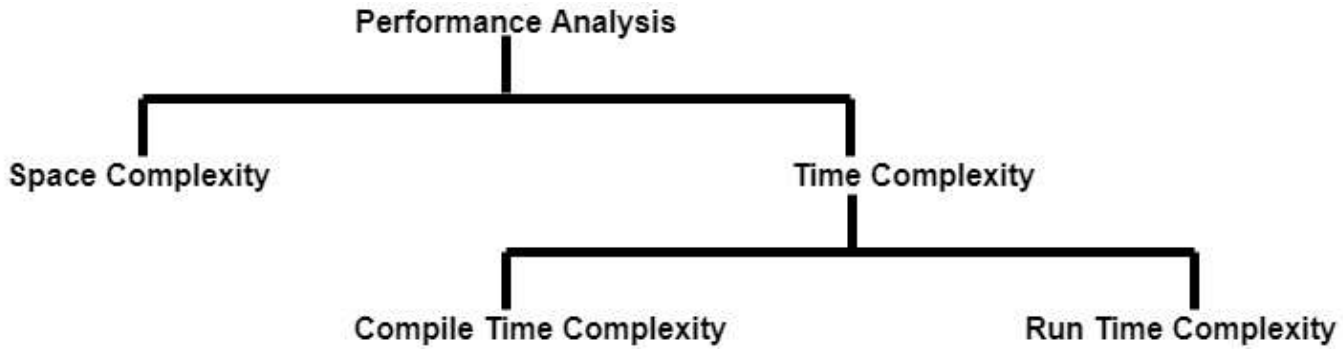


Figure 1.5: Analysis of Algorithm in terms of time and space complexity

The different types of complexity are represented *in figure 1.5* above

2.1.1 Space Complexity:

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input. Auxiliary space is the extra space or temporary space used by an Algorithm.

We consider following components to calculate space required by the Algorithm-

i) **Fixed Part: -**fixed part is space for code this is independent of the input and output. This part includes the instruction space, space for simple variables, space for constant, so on.

Example:

Int n ; → Memory taken by the n is fixed (2 for 32-bit compiler and 4 for 64 bit) compiler).

ii) **Variable Part: -**Variable part that consists of the space needed by component variable whose size is dependent on the particular problem. This part includes space needed by referenced variables and the recursion stack space.

Example:

```
Algorithm fact(int n)
{
If (n==0)
 return 1;
return n* factu(n-1)'
}
```

Memory taken by the Algorithm fact is dependent upon the value of n.

2.1.2 Time Complexity

The time complexity T(P) taken by a program P is the sum of the compile time and run (execution) time. Compiler time does not depend on the instance characteristics. Also we assume that a compiled program will be run several times without recompilation. That is why we focus only on Run (Execution) Time.

Example: Given the set of instruction (Pseudo code), Compute time complexity

Statement	Step/execution (s/e)	Frequency	Total step
Algorithm sum(a, n)	0	-	0
{	0	-	0
s=0.0;	0	1	0
for i=1 to n do	1	n+1	n+1
s=s + a [i];	1	n	n
return s;	0	1	0
}	0	-	0

Time complexity/ time taken by the above code = 0 + 0 + 0 + n+1 + n + 0 + 0 =2n + 1

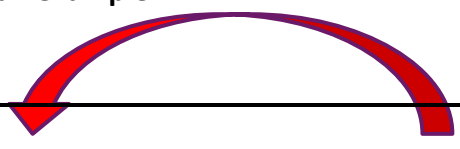
2.2 Common Mathematical Functions Used in Complexity Analysis:

Before analysing algorithm, we learn some common function. We will use this function to analyse Algorithm.

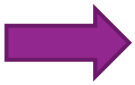
a. Constant Function *f*(*n*) = C

For any argument *n*, the constant function *f*(*n*) will execute some finite set of instructions only. The Algorithm that takes constant time does not depend on the input size.

Let's take an example-



```
for (i=0;i<100;i++)
{
printf (“%d”, *);
 }
```



for loop will execute 100 times every time. Execution of loop is fixed (constant) times.

Complexity of above mentioned example is constant.

b. **Logarithm Function *f*(*n*) = log *n***

For some change in input, the function grows as a function of logarithmic. This function is defined as follows:

```
for(i=2;i<n;i^2 )
{
printf (“%d”, *);
}
```

loop will execute in below sequence. Assume that n=500

i	2	4	16	256	65536
True/ false	2<500(true)	4<500(true)	16<500 (true)	256<500(true)	65536>500(false)

Now we can see that i increases in i^2 manner and the value of i^2 must be less or equal to n.

$i^2 <= n$

Take log both side

2 log i <= log n

i <= (log n)/ (log 2)

$i <= \log_2 n$

Then the complexity of above example is order of $\log_2 n$.

c. Linear Function *f*(*n*) = *n*

The Linear Function grows in a constant order. In another word it, it is a function which grows as a function whose graph can be plotted linearly.

```
for(i=0;i<n;i++)
{
        printf(“%d”, *);
}
```

For loop will execute n times from 1 to n. Hence, running time complexity will be n

d. Function *f*(*n*) = *n* log *n*

This type of function growth is bounded between linear growth and those functions which grow to exactly square of the input provided.

```
for(i=0;i<n;i++)
{
    for(j=2;j<n;j^2)
      {
       printf("%d", *);
      }
}
```

There are two nested loops
When i=0 then second loop will execute $\log_2 n$ times (already discussed).
Again, when i=1 then second loop will execute $\log_2 n$ times again.
Similarly, first loop will execute n times and each time second loop will execute $\log_2 n$ times.
So, the total execution is n* $\log_2 n$ times.

e. Quadratic Function $f(n) = n^2$

All those functions grow to like when we double the input size, and the function grows four times. For example, in nested loops,

```
for(i=0;i<n;i++)
{
        for(j=0;j<n;j++)
        {
                printf("%d", *);
          }
}
```

There are two nested loops.
When i=0 then second loop will execute n times (already discussed).
Again, when i=1 then second loop will execute n times again.
Similarly, first loop will execute n times and each time second loop will execute n times.
So, the total execution is n* n times.
So, the complexity is order of n^2.

f. Cubic Function and Other Polynomials:

This function grows faster than its counterpart that is Linear Search and quadratic equation. If we double the size of the input variable, then the function will grow eight times; hence the rate of growth will be cubic.

```
for(i=0;i<n;i++)
{
        for(j=0;j<n;j++)
        {
                for(k=0;k<n;k++)
                {
                        printf("%d",
                 *);
                }
        }
```

There are three nested loops.
When i=0 then second loop j=0 and third loop execute k=0 to n-1 times (total n times).
Again, when i=0 then second loop j=0, 1, 2, 3……n-1 will execute n times and k=0, 1, 2, 3……n-1.
Similarly, first loop will execute n times, second loop will execute n times and Third loop will execute n times.
So, the total execution is n* n* n times. So, the complexity is order of n^3.

g. Factorial Function *f*(*n*) = *n*!

The running time of this function is worst among all discussed till so far? If we are going to increase the value of n, then the function will grow using the concept of the Factorial function.

Example: permutations of *n* elements.

Permutation (a) = {a}
Permutation (ab) = {ab, ba}
Permutation (abc) = {abc, acb, bac, bca, cab, cba}

n=1, n! = 1! = 1
n=2, n! = 2! = 2*1 = 2
n=3, n! = 3! = 3*2*1 = 6

2.3 Growth of Functions:

The growth of function shown *in figure 1.6,* provides the understanding of the Algorithm's efficiency in order to understand its performance. The aim is to predict the performance of different algorithms in order to predict their behaviour with changes in input.

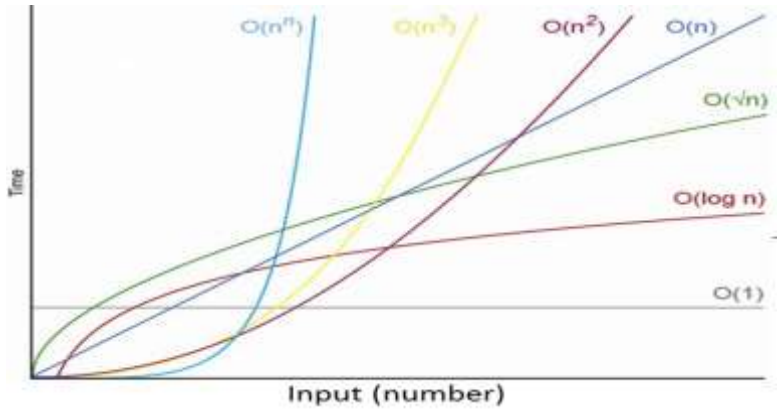


Figure 1.6: Representing the growth of various mathematical functions over input size n

Table 3: With the help of above diagram, we can compare the different Functions

Function	Remark
Constant (C)	Always order of 1. No matter value of C is 1, 10, 100, 1000 or any number
Logarithmic (log n)	Complexity is order of log n. If n=10 then log(10)=1 If n=100 then log(100)=2 **Faster than constant(C)**
Square root(√n)	If n=100 then √n= 10 If n=10000 then √n=100 **Faster than logarithmic (log n)**
Linear (n)	Complexity is order of n. Always grow in linear manner. If n=10 then complexity is 10 If n=100 then complexity is 100 Complexity is dependent on the value of n, and n is variable. **Faster than square root(√n)**
Quadratic (n^2)	Function growth is fast

	If n=2 then complexity is 4 If n=3 then complexity is 9 If n=10 then complexity is 100 **Faster than Linear (n)**
Cubic (n^3)	If n=2 then complexity is 8 If n=3 then complexity is 27 If n=10 then complexity is 1000 **Faster than Quadratic (n^2)**
Exponential (2^n)	If n=2 then complexity is 4 If n=3 then complexity is 8 If n=10 then complexity is 1024 It is slow in starting but for larger value it is **Faster than Cubic (n^3)**
Factorial (n!)	If n=2 then complexity is 2*1=2 If n=3 then complexity is 3*2*1=6 If n=10 then complexity is 10*9*8*7*6*5*4*3*2*1 = 36,28,800 **Faster than Exponential (2^n)**

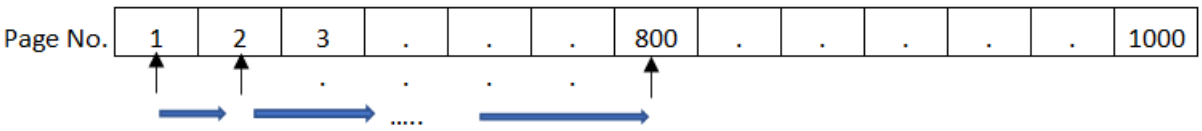
Comparison of various mathematical functions will result as follows:
Constant < log n < n < n^2 < n^3 < 2^n < n!

2.4. Why Analysis of Algorithms is important:

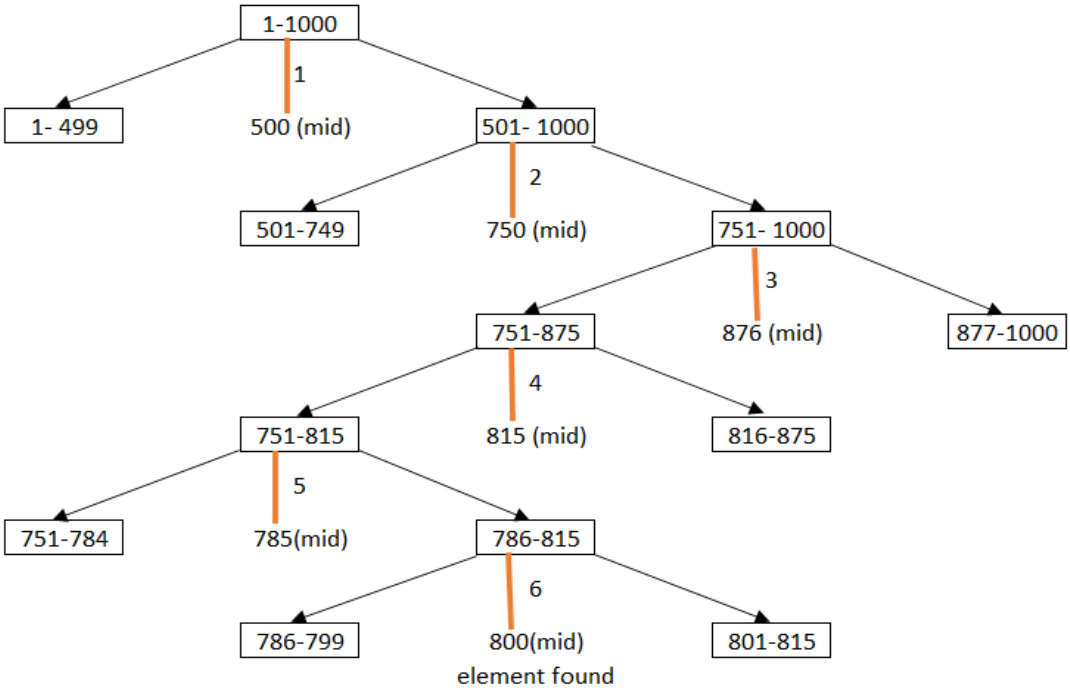
Let's understand the importance for performing the analysis of algorithm with the help of an example:

Example1: (Dictionary)
Let us assume that we have a dictionary of 1000 pages and we want to search a word "*Space*" in dictionary. This word is at 800th page number.

First method: we turn the pages one by one and after turning 799 pages we got the word "*Space*".
In this method we made 799 comparisons.



Second Method: we open the dictionary from the mid (500) now I know that word "*Space*" is in second half (501 to 1000), then again, I divide the pages (501-1000) from the mid (750). Now search the word between page 750 to 1000. Repeat this step till reach page 800.



Red line represents the page number and number written with line represent the comparison. With the help of this example, we can say that we need only 6 comparisons to reach page number 8, but in previous method 799 comparisons was required to reach page number 800.

Now the question arises: which method is better?

> **In day-to-day life we solve many problems and there are many possible ways to solve any problem. To find a best possible solution among them is analysis.**
>
> **So, we can say that if we have any many possible solution of any problem and we analyse all the solution to select best one to reduce the complexity (made it easy) in terms of time and space**

In order to analyse the solutions of any problem we have to understand three cases

Three cases to analyse any problem:

1. Best Case
2. Average Case
3. Worst Case

To understand these cases,
Problem: Consider given array of random elements, we have to categorize best, worst and average case.

Index	0	1	2	3	4	5	6	7	8	9
element	5	7	9	12	15	1	8	10	6	4

Best Case:

- Search element 5 in given array and check element one by one.
- Start with index 0 and found the element at first attempt.

What could be better than this that element found in first attempt? This is best condition to search any element.

Average Case:

- Search element 15 in given array and check element one by one.
- Start with index 0. First search at 0th index not found.
- Search at 1st index not found
- Repeat this step till element found.
- Element 15 found at 4th index.

Size of array= 10 (0 to 9)
Number of comparisons required= 5 (0, 1, 2, 3, 4)
Almost half of the array size comparison required to search element (n/2 where n is size of array)
This is average case where we found the element after n/2 comparisons.

Worst Case:

- Search element 4 in given array and check element one by one.
- Start with index 0. First search at 0th index not found.
- Search at 1st index not found
- Repeat this step till element found.
- Element 4 found at last index.

Size of array= 10 (0 to 9)
Number of comparisons required= 10 (0, 1, 2, ….9)
In this case we have traversed complete array and element found at last index. Maximum comparison can be made is size of array this is worst case.

Searching another element in an array

- Search element 19 in given array and check element one by one.
- Start with index 0. First search at 0th index not found.
- Search at 1st index not found
- Repeat this step till element found.
- Element 19 not found in array.

Size of array= 10 (0 to 9)
Number of comparisons required= 10 (0, 1, 2, ….9).
Element 19 is not in array but total number of comparisons made = maximum possible comparison = array size. This is also an example of worst case.

Now conclusion is that

- **If Minimum number of steps (comparisons) required to solve a problem (Best Case).**
- **If Average number of steps (comparisons) required to solve a problem (Average Case).**
- **If Maximum number of steps (comparisons) required to solve a problem (Worst Case).**

2.5. Asymptotic Notation:

As we know that we have millimetre, centimetre, metre or kilometre to measure distance similarly we want unit for measuring complexity. Asymptotic Notations are the units to measure complexity. Asymptotic Notation is a way of comparing functions that ignores constant factors and small input sizes. Let's understand it with the searching technique. We need to focus on how much time an algorithm takes with the change in **terms of the input** size. As the input size increases, we can easily find the Binary Search will provide better results than Linear Search. So here, the size of input matters to find the number of comparisons required in a worst-case scenario.

Analogy:- GPS:- As show in in figure 1.7 below , if GPS only knew about highways or interstate connected highway systems, and not about every small or little road, then it would not be able to find all routes as accurately as it is finding nowadays. Hence, we can see that the running time of the function is proportionate to the size of the input.

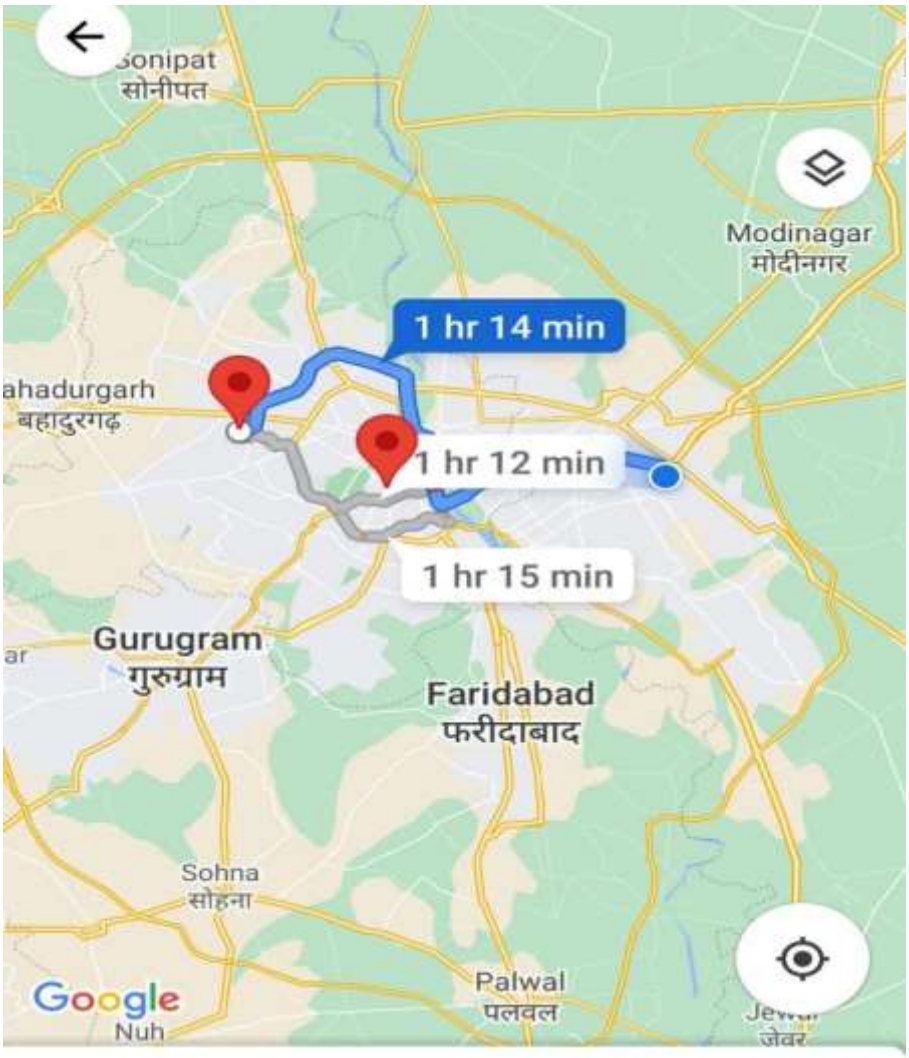
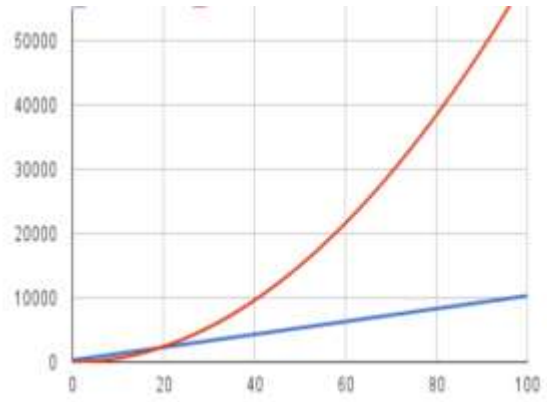


Figure: 1.7 - Google Map

The other consideration is how fast a function grows with an increase in the size of the input. We need to include more essential things, and we need to drop the less important ones.

Example: Let's understand this with the help of one example, one Algorithm which runs on a system takes 6n^2 + 50n + 80 machines instructions. The term 6n^2 becomes larger than 50n + 80 when n reaches 10. The given chart below compares 6n^2 and 50n + 80.



6n^2

5on + 8o

From the given graph, it is observed that 6n^2 will take more time, comparative to 50n+80.for the value of n>=10

6n^2 will take more time.

Hence running time complexity of this, will be n^2

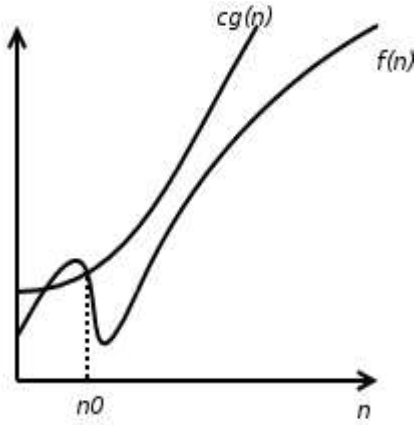
As we are increasing the input size, that is, the value of n, the value of term n^2 will be far larger than the value of term 50n + 80. Hence, we can drop 50n + 80 as this term would not affect the larger value of n, and the function grows with n^2.

When we drop these constant and less significant terms, we use asymptotic notations, and these are **Big-oh notation, Big-omega notation, Big-theta Notation, and their variants; we will discuss this one by one in the next section:-**

2.6. Types of Notation:

2.6.1. O-notation / Big-oh notation/ Upper bound 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.



There exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$." f(n) is thus O(g(n)).
O (g (n)) = {f(n): there exist positive constants c and n_0 such that
$0 \le f(n) \le c\, g(n)$ for all $n \ge n_0$ }

The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and CG(n), for sufficiently large n.
For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).
Since it gives the worst-case running time of an algorithm, it is widely used to analyse an algorithm as we are always interested in the worst-case scenario.

Example on Big Oh Notation:

1. **Given f(n)=3n+5, then f(n)=O(n)**

   Solution 1: As per def.: $f(n) \leq C.g(n), \forall\, n \geq n0$
   From given problem statement, f(n)=3n+5 and g(n)=n
   Now there exists a constant such that value of c>0 and n≥ 1
   So, let's write the equation as 3n+5<=C. n
   On solving the in-equality we get C=4
   Then checking for different values of n
   Case 1: n=1, C=4
   $3(1)+5 \leq 4(1)$
   8≤4 (False)
   Case 2: n=2, C=4
   $3(2)+5 \leq 4(2)$
   11≤ 8 (False)
   Case 3: n=3, C=4
   3(3)+5≤ 4(3)
   14≤ 12 (False)
   Case 4: n=4, C=4
   $3(4)+5 \leq 4(4)$
   17≤ 16 (False)
   Case 5: n=5, C=4
   3(5)+5≤ 4(5)
   20≤ 20 (True)
   So from case 5 f(n)=O(n) ∀ n ≥ 5

2. **Given f(n)= n^3 , then f(n)≠O(n2)**

   Solution 2: As per def.: $f(n) \leq C.g(n), \forall\, n \geq n0$
   From given problem statement f(n)= n^3 and g(n)=n^2
   Now there exists a constant such that value of c>0 and n≥ 1
   So, let's write the equation n^3<=C. n^2
   If we assume C=1 then n=1 (condition is true) but for any other value of n condition is false.

3. **Given f(n)= $2n^2$ + 5n +1, Prove that f(n)=O(n^2)**

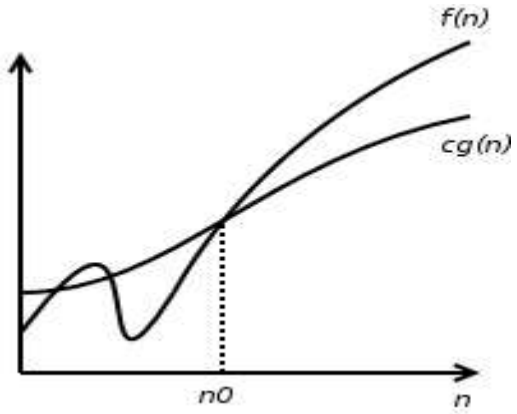
   Solution 3: As per def.: $f(n) \leq C.g(n), \forall\, n \geq n0$
   From given problem statement f (n) =$2n^2$ +5n +1 and g (n) =n^2

Now there exists a constant such that value of c>0 and n≥ 1
So let's write the equation as $C(n^2) \geq 2n^2 + 5n + 1$
We have to find the value of C
$C \geq 2\ n^2/\ n^2 + 5n/\ n^2 + 1/\ n^2$
We get $C \geq 1+5/n+1/\ n^2$
On putting different values of n we get C=4
Now check the value of n for different cases:
Case 1: n=1, C=4
$2(1^2)+5(1)+1 \leq 4(1)$
$8 \leq 4$ (False)
Case 2: n=2, C=4
$2(2^2)+5(2)+1 \leq 4(2^2)$
$19 \leq 16$ (False)
Case 3: n=3, C=4
$2(3^2) + 5(3) + 1 \leq 4(3^2)$
$34 \leq 36$(True)
So, from case 3 $f(n)=O(n^2)\ \forall\ n \geq 3$

4. **Given $f(n)= 4^n$, Prove that $f(n)=O(8^n)$**
   Solution 4:
   As per def.: $f(n) \leq C.g(n), \forall\ n \geq n0$
   From given problem statement f (n) $=4^n$ and g (n) $=8^n$
   Now there exists a constant such that value of c>0 and n≥ 1
   So let's write the equation as $4^n \leq C\ 8^n$
   $=4^n/8^n \leq C$
   $=1/2^n \leq C$
   So, try for different values of n i.e. 1,2 ---n, we get 1, ¼….
   So, let's choose C=1
   Case 1: n=1, C=1
   $4^n \leq 8^n$
   $4 \leq 8$ (True)
   So, from case 1 $f(n)=O(8^n)\ \forall\ n \geq 1$

2.6.2. Ω-notation/ Big-omega notation:

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



There exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$. f(n) is thus Ω(g(n)).
Ω(g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$ }

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above CG(n), for sufficiently large n

For any value of n, the minimum time required by the Algorithm is given by Omega Ω(g(n))
Analysis: - The above code has a runtime of O(n). the function f(i) is called for **at most** n times. The upper bound is n, and the lower bound can be Ω (1) or Ω(log(n)), depending on the value of foo(i).

Examples on Big Omega Notation:

1. **Given f(n)=3n+5, then f(n)=Ω(n)**

   Solution: f(n)=3n+5
   g(n)=n
   As per Big-Omega theory,
   C.g(n) <=f(n)
   C. n <=3n+5
   For C=1,

2. **Given f(n)= n^2 , then f(n)≠Ω(n^3).**

   Solution: f(n)= n^2
   g(n)=n^3
   By inequality,
   n^3 >=n^2
   or, 1. n^3 >=n^2
   or, 2. n^3 >=n^2
   for different value of C=1,2,3,..
   C n^3 >=n^2
   For any value of C and n condition is false.

3. **Prove that $10n^2 + 3n + 3 = \Omega(n^2)$**

   Solution: In the given equation, f(n)= $10n^2 + 3n + 3$ *and* g(n)= n^2

LHS=$10n^2 + 3n + 3$

RHS= $\Omega(n^2)$

By inequality,

$10n^2 + 3n + 3 > n^2$

Or,

$10n^2 + 3n + 3 > 1.n^2$

Where c=1

Further, one can find value for LHS and RHS for n=1,2,3…

For n=1, LHS=10 and, RHS=1

n=3 LHS=$10.3^2 + 3.3 + 3 = 90 + 9 + 3 = 102$

RHS=9

Hence, by definition of Big-Omega we can write,

$\boldsymbol{10n^2 + 3n + 3 = \Omega(n^2)} \quad \forall \boldsymbol{n > 1}$

4. **Prove that $\boldsymbol{100n + 5 = \Omega(n^2)}$**

Solution: As per Big-Omega theory,

$Cg(n) \leq f(n)$

$given\ g(n) = n^2$

$f(n) = 100n + 5$

C$n^2 \leq 100n + 5$

C$\leq \frac{100}{n} + \frac{5}{n^2}$

$for\ n = 2\ , C \leq 50$

LHS=100.2+5=205

RHS=50.4=200<205

Hence proved, $\boldsymbol{100n + 5 = \Omega(n^2)}$

5. **Prove that $\boldsymbol{n^3 = \Omega(n^2)}$**

Solution. As per Big-Omega notation,

$0 \leq C.g(n) \leq f(n)$

$0 \leq C.\boldsymbol{n^2} \leq \boldsymbol{n^3}$

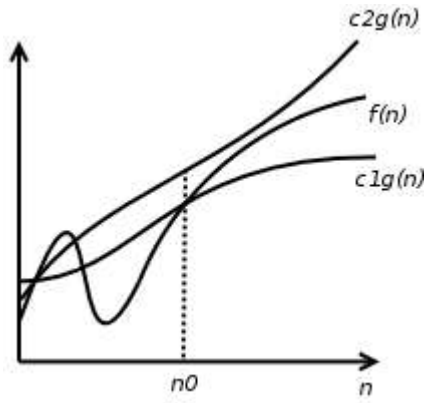
For C=1, and n>=1, $\boldsymbol{n^2 \leq n^3}$

For C=1, and n=2 $\quad \boldsymbol{2^2 \leq 2^3}$

$\boldsymbol{4}$ <8 (condition is true)

2.6.3. Theta notation / Tightly Bound:

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 analysing the average-case complexity of an algorithm.



There exist positive constants c_1, c_2 , and n_0 such that $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n > n_0$ ". f(n) is thus Θ(g(n)).
For a function g(n), Θ(g(n)) is given by the relation:
Θ(g(n)) = { f(n): there exist positive constants c_1, c_2 and n_0
such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ }

When we say that the function is Θ(n), then we are saying that once n gets large enough, the running time is at least $c_1(n)$ and maximum $c_2(n)$.

Examples on Theta Notation:

1. **f(n)= n^2 +3n+7, then g(n)= Θ(n^2)**

Solution. As per def.: C1.g (n) <=f (n) <=C2.g (n)
For the given problem statement, f (n) = n^2 +3n+7, g(n)=n^2
Now there exists a constants such that value of c>0 and n≥ 1
As per Big-Theta theory, C1.g (n) <=f (n) <=C2.g (n)
C1. n^2 <= n^2 +3n+7 <= C2. n^2
If C1=1 & C2=2 then n>=5

Hence, $n^2 + 3n + 7 = \Theta(n^2)$, $\forall n \geq 5$

2. **f(n)= 12 n^2 +3n+7, then g(n)= Θ(n^2)**

Solution. As per def.: C1.g (n) <=f (n) <=C2.g (n)
For the given problem statement, f(n)= n^2 +3n+7, g(n)=n^2
Now there exists a constants such that value of c>0 and n≥ 1
As per Big-Theta theory C1.g (n) <=f(n) <=C2.g(n)
C1. n^2 <= n^2 +3n+7 <= C2. n^2
If C1=1 & C2=2 then n>=5

Hence, $n^2 + 3n + 7 = \Theta(n^2)$, $\forall n \geq 5$

3. **Prove that $\frac{n^2}{2} - 2n = \Theta(n^2)$**

Solution: As per def.: C1.g (n) <=f (n) <=C2.g (n)

For the given problem statement, f(n)= $\frac{n^2}{2} - 2n$, g(n)=n^2

Now there exists a constants such that value of c>0 and n≥ 1

As per Big-Theta theory C1.g(n) <=f(n) <=C2.g(n)

Here C1 and C2 are constant.

$C1.n^2 \leq \frac{n^2}{2} - 2n \leq C2.n^2$

Case 1: Solving inequality, $\frac{n^2}{2} - 2n \leq C2.n^2$

$$\frac{1}{2} - \frac{2}{n} \leq C2$$

$$for\ n \to \infty, or\ very\ large\ value\ C2 \geq \frac{1}{2}$$

Case 2: Solving inequality**,** $C1.n^2 \leq \frac{n^2}{2} - 2n$

or, $C1.n^2 \leq \frac{n^2}{2} - 2n$

$or,\ C1 \leq \frac{1}{2} - \frac{2}{n}$

$for\ C1 = \frac{1}{4} and\ C2 = \frac{1}{2}\ and\ n = 8$

$$\frac{1}{4}.8^2 \leq \frac{8^2}{2} - 2 * 8 \leq \frac{1}{2}.8^2$$

$or, 16 \leq 32 - 16 \leq 32$

$or,\ 16 \leq 16 \leq 32$ (True condition)

4. **Show that $(n + a)^b = \Theta(n^b)$ where b>0**

Solution: As per def.: C1.g (n) <=f (n) <=C2.g (n)

For the given problem statement, f (n) = $(n + a)^b$**,** g(n)= n^b

Now there exists a constants such that value of c>0 and n≥ 1

As per Big-Theta theory C1.g (n) <=f (n) <=C2.g (n)

Note that, for n>a,

$$n + a \leq 2n$$

and

$$n + a \geq n/2$$

Thus,

$$n/2 \leq n + a \leq 2n$$

By taking raise to power 'b', on both side,

$$0 \leq (\frac{n}{2})^b \leq (n + a)^b \leq (2n)^b$$

$$0 \leq (\frac{1}{2})^b (n)^b \leq (n + a)^b \leq (2)^b (n)^b$$

Thus value of c1=$(\frac{1}{2})^b$, c2=2^b and n=2a,

Hence, proved.

5. **Prove that $2n - 2\sqrt{n} = \theta(n)$**

Solution: As per def.: C1.g (n) <=f (n) <=C2.g (n)

For the given problem statement, f (n)=$2n - 2\sqrt{n}$, g(n)= n

Now there exists a constants such that value of c>0 and n≥ 1

As per Big-Theta theory C1.g (n) <=f(n) <=C2.g(n)

Case 1: Solving inequality, $2n - 2\sqrt{n} \leq C2.n$

$C2 \geq 2\text{-}2\sqrt{n}/n$

C2=2

Case 2: Solving inequality, $C1.n \leq 2n - 2\sqrt{n}$

$C1 \leq 2\text{-}(2\sqrt{n})/n$

C1=1

$for\ C1 = 1 and\ C2 = 2\ \ and\ n = 4$

$$1(4) \leq 2(4) - 2\sqrt{4} \leq 2(4)$$
$$4 \leq 8 - 4 \leq 8$$

$4 \leq 4 \leq 8$ (True)

2.7 Solved Examples on Pseudo Code Complexity Analysis

Let's understand the concept discussed yet with the help of some examples.

Example 1: Compute the running time complexity for the given pseudo code:

Statement	Step/execution (s/e)	Frequency	Total step
Algorithm sum(a,n)	0	-	0
{	0	-	0
s=0.0;	0	1	0
for i=1 to n do	1	n+1	n+1
s=s+a[i];	1	n	n
return s;	0	1	0
}	0	-	0

Time complexity/ time taken by the above code = 0 + 0 + 0 + n+1 + n + 0 + 0 =2n + 1=O(n)

Example 2: Compute the running time complexity for the given pseudo code:

Statement	s/e	Frequency	Total step
Algorithm Add(a,b,c,m,n)	0	-	0

{	0	-	0
For i=1 to m do	1	m+1	m+1
for j=1 to n do	1	m(n+1)	m(n+1)
c[i , j]= a[i, j]+b[i, j]	1	Mn	mn
}	0		

Time complexity/ time taken by the above code = 0 + 0 + m+1 + m(n + 1) + mn
=m + 1 + mn + m + mn
= 2m + 2mn + 1= 2mn= O(mn)

Example 3: Compute the running time complexity for the given code:

```
#include <stdio.h>
int main(void)
{
        int sum=0,i;
        for(i=1;i<=n;i=i+2)…… ..n+1 times (n for true condition and 1 for false condition)
        {
                sum=sum+i; …….. n time (only for true condition)
        }
 return 0;
}
```

Then the complexity of above code is O (n)

Example 4: Compute the running time complexity for the given code:

```
#include <stdio.h>
int main(void)
{
        int sum=0,i;  …….. 0 time
        for(i=1;i<=n;i=i*2)
        {
                sum=sum+i;
         }
 return 0;
}
```

Let n = 20 then for loop will run from 1 to 20

I	1	2	4	8	16	32
N	1<20	2<20	4<20	8<20	16<20	false

So $i <= n$ and $i=2^k$ {k=0,1,…..}

$2^K <= n$ take log

$K \log 2 <= \log n$

$K <= \log_2 n$

Then the complexity of above code is $O(\log_2 n)$

Example 5: Compute the running time complexity for the given code:

```
#include <stdio.h>
int main(void)
{
        int sum=0,i;  ..... 0 time
        for(i=n;i>0;i=i/2)
        {
                sum=sum+i;
        }
 return 0;
}
```

Let n = 10 then for loop will run from 1 to 20

I	20	10	5	2	1	0
N	20>0	10>0	5>0	2>0	1>0	false

=> n , n/2 , n/4 , n/8 , ...

=> $n/2^0$, $n/2^1$, $n/2^2$, ...

=> $n(½)^k$ k=(0,1,2,....),

2^k =n take log both side

K log 2 = log n

K= $\log_2 n$

Then the complexity of above code is O ($\log_2 n$).

3.1 Why we study design approaches:

After discussing the basics of the algorithm along with various cases of performance analysis, now we will discuss about the various types of design approaches used in the algorithm.

3.1.1 Calculating the GCD:

Problem Statement: Given two numbers, 270 and 192. Find the GCD of the two numbers using an optimized approach.

First Approach: Number Theoretic Algorithms: For finding the GCD of two numbers, Step by Step Process of the Euclidean Algorithm is as follows:

Case 1. If number A = 0 then GCD (A, B) = B, since the GCD (0, B) =B, and we can stop.
Case 2: If number B = 0 then GCD (A, B) = A, since the GCD (A, 0) = A, and we can stop.
Then calculate the quotient remainder as (A = B·Q + R)
Now, we have to calculate GCD(B,R) using Euclidean Algorithm since GCD(A,B) = GCD(B,R)

Example:

i. Find the GCD of two numbers, A= 270 and B=192

- A=270, B=192
- A is not equal to 0
- B is not equal to 0
- Dividing A with B, 270/192 = 1 leave remainder of 78. This can be written as : 270 = 192 * 1 +78

ii. ***Find GCD(192,78), since GCD(270,192)=GCD(192,78***)

- A=192, B=78
- A is not equal to 0
- B is not equal to 0
- Dividing A with B, 192/78 = 2 leave remainder of 36. This can be written as: 192 = 78 * 2 + 36

iii. Find GCD(78,36), since GCD(192,78)=GCD(78,36)

- A=78, B=36
- A is not equal to 0
- B is not equal to 0
- Dividing A with B, 78/36 = 2 leave remainder of 6. This can be written as: 78 = 36 * 2 + 6

iv. Find GCD(36,6), since GCD(78,36)=GCD(36,6)

- A=36, B=6
- A is not equal to 0
- B is not equal to 0
- Dividing A with B, 36/6 = 6 leave remainder of 0. This can be written as : 36 = 6 * 6 + 0

v. Find GCD(6,0), since GCD(36,6)=GCD(6,0)

- A=6, B=0
- A is not equal to 0
- B =0, GCD(6,0)=6
- So we have shown:
- GCD(270,192) = GCD(192,78) = GCD(78,36) = GCD(36,6) = GCD(6,0) = 6
- GCD(270,192) = 6

The computed running time of the Euclidean Algorithm **is O(log n)**.

Second Approach: Calculating the GCD using Iterative Method

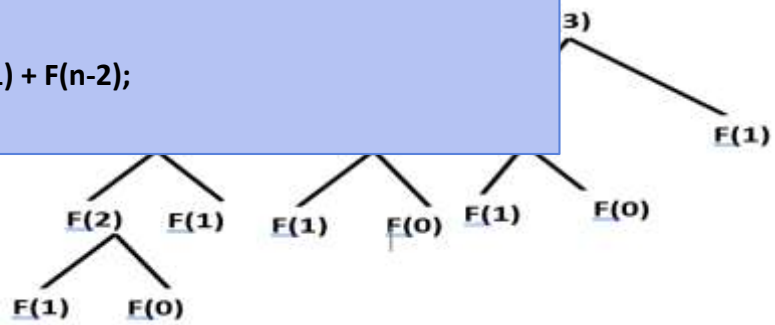
There are several algorithms to calculate the GCD (Greatest Common Divisor) between two numbers. The easiest and fastest process consists of decomposing each one of the numbers in products of prime factors, this is, and we successively divide each one of the numbers by prime numbers till we reach a quotient that equals 1.

Let us consider an example for calculating GCD between 168 and 180.
Step 1: Start by factorizing each number. Factorizing enables us to conclude that 168=2^3×3×7 and that 180=2^2×3^2×5.
Step 2: The following step is getting the product of common factor with a smaller exponent: 2×2×3=1×2×2×3=12.
Step 3: So, we can conclude the Greatest Common Divisor between 168 and 180 equals 12.
Result : The time complexity to calculate the GCD using an iterative approach is linear.

3.1.2. Fibonacci numbers:

Problem Statement: Numbers which follow the mentioned integer sequence is called as the Fibonacci numbers. For example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and 144. In mathematical terms, the sequence F_n of Fibonacci numbers is defined as $F_n = F_{n-1} + F_{n-2}$ with initial values $F_0 = 0$ and $F_1 = 1$.
Input: n = 3
Output: 2
Input: n = 11
Output: 89
Approach (Using recursion): This is the simple method used for representing the recursive function:

```
int F(int n)
{
  if (n = 0)
          return 0;
else if (n=1)
return 1;
else
          return F(n-1) + F(n-2);
}
```



Observation: For calculating the new number, it depends on the sum of the previous two numbers. For calculating the value of the nth number, we depend on the (n-1) th and (n-2) numbers. From this, we can simply say that the cost for calculating the nth Fibonacci number will be exponential.

3.1.3 Travelling Salesman Problem:

Problem Statement: Suppose there were n cities and the Distance/cost from one city to another city is given *in figure 1.8* below. The salesman needs to visit n cities in such a manner that they can start the journey from any city and visit all the cities exactly once during the tour and comes back to the city from where the journey started with the minimum distance/cost. The problem is to find out the minimum cost/distance salesman will take while traveling through all n cities.

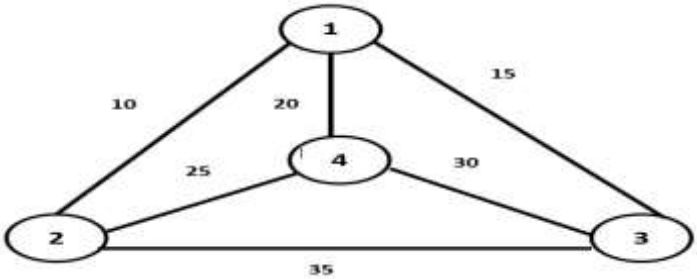


Figure 1.8: Representation of Travelling Salesman Problem

Example: In the given graph, we are given 4 cities. Traveling salesman tour of the given graph is 1-2-4-3-1. A cost that appears from the given tour is 10+25+30+15, which sums up to 80. There are various solutions to the traveling salesman problem. TSP problem can be solved using the Naïve method/Brute force method. It can also be solved using various Algorithmic design strategies.

1. Naïve/ Brute Force Solution: Step by Step procedure of Naïve/Brute Force Solution:

- First Step is to select City 1 as the starting city of the tour and the end city of the tour.
- As this is the brute force approach which means trying out all the possibilities. Here, we will calculate all the possibilities or the tour generated with the 4 cities. We can simply say that there may be 6 possible tours with 4 cities, calculated using (n-1)!
- After calculating the 6 possibilities of the tours from 4 cities, the Next Step calculates the cost of all the six tours.
- Last Step is to return the tour cost, which is minimum among all 6 tours.

Observation: The cost that occurs while trying out all the possibilities for calculating the number of tours from n cities is n!

2. Solutions that seem best at a particular moment based on the greed of a salesman:

Step by Step procedure of finding minimum cost tour based on the greed:

- First Step is to select City 1 as the starting city of the tour and the end city of the tour.
- After selecting the first city, we will see the corresponding cities which can be reached from the first city. Pick that city whose cost is minimum from city 1 based on the greed of the salesman.
- After selecting the next city, we will see the corresponding cities which can be reached from the next city. Pick that city whose cost is minimum from the city 2 based on the greed of the salesman. If any of the cities are already visited, then pick the city with the next minimum.
- Repeat Step 2 and Step 3 until we find the city from where our journey started.

Observation: Above mentioned approach is the greedy approach in which we pick the solution that seems best at the particular moment rather than focussing on whether this solution is optimal or not. In this, we always pick the city which appears at the minimum cost to the other city. But here, a question arises that picking minimum cost city will also lead to produce the minimum cost tour. The solution to this question is "NO" because there may be the possibility of having a minimum cost from some other tour as well where we have not picked the minimum cost city. This is the greedy approach

.

3. Solutions based on the recursive approach:

Step by Step procedure of finding minimum cost tour based on the recursive process:

- Given n cities, we need to start the journey from city 1. Initial, traveling cost of city 1 is x.
- TSP distance between two cities is calculated based on the recursive function.
- As a number of the cities is in the subset of two cities, then apply a recursive function to compute the minimum cost on the base case.

- If the number of the cities is greater than 2, we will calculate the distance from the given city to the next city. Then the minimum distance is calculated recursively among the remaining cities.
- Finally, the Algorithm returns the minimum cost as the TSP solution.

Observations: This is the dynamic programming approach. In this, we pick the minimum of the two cities and keep on finding the further minimum on tour recursively. This may or may not provide the optimal solution. The cost that appears from the above-mentioned step-by-step procedure is $O(n^2*2^n)$.

3.2 Various Design Approaches:

a) Brute-force or Exhaustive search: This is the most common method of trying every possible solution to see the best. For Example:

Analogy:- A child is given a mobile phone having a lock of 4-digit PIN. He is asked to guess the exact pin to unlock it.

Figure 1.9 : Representing the keypad for mobile phone

To guess the exact pin _ _ _ _, shown in *figure 1.9*, these four places can be filled with 0-9 numbers only so that each place can be filed with 10 choices. So total possible guessing will be 10X10X10X10 = 10,000 choices. So, according to this type, he needs to check all the possible conditions to unlock it.

b) Divide and Conquer: A Divide and Conquer Algorithm repeatedly divides the given problem into subproblems and then solves those different subproblems to provide the solution of the original problem.

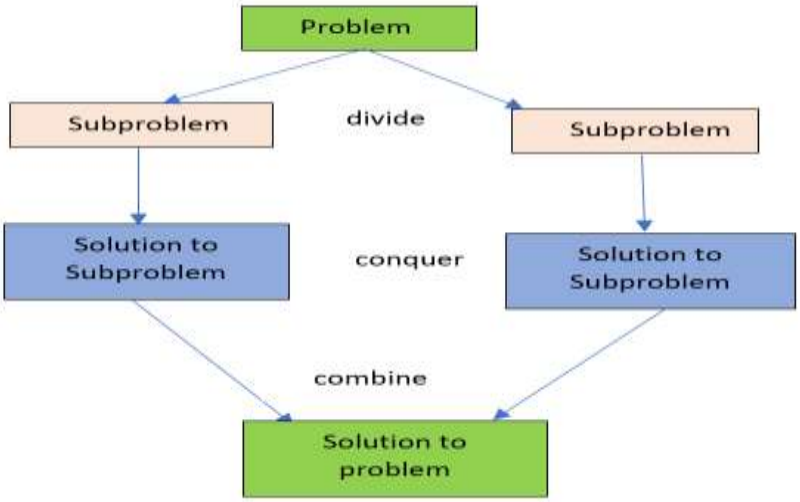


Figure 1.10: Representation of divide and conquer approach

In figure 1.10 above, we are dividing the above problem as below:

- **Divide:** Choose one number from each column for setting the first three-digit of the pin, and we can choose 0-9 for the last position.
- **Conquer:** When we pick 1 number out of three numbers from column1, 1 from 4 numbers in column2, 1 from available 3 from column 3, and anyone from 0-9 for the last digit.
- **Combine:** Possible combination, in this case, is 3X4X3X10 = 360. Now we can see that possible combination has reduced to a great extent.

c) Greedy Method:

In the Greedy Algorithm, the decision on each step is made based upon optimal utilization that can be made with the available resources, which is based on the maximum availability of that resource at that point.

Here, solutions to the sub-problem do not have to be known at each stage; instead, a "greedy" choice can be made of what looks best for the moment.

Analogy:-Get the best dress for your child from the shop.

Figure 1.12: Pictorial retransition of Greedy approach (in real life)

One day a lady visited a shop and saw another lady buying a beautiful dress for her kid. She played all tactics to ensure she does not buy that dress, and she, without much thinking, purchased it. But as soon as she left the shop after paying the money, she finds that the woman has purchased the more beautiful

dress for her kid and at a lower price. So, despite all her perceived smartness, she was left with a suboptimal choice.

Examples of Greedy Algorithms: Various Greedy algorithms will be discussed later are as follows:

- Minimum Spanning Tree
- Shortest Path.
- Activity Selection problem
- Fractional Knapsack problem
- Travelling Salesman Problem

d) Dynamic Programming:

Dynamic Programming tells us how to solve critical problems by breaking them down into a collection of simpler problems. To solve these problems, we, too, need to break down the complex problems into sub-problems and then solve them.

Analogy:- Duckworth Lewis method is based on a dynamic programming approach. It is used in cricket and when the game is interrupted due to weather, rain, or any other circumstances. To decide the winner in such cases, the resources available is made equally balanced, and the scenario is adjusted based upon the constraints available.

Remaining overs	Wicket Lost				
	0	2	5	7	9
50	100.0	85.1	49.0	22.0	4.7
40	89.3	77.2	47.6	22.0	4.7
30	75.1	67.3	44.7	21.8	4.7
20	56.6	52.4	38.6	21.2	4.7
10	32.1	30.8	26.1	17.9	4.7
5	17.2	16.8	15.4	12.5	4.6

Figure 1.13: Duckworth Lewis Method

Applications of dynamic programming:

- 0/1 knapsack problem
- Mathematical optimization problem
- All pair shortest path problem
- Reliability design problem
- Longest common subsequence (LCS)

e) Backtracking: Backtracking is an algorithmic technique that uses recursion to solve the sub-problems, and once, if any of the sub-problems fails to give the desired result, then the problem can roll back to that initial state by which it can reach another state of sub-problems which may lead to the final desired outcome.

Analogy:- We all have played sudoku games someday in our life as per the ***Figure 1.14.*** When we fill digit one by one and if the next digit to be filled does not lead the solution, then we remove it (backtrack it) and try the next digit. This is better than brute force as it does not test every possible combination, and it drops some choices whenever it backtracks.

	7			2			4	6
	6					8	9	
2			8			7	1	5
	8	4		9	7			
7	1						5	9
			1	3		4	8	
6	9	7			2			8
	5	8					6	
4	3			8			7	

Figure 1.14. Representing the Sudoko Game

f) Branch & Bound: A branch and bound algorithm is an optimization technique to get an optimal solution to the problem. It looks for the best solution for a given problem in the entire space of the solution. The bounds in the function to be optimized are merged with the value of the latest best solution. Some examples are :

- 8 puzzles Problem,
- Job Assignment Problem,
- Traveling Salesman Problem etc.

g) Randomization Algorithms: A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The Algorithm typically uses uniformly random bits as an auxiliary input to guide its behaviour in the hope of achieving good performance in the "average case" over all possible choices of random bits.

Conclusion:

In this unit, we have studied definition of algorithm, its applications, and characteristics of good algorithm. Also, studied about various mathematical parameters used to analyse the correctness of the algorithm. Analysis of algorithm is also done using various recurrences solving method. Various design approaches with their analogy is also discussed in brief in order to understand the flow of the syllabus. In coming units, we will discuss each of the design strategy one by one along with its complexity analysis.

Let us proceed towards various Companies Questions, GATE Questions developed from the Unit 1. Around 100 Objective type questions were framed on the topics of Unit 1.

4. Objective Type Questions

4.1. Company Based Objective Type Questions

1	**What will be the time and space complexity of following code?**
	`int x = 0, y = 0;`
	`for (int i = 0; i < n^2; i++) {`
	`  x = x + foo();`
	`}`
	`for (int j = 0; j < m ; j++) {`
	`  y= y + foo();`
	`}`

A	O(n^2 * m) , O (n^2 * m)
B	O(n^2 + m), O (n^2 + m)
C	O(n^2 + m), O (1)
D	O(n^2 * m) , O (m)
AN	**C**
DL	**M**

2	**What will be the time complexity of following code?**
	`int x = 0;`
	`for (int i = 0; i < N; i++) {`
	`  for (int j = N; j > i; j--) {`
	`  x = x + i *j;`
	`  }`
	`}`

A	O(n^2)
B	O(n)
C	O(n log n)
D	None of these
AN	**A**
DL	**M**

3	**What will be the time complexity of following code?**
	`int i, j, k = 0;`
	`for (i = n / 2; i >= n; i++) {`
	`  for (j = 2; j <= n; j = j * 2) {`
	`  k = k + n / 2;`
	`  }`
	`}`

A	O(n^2)
B	O(n)
C	O(n log n)
D	None of these
AN	**D**
DL	**M**

4	**What will be the time complexity of following code?**

	int i, j, k = 0;
	for (i = n / 2; i >= n; i++) {
	  for (j = 2; j <= n; j = j * 2) {
	  k = k + n / 2;
	  }
	}

A	O(n^2)
B	O(n)
C	O(n log n)
D	None of these
AN	**D**
DL	**M**

5	**What will be the time complexity of following code?**
	int i, sum;
	for (i = n ; i >= 1; i = i/2) {
	  sum = sum + n / 2;
	  }
	}

A	O(n^2)
B	O(logn)
C	O(n log n)
D	None of these
AN	**B**
DL	**M**

6	**What will be the complexity of the following code?**
	int a = 0, i = n;
	while (i > 0) {
	  a += i;
	  i /= 2;
	}

A	O(n log n)
B	O (n)
C	O (n^2)
D	O log(n)
AN	**D**
DL	**E**

7	**What will be the complexity of the following code?**
	int foo (n)
	{
	for (int i = n; i >= 1; i /= 2)
	for j = m to i
	a += (i * j);
	}

A	O (n + m)
B	O(n)
C	O(m logn)
D	O(n logm)
AN	**C**
DL	**M**

8	**What is the complexity of the following code?**
	int fun() {
	int a = 0;
	for (i = n; i >= 1; i /= 2){
	for (j = 1; j <= m; j *= 2){
	a += (i * j);
	}
	}
	}

A	O(n * m)
B	(log m log n)
C	(m log n)
D	(n log m)
AN	**B**
DL	**M**

9	**What is the complexity of the following code?**
	int fun(n) {
	int a = 0;
	while (n > 0) {
	a += n % 10;
	n /= 10;
	}
	}

A	O(log2 n)
B	O(log3 n)
C	O(log10 n)
D	O(n)
AN	**C**
DL	**M**

10	**What is the complexity of the following code?**
	int foo (n , m)
	{
	for (int i = n; i >= 1; i /= 2)
	for (int j = 1; j <= m; j *= 2)
	ans += (i * j);
	}

A	O (n logm)
B	O (m logn)
C	O (logm logn)
D	O(m + n)
AN	**C**
DL	**M**

11	**What is the complexity of the following code?**
	int c = 0;
	for (int i = 0; i < N; i++)
	for (int j = 0; j < i; j++)
	c++;

A	O(1)
B	O(n)
C	O(n^2)
D	O(n log n)
AN	**C**
DL	**E**

12	**What is the complexity of following code?**
	int foo (n)
	{
	for(int i = 1 ; i < n , i++)
	for (int j = i; j <= n; j += i)
	ans += (i * j);
	}

A	O(n)
B	O(n^2)
C	O(n logn)
D	None of these
AN	**C**
DL	**M**

13	**What is the complexity of following code?**

	int foo(int n)
	{
	  int sum = 0;
	  for (int i = n; i > 0; i /= 2)
	  for (int j = 0; j < i; j++)
	  sum += 1;
	  return sum;
	}

A	O (n^2)
B	O (nlog n)
C	O (n)
D	None of these
AN	**C**
DL	**D**

14	**What will be the complexity for following code?**
	int fun2(int n)
	{
	  if (n <= 1) return n;
	  return fun2(n-1) + fun2(n-1);
	}

A	O(n^2)
B	O(n^2 logn)
C	O(n^3 logn)
D	None of these
AN	**D**
DL	**M**

15	**What will be the number of the recursive calls made by the following code?**
	
	int foo(a , b)
	{
	  if (a % b ==0) return b;
	  a = a % b;
	  return gcd(a , b);
	}
	Assuming a >= b.

A	O (n)
B	O(nlogn)
C	O (log (n))
D	O(c) where c is a constant
AN	**C**
DL	**M**

16	**What is the time complexity of the following code?**

	int fun(n) {
	  if (n == 0 or n == 1)
	  return 1;
	  else
	  return f(n - 1) + f(n - 2);
	}

A	O(log n)
B	O(n)
C	O(n^2)
D	O(2^n)
AN	**D**
DL	**E**

17	**Form a recurrence relation for the following code-**
	
	foo(x)
	
	{
	  if(x<=1)
	  return 1;
	  else
	  return foo(√x);
	}

A	T(n) = T(n) + 1
B	T(n) = T(√n) + 1
C	T(n) = T(n) – 1
D	None of these
AN	**B**
DL	**D**

18	**Solve the above formed recurrence relation-**
	
	foo(x)
	
	{
	  if(x<=1)
	  return 1;
	  else
	  return foo(√x);
	}

A	O (n^2)
B	O (nlogn)
C	O (loglogn)
D	None of these
AN	**C**
DL	**D**

19	**What is the time complexity of the following code?**

	int foo(n)
	{
	if(n >= 1)
	{
	foo(n - 1);
	printf ("hii");
	}
	}

A	O(c) where c is constant
B	O (1)
C	O (n)
D	O (n^2)
AN	**C**
DL	**E**

20	**How many times the foo (6) will be called in the following code ?**
	int foo(n)
	{
	if(n >= 1)
	{
	foo(n - 1);
	printf ("hii");
	}
	}

A	5
B	6
C	7
D	8
AN	**C**
DL	**M**

21	**How many times “hii “will be printed when foo (6) will be called in the following code ?**
	int foo(n)
	{
	if(n >= 1)
	{
	foo(n - 1);
	printf ("hii");
	}
	}

A	6
B	5
C	7
D	4
AN	**A**
DL	**M**

22	**What is the space complexity of the following code?**

	# include <stdio.h>
	int foo(n)
	{
	  if(n >= 1)
	  {
	  foo(n - 1);
	  printf ("hii");
	  foo(n - 1);
	  }
	}

A	O (k) where k is constant
B	O (1)
C	O (n)
D	O (n^2)
AN	**C**
DL	**M**

23	**How many times function foo will be called when n = k in the following code?**
	# include <stdio.h>
	int foo(n)
	{
	  if(n >= 1)
	  {
	  foo(n - 1);
	  printf ("hii");
	  foo(n - 1);
	  }
	}

A	2^k
B	2^k – 1
C	2*k + 1
D	2^(k+1) - 1
AN	**D**
DL	**M**

24	**What is the time complexity of the following code?**
	# include <stdio.h>
	int foo(n)
	{
	  if(n >= 1)
	  {
	  foo(n - 1);
	  printf ("hii");
	  foo(n - 1);
	  }
	}

A	O(n ^2)
B	O (n ^n)
C	O (2^n)
D	O(1)
AN	**C**
DL	**M**

25	**What will be the value of foo (4) in the following code?**
	# include <stdio.h>
	int foo(n)
	{
	  if(n >= 1)
	  {
	  foo(n - 1);
	  printf ("hii");
	  foo(n - 1);
	  }

A	1232
B	12321
C	12321423
D	12321412321
AN	**D**
DL	**M**

4.2. GATE/PSUs Objective Type Questions:

1	**What will be the time complexity of following code?** **GATE 2014**
	
	int fun(int n)
	{
	int count = 0;
	for(int i=n; i>0; i/=2)
	for(int j=0 ; j< i; j++)
	count ++
	return count;
	}

A	O(n^2)
B	O(logn)
C	O(n log n)
D	None of these
AN	**D**
DL	**M**

2	**What will be the time complexity of following code?** **GATE 2013**
	int unknown(int n) {
	int i, j, k = 0;
	for (i = n/2; i <= n; i++)
	for (j = 2; j <= n; j = j * 2)
	k = k + n/2;
	return k;
	}

A	O(n^2)
B	O(n^2 logn)
C	O(n^3 logn)
D	O(n^3)
AN	**B**
DL	**M**

3	**An algorithm is made up of two modules M1M1 and M2.M2. If order of M1 is f(n) and M2 is g(n) then the order of algorithm is** **NIELIT 2017**
A	max(f(n),g(n))
B	min(f(n),g(n))
C	f(n)+g(n)
D	f(n)×g(n)
AN	**A**
DL	**D**

4	**There are *n* unsorted arrays: A_1, A_2, …, A_n. Assume that *n* is odd. Each of A_1, A_2, …, A_n contains *n* distinct elements. There are no common elements between any two arrays. The worst-case time complexity of computing the median of the medians of A_1, A_2, …, A_n is** **GATE 2019**
A	O(n)
B	O(n logn)
C	$O(n^2)$
D	$O(n^2 \log n)$
AN	**C**
DL	**D, please verify**

5	Consider the following function from positive integers to real numbers: 10, √n, n, log2n, 100/n The CORRECT arrangement of the above functions in increasing order of asymptotic complexity is: a) log2n, 100/n, 10, √n, n b) 100/n, 10, log2n, √n, n c) 10,100/n, √n, log2n, n d) 100/n, log2n, 10, √n, n **GATE 2017**
A	A
B	B
C	C
D	D
AN	**B**
DL	**D, please verify**

6	Match the algorithms with their time complexities: **Algorithm** / **Time complexity** (P) Towers of Hanoi with *n* disks — (i)Θ(n2)Θ(n2) (Q) Binary search given *n* stored numbers — (ii)Θ(nlogn)Θ(nlogn) (R) Heap sort given *n* numbers at the worst case — (iii)Θ(2n)Θ(2n) (S) Addition of two *n*××*n* matrices — (iv)Θ(logn)Θ(logn) (A) p→(iii),Q→(iv),R→(i),S→(ii) (B) p→(iv),Q→(iii),R→(i),S→(ii) (C) p→(iii),Q→(iv),R→(ii),S→(i) (D) p→(iv),Q→(iii),R→(ii),S→(i) **GATE 2017**
A	A

B	B
C	C
D	D
AN	**C**
DL	**D, please verify**

7	Consider the following C function
	int. fun(int.n)
	{
	int i, j;
	for(i = 1; i<= n; i++)
	{
	for(j=1; j<n; j += i)
	{
	printf(" %d %d",i,j);
	}
	}
	}
	Time complexity of fun in terms of θ notation is **GATE 2017**

A	Θ(n√n)
B	$\Theta(n^2)$
C	Θ(nlogn)
D	$\Theta(n^2 \log n)$
AN	**C**
DL	**D**

8	The worst case running times of *Insertion sort, Merge sort and Quick sort*, respectively, are: a. Θ(nlogn),Θ(nlogn),andΘ(n2) b. Θ(n2),Θ(n2),andΘ(nlogn) c. Θ(n2),Θ(nlogn),andΘ(nlogn) d. Θ(n2),Θ(nlogn),andΘ(n2) **GATE 2016**
A	A
B	B
C	C
D	D
AN	**D**
DL	**D**

9	Consider the following C program segment.
	while(first <= last)
	{
	if(array[middle] < search)
	_____; line 1
	else if (array[middle] = = search)
	found = TRUE;
	else _____; line 2
	middle = (first + last)/2;
	}
	if (first > last) not Present = TRUE
	What will come at Line 1 and Line 2. **GATE 2015**

A	last = middle – 1, first = middle + 1
B	first = middle + 1, last = middle – 1
C	first = middle + 1, first = middle + 1
D	last = middle – 1; last = middle – 1;
AN	**B**
DL	**D**

10	An algorithm performs $(\log N)^{1/2}$ find operations, N insert operations, $(\log N)^{1/2}$ delete operations, and $(\log N)^{1/2}$ decrease-key operations on a set of data items with keys drawn from a linearly ordered set. For a delete operation, a pointer is provided to the record that must be deleted. For the decrease-key operation, a pointer is provided to the record that has its key decreased. Which one of the following data structures is the most suited for the algorithm to use, if the goal is to achieve the best asymptotic complexity considering all the operations? **GATE 2015**
A	Unsorted array
B	Min–heap
C	Sorted array
D	Sorted doubly linked list.
AN	**B**
DL	**D**

11	Consider the following C function. **GATE 2015**
	int fun1 (int n){
	int i, j, k, p, q = 0;
	for (i = 1; i < n; ++i) {
	p = 0;
	for(j = n; j > 1; j = j/2)
	++p;
	for(k =1; k < p; k = k*2)
	++q;
	}
	return q;
	}
	Which one of the following most closely approximates the return value of the function fun1?

A	n^3

B	$n (\log n)^2$
C	$n \log n$
D	$n \log(\log n)$
AN	**D**
DL	**D**

12	An unordered list contains nn distinct elements. The number of comparisons to find an element in this list that is neither maximum nor minimum is **GATE 2015**
A	Θ(nlogn)
B	Θ(n)
C	Θ(logn)
D	Θ(1)
AN	**D**
DL	**D**

13	Let $f(n) = n$ and $g(n) = n^{(1+\sin n)}$, where n is a positive integer. Which of the following statements is/are correct? **GATE 2015** I. $f(n) = O(g(n))$ II. $f(n) = \Omega(g(n))$
A	only I
B	only II
C	Both I and II
D	Neither I and II
AN	**D**
DL	**D**

14	Consider the following function: **GATE 2013**
	int unknown(int n){
	int i, j, k=0;
	for (i=n/2; i<=n; i++)
	for (j=2; j<=n; j=j*2)
	k = k + n/2;
	return (k);
	}
	The return value of the function is

A	Θ(n2)
B	Θ(n2log n)
C	Θ(n3)
D	Θ(n3log n)
AN	**B**
DL	**D**

15	Which of the given options provides the increasing order of asymptotic complexity of functions: $f1(n)=2^n$ $f2(n)=n^{3/2}$ $f3(n)=n\log_2 n$ $f4(n)=n^{\log_2 n}$ **GATE 2013**
A	f3,f2,f4,f1
B	f3,f2,f1,f4
C	f2,f3,f1,f4
D	f2,f3,f4,f1
AN	**B**
DL	**D**

4.3 Unsolved Questions:

1	**What will be the time complexity of following code**?
	#include <stdio.h>
	int main(void)
	{
	int sum=0,i,j;
	for(i=n;j=0,i>0;i=i/2,j=j+i)
	{
	sum=sum+i;
	}
	return 0;
	}

2	**What will be the time complexity of following code?**
	#include <stdio.h>
	int main(void)
	{
	int sum=0, i, j;
	for(i=1;i<=n;i++)
	{
	for(j=1;j<=n;j++)
	{
	sum=sum+j;
	}
	}
	return 0;
	}

3	**What will be the time complexity of following code?**
	#include <stdio.h>
	int main(void)
	{
	int sum=0,i,j;
	for(i=1;i<=n;i++)
	{
	for(j=1;j<=n;j=j+2)
	{
	sum=sum+j;
	}
	}
	return 0;
	}

4	What will be the time complexity of following code?
	
	#include <stdio.h>
	int main(void)
	{
	int sum=0,i,j;
	for(i=1;i<=n;i++)
	{
	for(j=1;j<=n;j=j*3)
	{
	sum=sum+j;
	}
	}
	return 0;
	}

5	**What will be the time complexity of following code?**
	#include <stdio.h>
	int main(void)
	{
	int sum=0,i,j;
	for(i=1;i<=n;i=i*2)
	{
	for(j=1;j<=n;j=j+2)
	{
	sum=sum+j;
	}
	}
	return 0;
	}

6	**What will be the time complexity of following code?**
	#include <stdio.h>
	int main(void)
	{
	int sum=0, i, j;
	for(i=1;i<=n;i=i+1)
	{
	for(j=1;j<=i;j=j+2)
	{
	for(k=1;k<=n;k=k*2)
	{
	sum=sum+k;
	}
	}
	}
	return 0;
	}

7	**What will be the time complexity of following code?**
	#include <stdio.h>
	int main(void)
	{
	int sum=0,i,j;
	for(i=0;i<=n;i=i+1)
	{
	for(j=1;j<=i;j=j+1)
	{
	if(j%i==0)
	{
	for(k=1;k<=n;k=k+1)
	{
	sum=sum+k;
	}
	}
	}
	}
	return 0;
	}

8	**What will be the time complexity of following code?**
	int fact(int n)
	{
	if(n==0 || n==1)
	return 1;
	else
	return n*fact(n-1);
	}

9	**What will be the time complexity of following code**?
	
	iint fib(int n)
	{
	if(n==0)
	return 0;
	if(n==1)
	return 1;
	else
	return fib(n-1) +fib(n-2);
	}

10	**What is the time, space complexity of following code:**
	
	int a = 0, b = 0;
	for (i = 0; i < N; i++)
	{
	a = a + rand();
	}
	for (j = 0; j < M; j++)
	{
	b = b + rand();
	}

11	**What is the time complexity of following code: int a = 0, i = N;**
	
	while (i > 0)
	{
	a += i;
	i /= 2;
	}

12	**What is the time, space complexity of following code:**
	int a = 0, b = 0;
	for (i = 0; i < N; i++)
	{
	a = a + rand();
	}
	for (j = 0; j < M; j++)
	{
	b = b + rand();
	}

13	**What is the time complexity of following code:**
	int a = 0;
	for (i = 0; i < N; i++)
	{
	for (j = N; j > i; j--)
	{
	a = a + i + j;
	}
	}

14	**What is the time complexity of following code:**
	int i, j, k = 0;
	for (i = n / 2; i <= n; i++)
	{
	for (j = 2; j <= n; j = j * 2)
	{
	k = k + n / 2;
	}
	}

15	**What is the time complexity of following code:**
	def f():
	a = 0
	for i = 1 to n:
	a += i;
	b = 0
	for i = 1 to m:
	b += i;
	Time Complexity of this program:

16	**Find the sum of digits of a number in its decimal representation.**
	def f(n):
	ans = 0
	while (n > 0):
	ans += n % 10
	n /= 10;
	print(ans)
	Time Complexity of this program:

17	**What is the time complexity of following code:**
	int a = 0, i = N;
	while (i > 0)
	{
	a += i;
	i /= 2;
	}

2.1 Definition of Recursion:

When a function calls itself, it is called recursion.

A recursive problem is generally divided into smaller parts and then work on those smaller problems to get the result of the original problem.

It is very important to keep in mind that the recursive solution must terminate or must have some base condition to end the recursive call.

2.2 Why Recursion:

It is very general question that if we can solve any problem using the iteration method, then why there is need for the recursive solution for the same problem.

The solution of this question is that recursive solutions are generally small and easy to understand as compare to iterative solution.

Recursive approach is very helpful in those conditions where solution of small part of original problem helps to solve the original problem in a faster way.

For example, suppose we want to calculate the factorial of 10:

In this case, 10! = 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1

But if we look more closely in the above solution, then we can find that

10! = 10 * 9! OR

10! = 10 * 9 * 8! OR

10! = 10 * 9 * 8 * 7! And so on.
Here, it can be understood that if we got the solution of smaller problem, then it is very easy and fast to calculate the solution of the original problem.

2.3 Format of a Recursive function:

Basically, a recursive function calculates the solution by performing the task in smaller part by calling itself to perform the subparts.

At some point of time, there is a sub part for which recursive call is not required. This is called base case or base condition for the recursive solution.

It is very important to keep in mind that the recursive solution must terminate or must have some base condition to end the recursive call.

We can write recursive functions using the below format:

If (test for base condition)
    Return some base condition value;
Else if (test for another base condition)
    Return some another base condition value;
// the recursive case
Else
    Return (some work and then a recursive call)

Let's write down the solution of above factorial problem in the recursive way:

n! = 1; if n = 0
n! = n * (n – 1)!; if n > 0

Here, the original problem is to find the factorial of n and
Sub-problem is to find the factorial of (n - 1) and multiplies the result by n.

In the base condition, if n is 0 or 1, then the function simple returns 1.

That's how we can write down the solution of the problems in a recursive manner.

2.4 Approaches to solve the recurrence:

There are four popular approaches to solve the recurrence:

- Substitution Method
- Iteration Method
- Recursion Tree Method
- Master Method

2.5 Substitution Method:

This method for solving the recurrence problem using the Principle of Mathematical Induction. It involves guessing the form of the solution and then using mathematical induction to find the Constants and show that the solution works. It is a good technique to solve the recurrence, but This method can be used only where it is easy to guess the solution of the problem.
Let's take some example to understand how we can solve the problems using this method:

Example 1: Consider the recurrence

T(n) = T(n / 2) + 1 and we have to show that it is asymptotically bound by O(log n).

Solution:

For the given problem, T(n) = O(log n)

Now, we have to show that for some constant c, i.e. T(n) <= clogn

Put this in the given recurrence equation:

T(n) <= c log(n / 2) + 1

<= c logn – clog2 + 1

<= c logn for c>=1

As it has proved that the solution of the given recurrence problem is asymptotically bound by O(log n).

Thus, T(n) = O(log n)

Example 2: Consider the recurrence

T(n) = 2T((n / 2) + 16) + n and we have to show that it is asymptotically bound by O(n log n).

Solution: For the given problem, T(n) = O(n log n)

Now, we have to show that for some constant c, i.e. T(n) <= cn logn

Put this in the given recurrence equation:

T(n) <= 2 [c((n / 2) + 16) log((n / 2) + 16)] + n

<= cn log(n / 2) + 32 + n

<= cn logn – cn log2 + 32 + n

<= cn logn – cn + 32 + n

<= cn logn – (c – 1)n + 32

<= cn logn (for c >= 1)

As it has proved that the solution of the given recurrence problem is asymptotically bound by O(n log n).

Thus, T(n) = O(n log n)

2.6 Iteration Method:

This method uses the technique to iterate or expand the recurrence and express the problem as a summation of given term n and base case or base condition.

Let's take some example to understand how we can solve the problems using this method:

Example 1:

T(n) = 1 + T(n – 1)

= 1 ; if n = 1

Solution:

As per the question, let's consider the below equation as i)

T(n) = 1 + T(n – 1)--------(i)

If we put (n – 1) instead of n in equation (i), then we got:

T(n - 1) = 1 + T(n – 2)--------(ii)

Let's put (n – 2) instead of n in equation (i), then we got:

T(n - 2) = 1 + T(n – 3)--------(iii)

Now, let's substitute equation (ii) into (i)

T(n) = 1 + 1 + T(n – 2)

= 2 + T(n – 2)-------- (iv)

Now, again substitute equation (iii) in equation (iv), then

T(n) = 2 + 1 + T(n – 3)

= 3 + T(n – 3)

…….

…….

…….

…….

Now, we got a pattern that

T(n) = k + T(n – k) --------- (v)

From the given question, we know that T(1) = 1

So, if we have to make T(n – k) as T(1), then equate as:

n – k = 1 => k = n-1 -------(vi)

Now, let's substitute value of k from (vi) in (v), then

T(n) = (n – 1) + T(n – (n – 1))

= (n – 1) + T(1)

= (n – 1) + 1

= n

Therefore, the solution for the given recursive problem is O(n).

Example 2:

T(n) = n + T(n – 1); for n > 1

= 1 ; for n = 1

Solution:

As per the question, let's consider the below equation as i)

$$T(n) = n + T(n - 1) \text{--------} (i)$$

If we put $(n - 1)$ instead of n in equation (i), then we got:

$$T(n - 1) = (n - 1) + T(n - 2) \text{--------} (ii)$$

Let's put $(n - 2)$ instead of n in equation (i), then we got:

$$T(n - 2) = (n - 2) + T(n - 3) \text{-------} (iii)$$

Now, let's substitute equation (ii) into (i)

$$T(n) = n + (n - 1) + T(n - 2) \text{------} (iv)$$

Now, again substitute equation (iii) in equation (iv), then

$$T(n) = n + (n - 1) + (n - 2) + T(n - 3)$$

…….
…….
…….
…….
…….

Now, we got a pattern that

$$T(n) = n + (n - 1) + (n - 2) + \text{……}(n - k) + T(n - (k + 1)) \text{----------} (v)$$

From the given question, we know that $T(1) = 1$

So, if we have to make $T(n - (k + 1))$ as $T(1)$, then equate as:

$$n - (k + 1) = 1 \Rightarrow k = n - 2 \text{-------} (vi)$$

Now, let's substitute value of k from (vi) in (v), then

$$T(n) = n + (n - 1) + (n - 2) + \text{.....} (n - (n - 2)) + T(n - (n - 2 + 1))$$

$$= n + (n - 1) + (n - 2) + \text{.....} 2 + T(1)$$

$$= n + (n - 1) + (n - 2) + \text{.....} 2 + 1$$

It is nothing but sum of first n natural numbers, which is equivalent to:

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

= O (n ^ 2)

Therefore, the solution for the given recursive problem is O(n ^ 2).

Example 3:

T(n) = T(n / 2) + c ; for n > 1

= 1 ; for n = 1

Solution:

As per the question, let's consider the below equation as i)

T(n) = T(n / 2) + c ------- (i)

If we put (n / 2) instead of n in equation (i), then we got:

T(n / 2) = T (n / 4) + c ------- (ii)

Let's put (n / 4) instead of n in equation (i), then we got:

T(n / 4) = (n / 8) + c--------(iii)

Now, let's substitute equation (ii) into (i)

T(n) = T (n / 4) + c + c => T(n / 2^2) + 2c ------ (iv)

Now, again substitute equation (iii) in equation (iv), then

T(n) = T (n / 8) + c + 2c => T(n / 2^3) + 3c

…….
…….
…….
…….
…….

Now, we got a pattern that

T(n) = T (n / 2^k) + kc ---------- (v)

From the given question, we know that T(1) = 1

So, if we have to make T(n / 2^k) as T(1), then equate as:

$2^k = n$ ----------(vi)

Now, let's substitute value of k from (vi) in (v), then

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

$= T(1) + kc$

$= 1 + kc$ --------(vii)

But, we have compute the solution in terms of n, so let's take log on both sides of equation (vi)

$\log n = \log 2^k$

$\log n = k \log 2$

$\log n = k$ (since $\log 2 = 1$)

Now, let's put this value of k in equation (vii)

$T(n) = 1 + (\log n). c$

$= (\log n \text{ base } 2)$

Therefore, the solution for the given recursive problem is O(log n base 2).

Example 4:

$T(n) = 2\,T(n - 1)$, if $n > 1$

$= 1$, if $n = 1$

Solution:

As per the question, let's consider the below equation as i)

$T(n) = 2\,T(n - 1)$ --------(i)

If we put (n - 1) instead of n in equation (i), then we got:

$T(n - 1) = 2\,T(n - 2)$ --------(ii)

Let's put (n - 2) instead of n in equation (i), then we got:

$T(n - 2) = 2\ T(n – 3)$ ------- (iii)

Now, let's put equation (ii) into (i)

$T(n) = 2\ [2\ T(n – 2)\ ]$

$= 2^2\ T(n – 2)$ --------- (iv)

Now, let/s put equation (iii) into (iv)

$T(n) = 4\ [2\ T(n – 3)\ ]$

$= 2^3\ T(n – 3)$ ---------- (v)

So, from equation (iv) and (v), we got a pattern that if we repeat the above iteration till k times,

then we got as:

$T(n) = 2^k\ T(n – k)$ --------- (vi)

Now, if we have to use the base condition, then we have to equate:

$n – k = 1 \Rightarrow k = n – 1$ ----------(vii)

Let's put this value of k in equation (vi), then

$T(n) = 2^{(n – 1)}\ T(n – (n – 1))$

$= 2^{(n – 1)}\ T(1)$

$= 2^{(n – 1)}$

Therefore, the solution for the given recursive problem is $2^{(n – 1)}$.

2.7 Drawbacks of Substitution and Iteration Method:

Changing Variable Method:

Sometimes, there are some recurrence problems for which substitution and iterative method does not help to find the answer easily. In that case, we have to change the variable to make

the recursive problem easy so that it can be efficiently solved. Sometimes, a little algebraic manipulation can make an unknown recurrence similar to one we have seen before.

Let's see these types of problems and check how can we solve such recurrences:

Example 1: Consider the recurrence:

T(n) = 2T(root n) + log n. Solve the recurrence by changing variable.

Solution:

We have, T(n) = 2T(root n) + log n -----(i)

Suppose m = log n => n = 2^m ---- (ii)

Therefore, n^(1/2) = 2^(m/2) => root n = 2^(m/2) ------ (iii)

Now, let's put these values in the given recurrence:

T(2^m) = 2T(2^(m/2)) + m

Again, lets consider S(m) = T(2^m)

Therefore, S(m) = 2 S(m/2) + m ----- (iv)

Now, by using the master theorem, we know the solution of equation (iv)

S(m) = O(m logm)

Now, substitute the values of m in terms of n, we get

T(n) = S(m) = O(m logm)

= O(log n log log n)

Hence, the solution of the given recurrence is O(log n log log n).

Example 2: Consider the recurrence:
T(n) = 2T(root n) + 1. Solve the recurrence by changing variable.

Solution:

We have, T(n) = 2T(root n) + 1 -----(i)

Suppose m = log n => n = 2^m ---- (ii)

Therefore, n^(1/2) = 2^(m/2) => root n = 2^(m/2) ------ (iii)

Now, let's put these values in the given recurrence:

T(2^m) = 2T(2^(m/2)) + 1

Again, lets consider S(m) = T(2^m)

Therefore, S(m) = 2 S(m/2) + 1 -----(iv)

Now, by using the master theorem, we know the solution of equation (iv)

S(m) = O(logm)

Now, substitute the values of m in terms of n, we get

T(n) = S(m) = O(logm)

= O(log log n)

Hence, the solution of the given recurrence is O(log log n).

NOTE: From the above examples, we have seen that how easily we have solved the recurrence which seems to be too difficult in first go as these recurrences contains the terms in root.

Master Method:

2.8 Why Master's Method:

There are various methods used for solving recurrences. Every method has some advantages and disadvantages associated with it. One of the most important methods used for solving the recurrences is the Master method. Master Method provides the running time complexity for the recurrences of the type that is abiding by the divide and conquer approach.

Axiom of Master Method:

Consider a problem that can be solved using a recursive algorithm such as the following:

Procedure T (n: the size of the problem) **defined as**:

if n < 1 then exit

Do work of amount f (n)

T(n/b)

T(n/b)

…….repeat for a total of times...

T (n/b)

End procedure

Solution: In the above function, given a problem of size n and we need to divide the problem into two sub-problem of the size of n/b. This function can further be interpreted as a recursive call to the tree, with each node in the tree as an instance of one of the recursive calls and its child nodes to the other subsequent calls. In this, a function is divided into the two sub-function of size T (n/b). The cost of dividing the functions into sub-functions will be computed further. F (n) represents the cost that occurs in dividing the given problem into sub-problem and then merging it further.

Recurrence for the above mentioned algorithm will be represented as T (n) = a T (n/b) + f (n).

2.8.1 Definition of Master Method:

Master Method deals with the recurrences of the following type of form:

T (n) = a T (n/b) + f (n) where a ≥ 1 and b>1

- n is the problem size.
- a is the number of sub-problems.
- n/b is the sub-problem size.
- f (n) is the amount of the word done in recursive calling, which includes the cost of dividing the problem and the cost of merging the solutions to the sub-problems

2.8.2 Three Cases of Master Method:

Case 1: If f (n) = O ($n^{\log_b a - \varepsilon}$) for some constant ε > 0, then T (n) = Θ ($n^{\log_b a}$).
Case 2: If f (n) = Θ ($n^{\log_b a}$), then T (n) = Θ ($n^{\log_b a}$ lg n).
Case 3: If f (n) = Ω ($n^{\log_b a + \varepsilon}$) with ε > 0, and f(n) satisfies the regularity condition, then T (n) = Θ(f(n)). Regularity condition: af(n/b) ≤ cf(n) for some constant c < 1 and all sufficiently large n.

2.8.3 Gaps Identified in Master's Method:

In each of the three cases, we compare the function f (n) with the function $n^{\log_b a}$. Intuitively, the larger of the two functions determine the solution to the recurrence.

2.8.3.1 In case 1, the function $n^{\log_b a}$ is larger, and the solution is T(n)= Θ ($n^{\log_b a}$). In case 3 function f(n) is larger, solution is T(n)= Θ (f (n)). In case 2, both functions have the same value. The solution is T (n)= Θ($n^{\log_b a}$ log n).

2.8.3.2 In the first case, not only must f(n) be smaller than $n^{\log_b a}$, it must be polynomially smaller.

2.8.3.3 In the third case, not only must f (n) be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that a T (n/b) ≤c f(n).

2.8.3.4 In addition to that, all three cases do not cover all possibilities. Some function might be lies in between case 1 and 2, and some other lies in-between case 2 and 3 because the comparison is not polynomial larger or smaller. In case 3, the regularity condition fails.

2.8.4 Examples on Master Method:

Example 1: T (n) = 3T (n/2) + n^2
T (n) = 3T (n/2) + n^2 ⇒ T (n) = Θ (n^2) (Case 3)
Example 2: T (n) = 4T (n/2) + n^2
T (n) = 4T (n/2) + n^2 ⇒ T (n) = Θ (n^2 log n) (Case 2)
Example 3: T (n) = T (n/2) + 2^n
T (n) = T (n/2) + 2^n ⇒ T (n) = Θ (2^n) (Case 3)
Example 4: T (n) = 2^nT (n/2) + n^n
T (n) = 2nT (n/2) + n^n ⇒ Does not apply (a is not constant)
Example 5: T (n) = 16T (n/4) + n
T (n) = 16T (n/4) + n ⇒ T (n) = Θ(n^2) (Case 1)
Example 6: T (n) = 2T (n/2) + n log n
T (n) = 2T (n/2) + n log n ⇒ T (n) = n $\log_2$ n (Case 2)

Example 7: T (n) = 2T (n/2) + n/ log n

T (n) = 2T (n/2) + n/ log n ⇒ Does not apply (non-polynomial difference bet f(n) and n log $_b$ a)

2.8.5 Modified Master Method:

Master Method provides the running time complexity for the recurrences of the type that is abiding by the divide and conquer approach.

So, according to the master theorem, the runtime of the above Algorithm can be expressed as:

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

- n is the problem size.
- a is the number of sub-problems.
- n/b is the sub-problem size.
- f (n) is the amount of the word done in recursive calling, which includes dividing the problem and the cost of merging the solutions to the sub-problems.

Not all recurrence relations can be solved with the use of the master theorem, i.e., if

- T(n) is not monotone, ex: T(n) = sin n
- f(n) is not a polynomial, ex: T(n) = 2T(n/2) + 2^n

This theorem is an advanced version of the master theorem that can be used to determine the running time of divide and conquer algorithms if the recurrence is of the following form:-

Where n = size of the problem

a = number of sub-problems in the recursion and a >= 1, n/b = size of each sub-problem

b > 1, k >= 0 and p is a real number.

Case 1: If a > b ^{k}, then T (n) = θ (n $^{\log_b a}$)

Case 2: if a = b ^{k}, then

(a) If p > -1, then T (n) = θ (n $^{\log_b a}$ log^{p+1}n)

(b) if p = -1, then T (n) = θ (n $^{\log_b a}$ log log n)

(c) if p < -1, then T (n) = θ (n $^{\log_b a}$)

Case 3: if a < b ^{k}, then

(a) if p >= 0, then T (n) = θ (n ^{k} log ^{p} n)

(b) if p < 0, then T (n) = θ (n ^{k})

2.8.6 Practice Questions on Modified Master Method:

Problem-01: Solve the following recurrence relation using Master's theorem:

T (n) = 3 T (n/2) + n^2

Solution: We compare the given recurrence relation with T (n) = a T (n/b) + θ (n ^{K} log ^{P} n).

Then, we have: a = 3, b = 2, k = 2, p = 0

Now, a = 3 and b $_K$ = 22 = 4.

Clearly, $a < b_K$

So, we follow case-03.

Since p = 0, so we have-

$T(n) = \theta(n^K \log^P n)$.

$T(n) = \theta(n^2 \log^0 n)$

Thus, $T(n) = \theta(n^2)$

Problem-02: Solve the following recurrence relation using Master's theorem:

$T(n) = 2\,T(n/2) + n \log n$

Solution: We compare the given recurrence relation with $T(n) = a\,T(n/b) + \theta(n^K \log^P n)$.

Then, we have-

a = 2, b = 2, k = 1, p = 1

Now, a = 2 and $b_K = 21 = 2$.

Clearly, $a = b_k$. So, we follow case-02. Since p = 1, so we have-

$T(n) = \theta(n^{\log_b a} . \log^{p+1} n)$

$T(n) = \theta(n \log_2 2 . \log^{1+1} n)$

Thus,

$T(n) = \theta(n \log^2 n)$

Problem-03: Solve the following recurrence relation using Master's theorem:

$T(n) = 2\,T(n/4) + n^{0.51}$

Solution: We compare the given recurrence relation with $T(n) = a\,T(n/b) + \theta(n^K \log^P n)$.

Then, we have: a = 2, b = 4, k = 0.51, p = 0

Now, a = 2 and $b_k = 40.51 = 2.0279$.

$a < b_k$.

So, we follow case-03. Since p = 0, so we have-

$T(n) = \theta(n^K \log^P n)$

$T(n) = \theta(n^{0.51} \log^0 n)$

Thus, $T(n) = \theta(n^{0.51})$

Problem-04: Solve the following recurrence relation using Master's theorem:

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

Solution: We compare the given recurrence relation with $T(n) = a\,T(n/b) + \theta(n^K \log^P n)$.

Then, we have: $a = \sqrt{2}$, b = 2, k = 0, p = 1

Now, $a = \sqrt{2} = 1.414$ and $b_k = 20 = 1$.

$a > b_k$

So, we follow case-01.

So, we have-

$T(n) = \theta (n \log^{b} a)$

$T(n) = \theta (n\log^{2}\sqrt{2})$

$T(n) = \theta (n^{½})$

Thus, $T(n) = \theta (\sqrt{n})$

Problem-05: Solve the following recurrence relation using Master's theorem:

$T(n) = 8\,T(n/4) - n^{2}\log n$

Solution: The given recurrence relation does not correspond to the general form of Master's theorem. So, it cannot be solved using the Master's theorem.

Problem-06: Solve the following recurrence relation using Master's theorem:

$T(n) = 3\,T(n/3) + n/2$

Solution: We write the given recurrence relation as $T(n) = 3\,T(n/3) + n$.

This is because, in the general form, we have θ for function $f(n)$ which hides constants.

Now, we can easily apply the Master's theorem. We compare the given recurrence relation with $T(n) = a\,T(n/b) + \theta (n^{K} \log^{P} n)$.

Then, we have: $a = 3$, $b = 3$, $k = 1$, $p = 0$

Now, $a = 3$ and $b_{k} = 31 = 3$.

$a = b_{k}$. So, we follow case-02.

Since $p = 0$, so we have-

$T(n) = \theta (n\log^{b}a.\log^{p+1} n)$

$T(n) = \theta (n\log^{3}3.\log^{0+1} n)$

$T(n) = \theta (n^{1}.\log^{1}n)$

Thus, $T(n) = \theta (n \log n)$

Problem-07: Form a recurrence relation for the following code and solve it using Master's theorem-

```
A (n)
{
IF (n<=1)
Return 1;
  Else
   Return A  (√n);
}
```

Solution: We write a recurrence relation for the given code as $T(n) = T(\sqrt{n}) + 1$.

Here 1 is a Constant time taken for comparing and returning the value. We cannot directly apply the Master's Theorem to this recurrence relation. This is because it does not correspond

to the general form of the Master's theorem. However, we can modify and bring it in the general form to apply the Master's theorem.

Let, $n = 2^m$ ..... (1)

Then, $T(2^m) = T(2^m/2) + 1$

Now, let $T(2^m) = S(m)$, then $T(2^m/2) = S(m/2)$

So, we have, $S(m) = S(m/2) + 1$

Now, we can easily apply Master's Theorem.

We compare the given recurrence relation with $S(m) = a\,S(m/b) + \theta(m^k \log^p m)$.

Then, we have, $a = 1$, $b = 2$, $k = 0$, $p = 0$,

Now, $a = 1$ and $b_k = 20 = 1$.

$a = b_k$

So, we follow case-02.

Since $p = 0$, so we have-

$S(m) = \theta(m\log^b a.\log^{p+1} m)$

$S(m) = \theta(m\log^2 1.\log^{0+1} m)$

$S(m) = \theta(m^0.\log^1 m)$

Thus, $S(m) = \theta(\log m)$ ..... (2)

Now, from (1), we have $n = 2^m$.

So, $\log n = m\log 2$ which implies $m = \log_2 n$.

Substituting in (2), we get, $S(m) = \theta(\log\log_2 n)$ or $T(n) = \theta(\log\log_2)$

2.9 Recursion Tree Method

2.9.1 Definition of Recursion Tree:

- This is another method used for solving recurrences.
- The recursion Tree is the binary tree where each node carries the cost of the respective sub-problem.
- After calculating the cost of every node at all the levels, we add the cost of all nodes to obtain the cost of the entire problem.
- Calculating the cost is calculating the running time complexity of a given recurrence.

2.9.2 Three-Step Process to Solve Recurrences Using Recursion Tree Method:

Step 01: Formulate the recursion tree for the given recurrences based on the function given and the cost at each level.

Step-02: Calculate the following from the obtained recursion tree:

- Determine the cost of each node and then the cost at each level.
- Calculate the total number of the levels in the recursion tree.
- The next step is to calculate the number of nodes at the last level.
- Further, calculate the cost of the last level in the recursion tree.

Step-03: The last step is to add the cost obtained at each level of the recursion tree to get the running time complexity of a recurrence.

2.9.3 Sample Recurrences using Recursion Tree Method:

Problem-01:

Solve the given recurrence using recursion tree method-

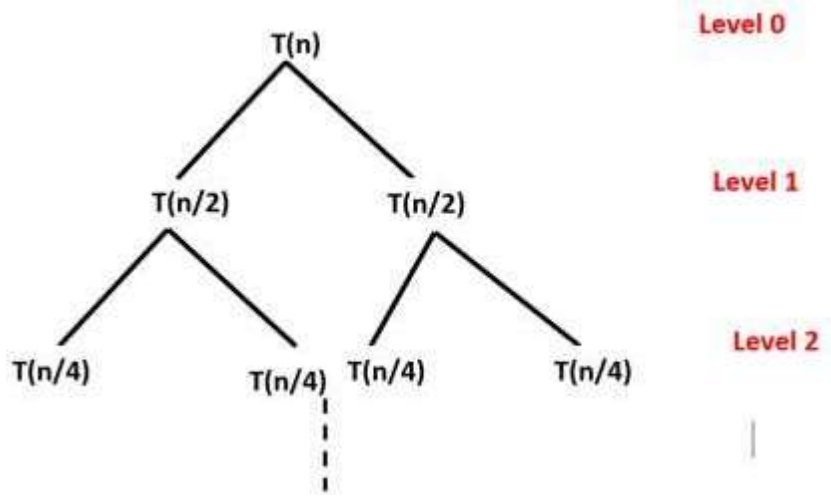
$$T(n) = 2T(n/2) + n$$

Solution:

Step-01:

Formulate the recursion tree for the given recurrences based on the function given and the cost at each level. The given recurrence function works as:

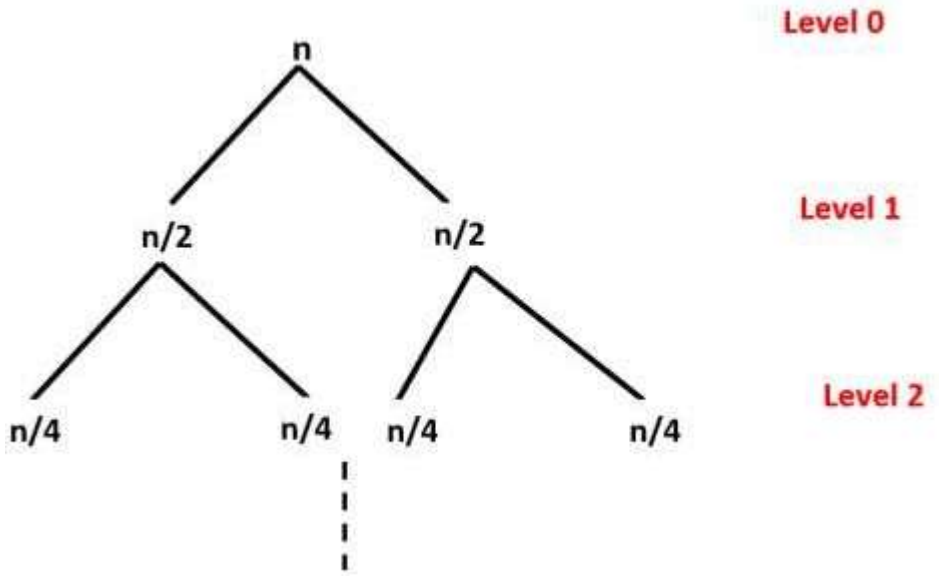
- a problem is given of size n, which will be divided into 2 sub-problems of size n/2.
- Then, each sub-problem of size n/2 will further get divided into 2 sub-problems of size n/4 and so on.
- At the bottom-most layer, the size of sub-problems will reduce to 1.



After the working of the recurrence function T (n), we will further see the cost function through which we can determine the cost at each level of the recursion tree: The given cost recurrence shows:

- Determine the cost of dividing a problem of size n into its 2 sub-problems of size n/2 and then merging the solutions back to the size of the problem as n.
- Determining the cost of dividing a problem of size n/2 into its 2 sub-problems of size n/4 and then merging the solutions back to the size of the problem as n/2
- It determines the cost of dividing a problem of size n/4 into its 2 sub-problems of size n/8 and then merging the solutions back to the size of the problem as n/4 and so on.

Above mentioned observations can be shown diagrammatically as follows in the form of the cost function to calculate the cost at each level of the recursion tree.



Step-02(a): Determining the cost at each level of the recursion tree:

- The cost of the node available at level 0 is n,
- Cost of the nodes available at level 1 is n/2 + n/2 = n
- Cost of the nodes available at level 2 is n/4 + n/4 + n/4 + n/4 = n and so on.

Step-02(b): Calculating the number of the levels available in the recursion tree:

- The size of the sub-problem at level 0 is $n/2^0$
- The size of the sub-problem at level 0 is $n/2^1$
- The size of the sub-problem at level 0 is $n/2^{2,}$ and so on.

Similarly, we can calculate the size of the sub-problem at level i

Size of sub-problem at level i = $n/2^i$

Consider at any level-x (last level), size of sub-problem becomes 1. Then-

$n / 2^x = 1$

$2^x = n$

Taking log on both sides, we get: x log2 = log n , $x = \log_2 n$

∴ Hence, the total number of levels in the recursion tree = $\log_2 n + 1$

Step-02(c): Calculating the number of nodes in the last level:

- Level-0 has 2^0 nodes, i.e., 1 node
- Level-1 has 2^1 nodes, i.e., 2 nodes
- Level-2 has 2^2 nodes, i.e., 4 nodes

Continuing similarly to the number of nodes at the last level, we have-

Level-$\log_2 n$ has $2^{\log_2 n}$ nodes, i.e., n nodes

Step-02(d) : Calculating the cost of the last level-

Cost of last level = n x T (1) = θ(n)

Step-03: The last step is to add the cost obtained at each level of the recursion tree to get the running time complexity of a recurrence.

T (n) = n + n + n + n +……. + θ (n)

(Total Levels $\log_2 n$ +1) Last level

= n x $\log_2 n$ + θ (n)

= n$\log_2 n$ + θ (n)

= θ (n$\log_2 n$)

Problem-02: Solve the given recurrence using recursion tree method:

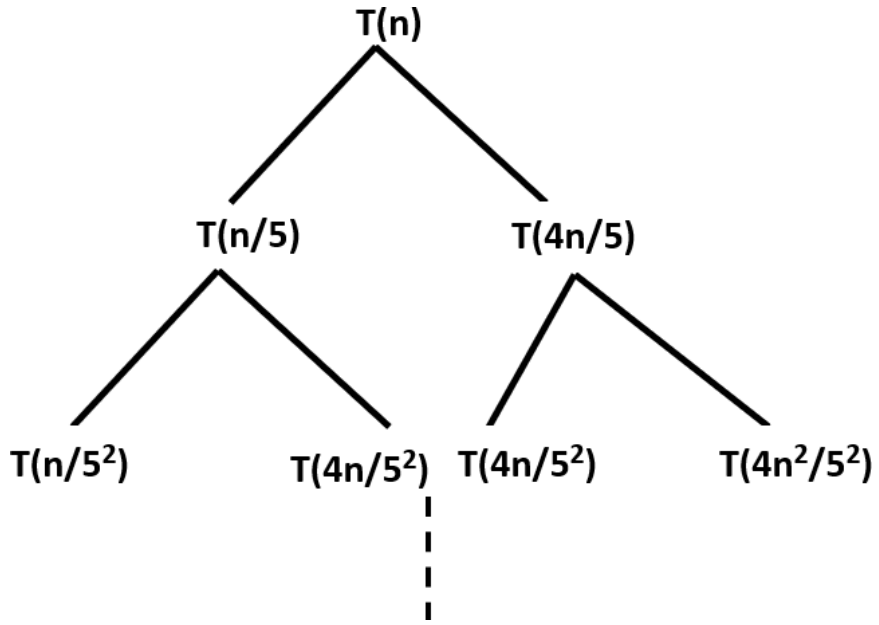
T (n) = T (n/5) + T (4n/5) + n

Solution-

Step-01: Formulate the recursion tree for the given recurrences based on the function given and the cost at each level. The given recurrence function works as:

- Division of problem of size n into two sub-problems of size n/5 and size 4n/5
- Further on the left side, the sub-problem of size n/5 will get divided into 2 sub-problems- size $n/5^2$ and size $4n/5^2$.
- On the right side, the sub-problem of size 4n/5 will get divided into 2 sub-problems- size $4n/5^2$ and size $4^2n/5^{2,}$ and so on.
- At the last level, the size of sub-problems will further reduce to 1.

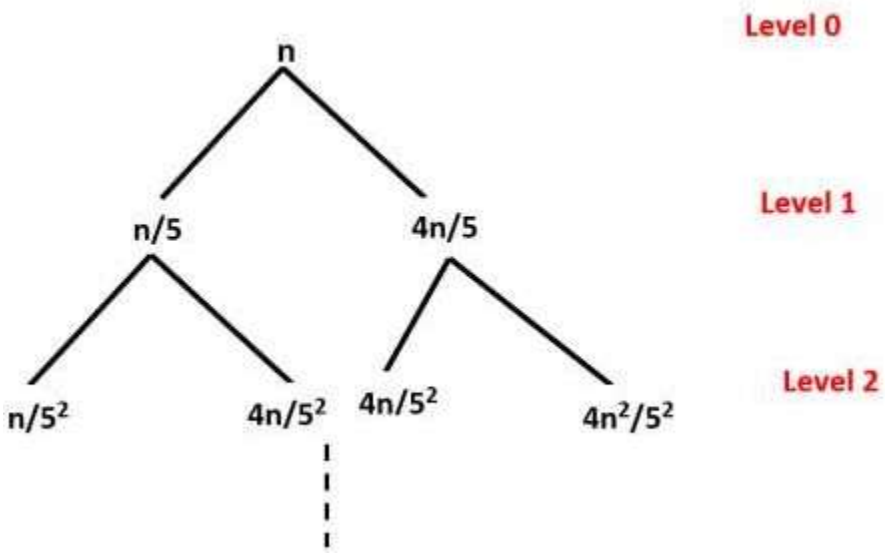
T (n) function will be elaborated as follows:



Expansion of T (n) Function to the cost function is as follows:

- First, Determine the cost of dividing a problem of size n into its 2 sub-problems of size n/5 and 4n/5 and then merging the solutions back to the size of the problem as n.
- Determining the cost of dividing a problem of size n/5 into its 2 sub-problems of size n/25 and 4n/25 and then merging the solutions back to the size of the problem as n/5
- It determines the cost of dividing a problem of size 4n/5 into its 2 sub-problems of size 4n/25 and 16n/25 and then merging the solutions back to the size of the problem as 4n/5 and so on.

This is illustrated through the following recursion tree where each node represents the cost of the corresponding sub-problem-



Step-02(a): Determine cost of each level-

- Cost of level-0 = n
- Cost of level-1 = n/5 + 4n/5 = n
- Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$

Step-02(b): Determine total number of levels in the recursion tree. We will consider the rightmost sub tree as it goes down to the deepest level-

- Size of sub-problem at level-0 = $(4/5)^0 n$
- Size of sub-problem at level-1 =$(4/5)^1 n$
- Size of sub-problem at level-2 =$(4/5)^2 n$

Continuing in similar manner, we have-

Size of sub-problem at level-i = $(4/5)^i n$

Suppose at level-x (last level), the size of the sub-problem becomes 1. Then-

$(4/5)^x n = 1$

$(4/5)^x = 1/n$

Taking log on both sides, we get-

X log (4/5) = log (1/n)

$x = \log_{5/4} n$

∴ Total number of levels in the recursion tree = $\log_{5/4} n + 1$

Step-02(c): Determine the number of nodes in the last level-

- Level-0 has 2^0 nodes, i.e., 1 node
- Level-1 has 2^1 nodes, i.e., 2 nodes
- Level-2 has 2^2 nodes, i.e., 4 nodes

Continuing similarly, we have-

Level-$\log_{5/4} n$ has $2^{\log_{5/4} n}$ nodes

Step-02(d): Determine cost of the last level-

Cost of last level = $2^{\log_{5/4} n} \times T(1) = \theta(2^{\log_{5/4} n}) = \theta(n^{\log_{5/4} 2})$

Step-03: Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic Notation-

$$T(n) = \{\underbrace{n + n + n + \ldots\ldots\ldots}_{\text{For } \log_{5/4} n \text{ levels}}\} + \Theta(n^{\log_{5/4} 2})$$

$= n\log_{5/4} n + \theta\,(n^{\log_{5/4} 2})$

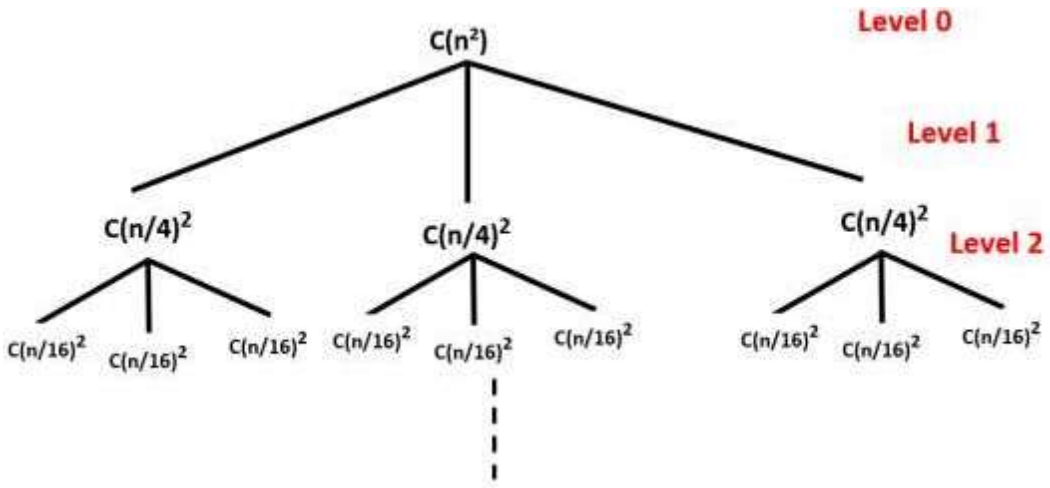
$= \theta\,(n\log_{5/4} n)$

Problem-03: Solve the given recurrence relation using the recursion tree method-

$T(n) = 3\,T(n/4) + cn^2$

Solution-

Step-01: Draw a recursion tree based on the given recurrence relation-



Step-02(a): Determine cost of each level-

- Cost of level-0 = cn^2
- Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
- Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

Step-02(b): Determine total number of levels in the recursion tree-

- Size of sub-problem at level-0 = $n/4^0$
- Size of sub-problem at level-1 = $n/4^1$
- Size of sub-problem at level-2 = $n/4^2$

Continuing in similar manner, we have-

Size of sub-problem at level-i = $n/4^i$

Suppose at level-x (last level), the size of the sub-problem becomes 1. Then-

$n /4^x = 1$

$4^x = n$

Taking log on both sides, we get-

$x\log 4 = \log n$

$x = \log_4 n$

∴ Total number of levels in the recursion tree = $\log_4 n + 1$

Step-02(c): Determine the number of nodes in the last level-

- Level-0 has 3^0 nodes, i.e., 1 node
- Level-1 has 3^1 nodes, i.e., 3 nodes
- Level-2 has 3^2 nodes, i.e., 9 nodes

Continuing similarly, we have-

Level-$\log_4 n$ has $3^{\log_4 n}$ nodes, i.e., $n^{\log_4 3}$ nodes

Step-02(d): Determine cost of the last level-

Cost of last level = $n^{\log_4 3} \times T(1) = \theta(n^{\log_4 3})$

Step-03: Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic Notation-

$$T(n) = \underbrace{\{cn^2 + (3/16)cn^2 + (9/16^2)cn^2 + \ldots\ldots\ldots\}}_{\text{For } \log_4 n \text{ levels}} + \Theta(n^{\log_4 3})$$

$= cn^2 \{1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots\} + \theta(n^{\log_4 3})$

Now, $\{1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots\}$ forms an infinite Geometric progression.

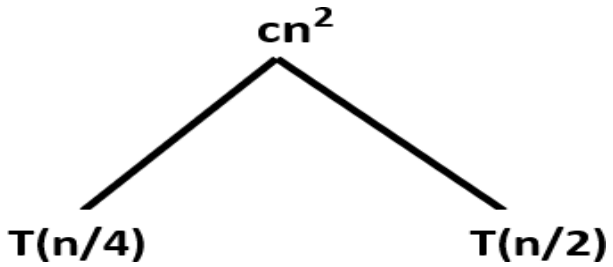
On solving, we get-

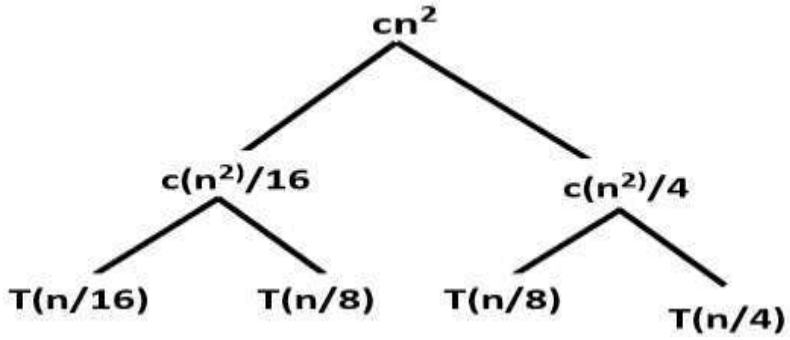
$= (16/13)\, cn^2 \{1 - (3/16)^{\log_4 n}\} + \theta(n^{\log_4 3})$

$= (16/13)\, cn^2 - (16/13)\, cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3}) =$ **$O(n^2)$**

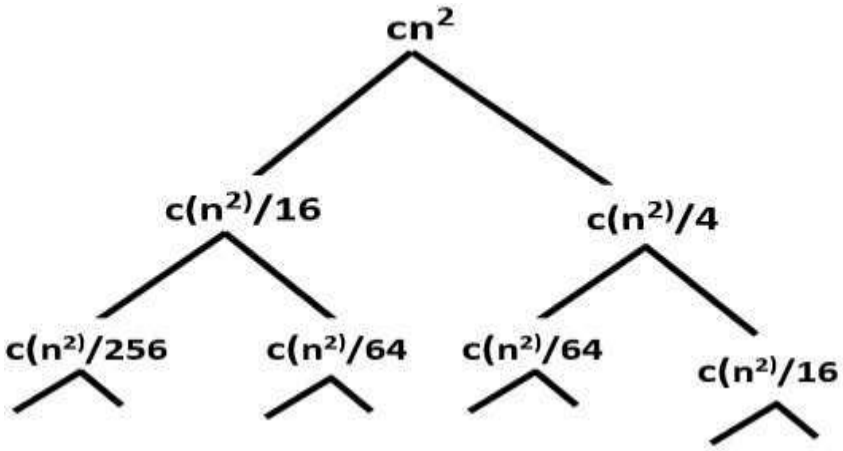
Problem 04: Solve the given recurrence using recursion tree method:

$T(n) = T(n/4) + T(n/2) + cn^2$





Breaking down further give us following



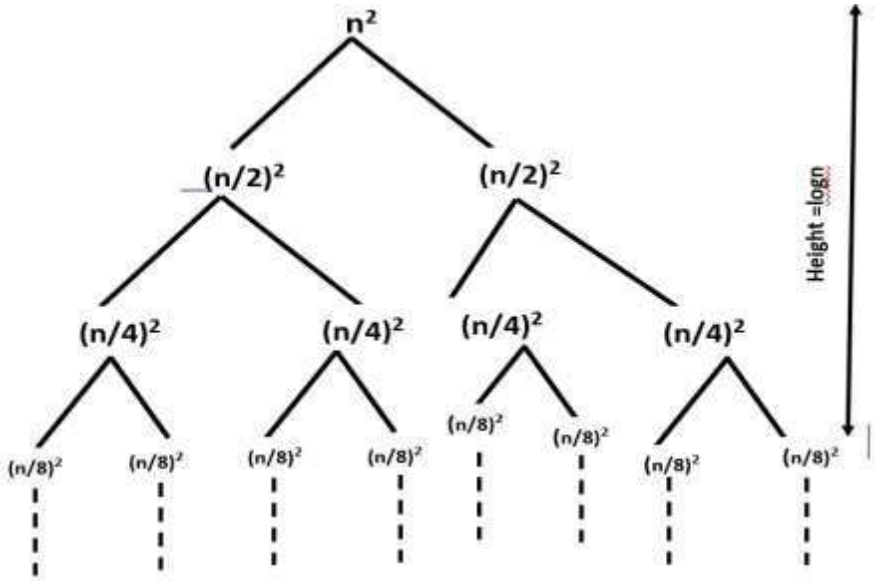
For calculating the value of the function T (n), we have to calculate the sum of the cost of each node at every level. After calculating the cost of all the levels, we generate the series T (n) = c (n^2+ 5(n^2)/16 + 25(n^2)/256) +-------

Mentioned series is a geometric progression with a ratio of 5/16.

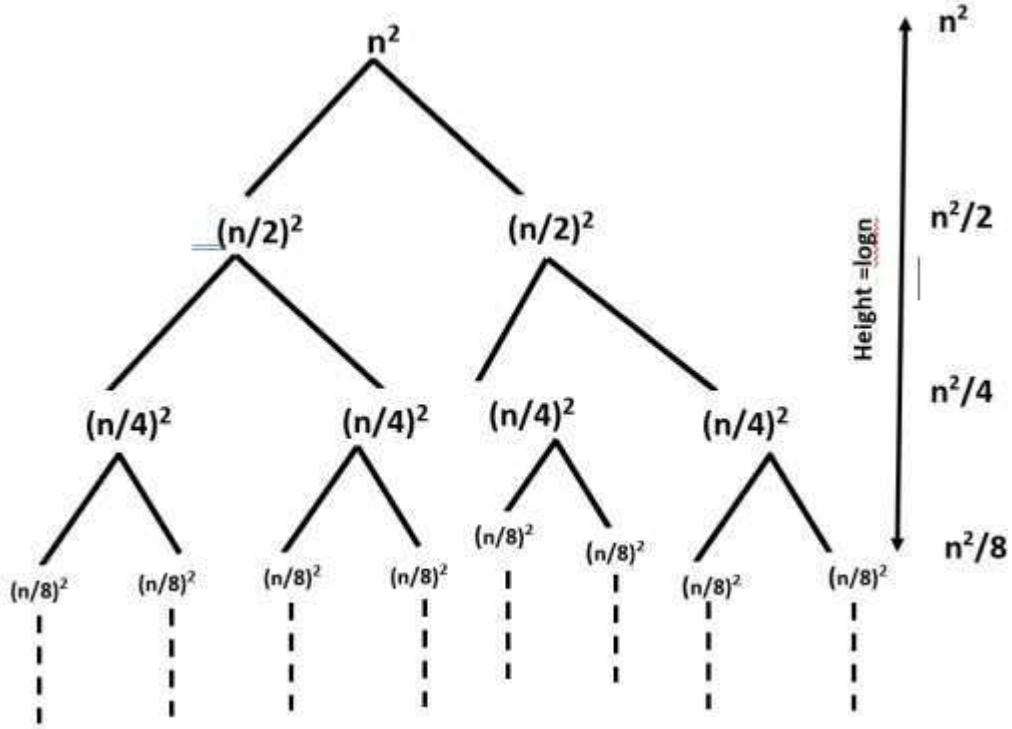
To get an upper bound, we can apply the formula of the sum to the infinite series. We get the sum as (n^2)/ (1 - 5/16), which is O (n^2).

Problem 05: Solve the given recurrence using recursion tree method:
T (n) = 2T (n/2) + n^2.

The recursion tree for this recurrence has the following form:



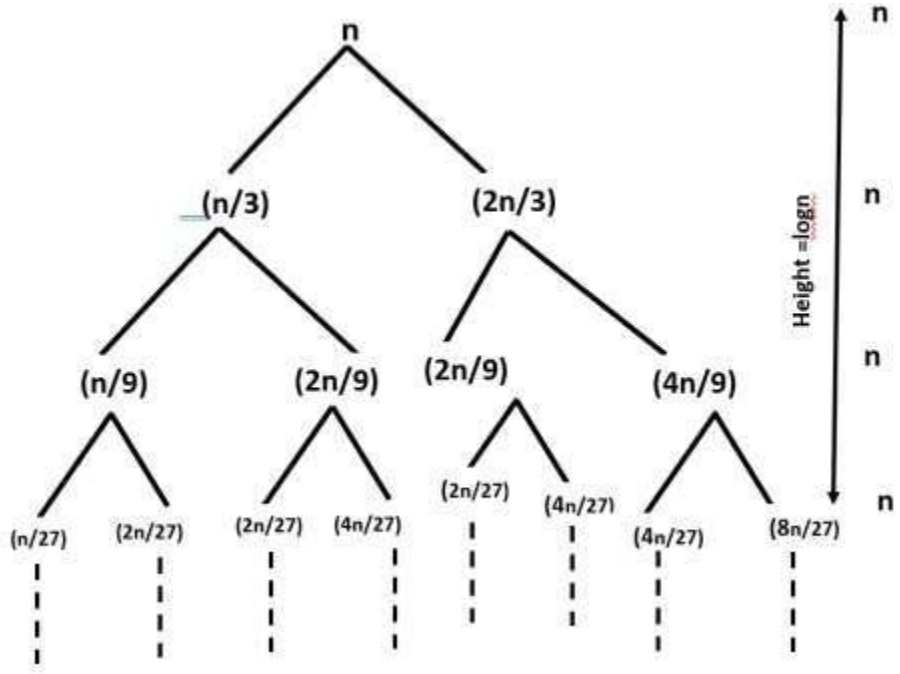
In this case, we can simply calculate the cost at each level by adding the costs of each node at the same level. After adding all the costs at all the levels, we get the geometric series.



Given geometric series, this will limit the running time to $O(n^2)$. Cost at the last level will not bother as it continuously decreases from root to the leaf. Hence we can say that the running time complexity of given recurrence is $O(n^2)$.

Problem 06: Solve the following recurrence relation using the recursion tree method:
T (n) = T (n/3) + T (2n/3) + n

The recurrence tree is:



Given recursion tree is not balanced: the shortest path is the leftmost one whose length is $log_{1/2} n$, *and* the longest path is the rightmost one whose length is $log_{3/2} n$.

The cost at each level of the recursion tree is n.

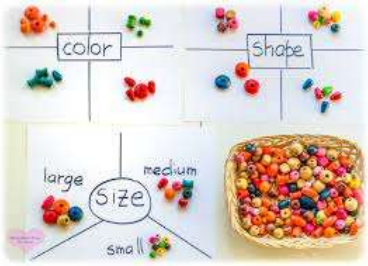
Therefore in the case of the upper bound, the running time complexity is O(n $log_{3/2} n$), *and* in the case of the lower bound, the running time complexity is O(n $log_{1/2} n$)

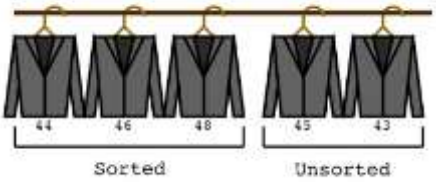
2.1. Introduction to Sorting

2.1.1. Case Studies to understand the requirement of Sorting:

In real life scenario we knowingly and unknowingly use sorting extensively.

1. In playing cards, a player gets a bunch of card during play. We need to arrange the cards of a particular player in ascending or descending order.
2. When we want to call someone we use only few characters of name because contacts in our phone are arranged in lexicographically (sorted).
3. We usually arrange student marks in decreasing order to get top three students name and roll number.
4. We might want to arrange sales data by calendar month so that we can produce a graph of sales performance.
5. We have a sorted Dictionary / Telephone Directory and an un-sorted Dictionary / Telephone Directory with the same collection of words / telephone numbers printed on about 500 pages. We will save about 99% of our time by using the sorted dictionary Telephone Directory.
6. The tailor shop puts stitched clothes either in descending order or ascending order of size.
7. Searching any item in junk drawer of our kitchen is time consuming if items are not arranged in the drawer. It is easier and faster to locate items in a sorted list than unsorted.



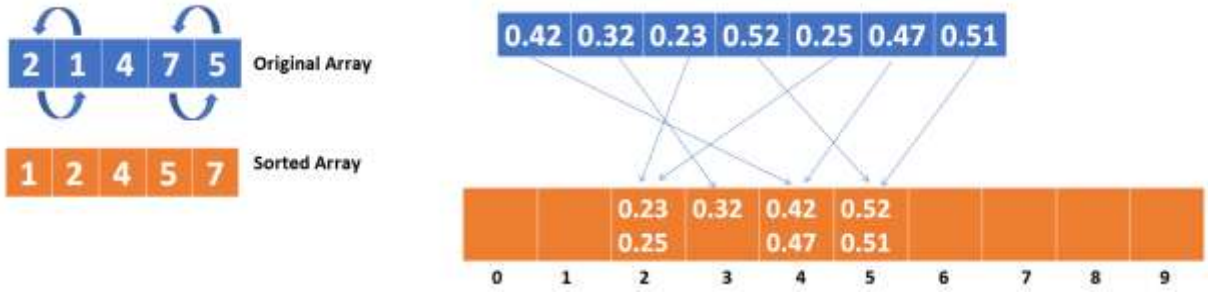


Sorting is the process of arranging data into meaningful sequence so that we can consider it with more ease and convenience. Sort means arranging items into an order either alphabetical or, numerical.

Sorting is particularly helpful in the context of computer science or, programming for the same reason it is important in everyday life. Sorting a list of items can take a long time, especially if it is a large list. A computer program can be created to do this, making sorting a list of data much easier. From a strictly human-friendly perspective, it makes a single dataset a whole lot easier to read. It makes it easier to implement search algorithms in order to find or retrieve an item from the entire data set. Sorting algorithms can be used in a program to sort an array for later searching or writing out to an ordered file or report.

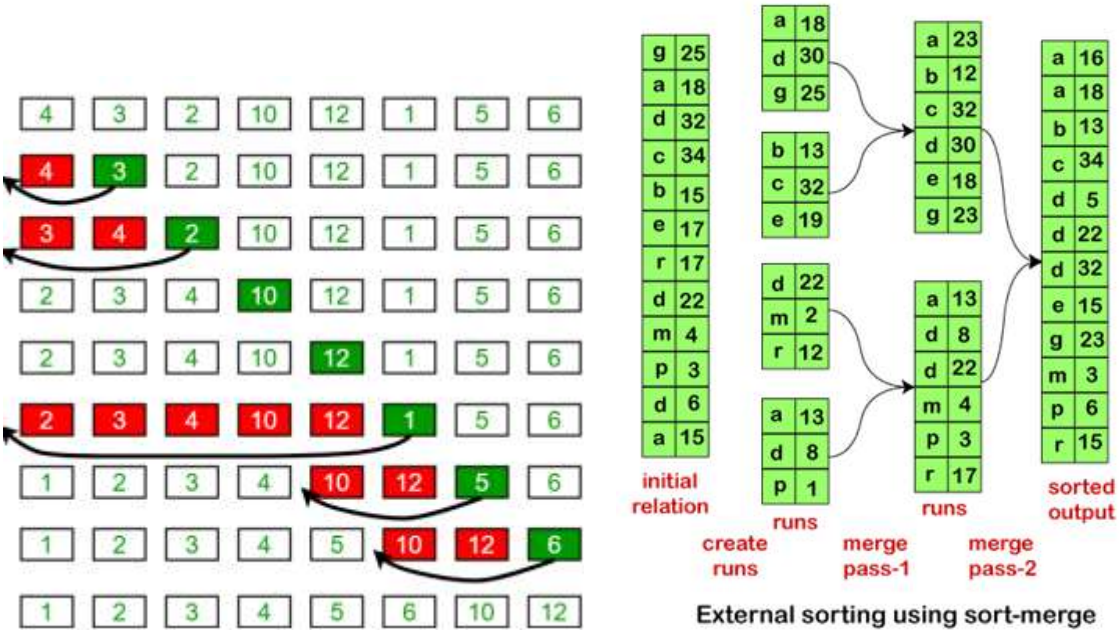
2.1.2 Types of Sorting Algorithm:

(a)*In-place sorting and not-in-place sorting* - In in-place sorting algorithm we use fixed additional space for producing the output (modifies only the given list under consideration). It sorts the list only by modifying the order of the elements within the list. e.g. Bubble sort, Comb sort, Selection sort, Insertion sort, Heap sort, and Shell sort. In not-in-place sorting, we use equal or more additional space for arranging the items in the list. Merge-sort is an example of not-in-place sorting.

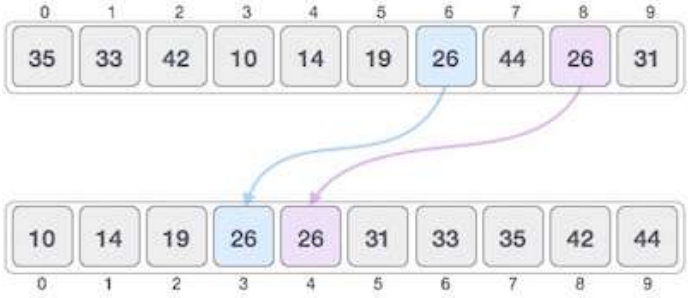


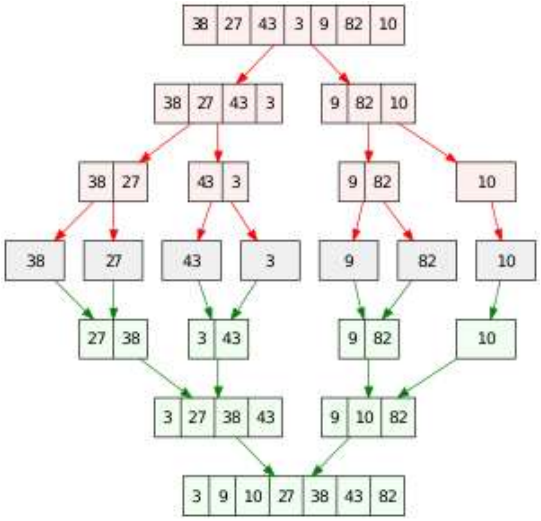
(b) *Stable sorting and Unstable sorting*– In stable sorting algorithm the order of two objects with equal keys in output remains same after sorting as they appear in the input array to be sorted. e.g. Merge Sort, Insertion Sort, Bubble Sort, and Binary Tree Sort. If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable Sorting. e.g. Quick Sort, Heap Sort, and Selection sort

(c) ***Internal sorting and External sorting***- If the input data is such that it can be adjusted in the main memory at once or, when all data is placed in-memory it is called internal sorting e.g. Bubble Sort, Insertion Sort, Quick Sort. If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting. External sorting algorithms generally fall into two types, distribution sorting, which resembles quick sort, and external merge sort, which resembles merge sort. The latter typically uses a hybrid sort-merge strategy



(d) ***Adaptive Sorting and Non-Adaptive Sorting***- A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them. A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.





(e) ***Comparison based and non-comparison based*** - Algorithms, which sort a list, or an array, based only on the comparison of pairs of numbers, and not any other information (like what is being sorted, the frequency of numbers etc.), fall under this category. Elements of an array are compared with each other to find the sorted array. e.g. Bubble Sort, Selection Sort, Quick Sort, Merge Sort, Insertion Sort. In non-comparison based sorting, elements of array are not compared with each other to find the sorted array. e.g. Radix Sort, Bucket Sort, Counting Sort.

Here, in the subsequent portion of this chapter, we will discuss about following Sorting Techniques and their variants.

1. Bubble Sort
2. Selection Sort
3. Insertion Sort

4. Heap Sort
5. Merge Sort
6. Quick Sort
7. Counting Sort
8. Radix Sort
9. Bucket Sort

The Sorting Techniques can broadly be categorised (based on time complexity) into

- Order of n^2 (Bubble Sort, Selection Sort, Insertion Sorts),
- Order of nlog n (Heap Sort, Quick Sort, Merge Sort)
- Order of n (Counting Sort, Bucket Sort and Radix Sort)

The document discusses about various cases related to these sorting and strategies to improve the run time.

2.2. Bubble Sort:

2.2.1. Working of Bubble Sort:

Consider a situation when we pour some detergent in the water and shake the water, the bubbles are formed. The bubbles are formed of different sizes. Largest volume bubbles have a tendency of coming to the water surface faster than smaller ones.

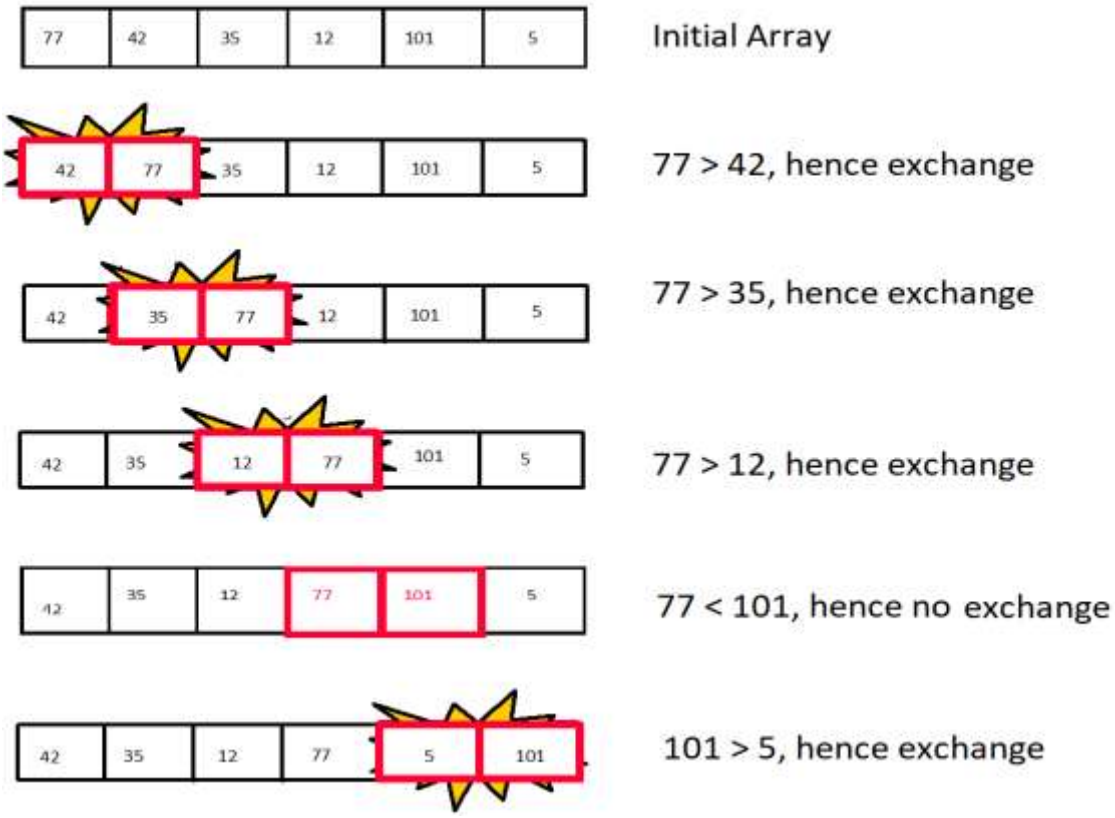
Bubble sort, which is also known as **sinking sort**, is a simple Brute force Sorting technique that repeatedly iterates through the item list to be sorted. The process compares each pair of adjacent items and swaps them if they are in the wrong order. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating with one less comparison than the last pass. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a concern).

In Bubble Sort Algorithm largest element is fixed in the first iteration followed by second largest element and so on so forth. In the process 1^{st} element is compared with 2^{nd} and if previous element is larger than the next one, elements are exchanged with each other. Then 2^{nd} and 3^{rd} elements are compared and if second is larger than the 3^{rd}, they are exchanged. The process repeats and finally $(n-1)^{st}$ element gets compared with n^{th} and if previous one is larger than the next one, they are exchanged. This completes the first iteration. As a result largest element occupies its appropriate position in the Sorted array i.e. last position.

In the next iteration, the same process is repeated but one less comparison is performed than previous iteration (as largest number is already in place). As a result of second iteration, second largest number reaches to its appropriate position in the sorted array.

The similar iterations are repeated n-1 times. The result is the sorted array of n elements.

Below given figure shows how Bubble Sort works:



Above diagram shows the first pass of the Bubble Sort.

42	35	12	77	5	**101**	**Result of Iteration 1**
35	12	42	5	**77**	**101**	**Result of Iteration 2**
12	35	5	**42**	**77**	**101**	**Result of Iteration 3**
12	5	**35**	**42**	**77**	**101**	**Result of Iteration 4**
5	**12**	**35**	**42**	**77**	**101**	**Result of Iteration 5**

In first pass

```
FOR j=1 TO N-1 DO
        IF A[j] >A[j+1] THEN
                Exchange (A[j], A[j+1])
```

Total N-1 passes of similar nature

ALGORITHM Bubble Sort(A[ ],N)

BEGIN:

```
        FOR i=1 TO N-1 DO
           FOR j= 1 To N-i DO
              IF A[j] >A[j+1] THEN
                 Exchange (A[j], A[j+1])
```

END;

Analysis:

N-1comparisons in first iteration

N-2 comparison in second iteration

N-3 comparison in third iteration

…

1 Comparisons in $(N-1)^{st}$ iteration

Total comparisons $= (N-1) + (N-2) + (N-3) + \ldots + 1$

$= N(N-1)/2$

$= N^2/2 - N/2$

If exchanges take place with each comparison, Total number of statements to be executed are $(N^2/2 - N/2)*3$ as 3 statements are required for exchange.

The **Time Complexity** can be written as $\theta(N^2)$

There are 2 extra variables used in the logic. Hence **space Complexity** is $\theta(1)$

2.2.2. Optimized Bubble Sort

In bubble sort time complexity for the case when elements are already sorted is $\theta(n^2)$, because fixed number of iterations need has to be performed.

There are two scenarios possible –

Case 1 - When elements are already sorted: Bubble Sort does not perform any swapping.

Case 2 - When elements are sorted after k-1 passes: In the k^{th} pass no swapping occurs.

A small change in Bubble Sort Algorithm above makes it optimized. If no swap happens in some iteration means elements are sorted and we should stop comparisons. This can be done by the use of a flag variable.

ALGORITHM Bubble Sort Optimized (A [ ],N)

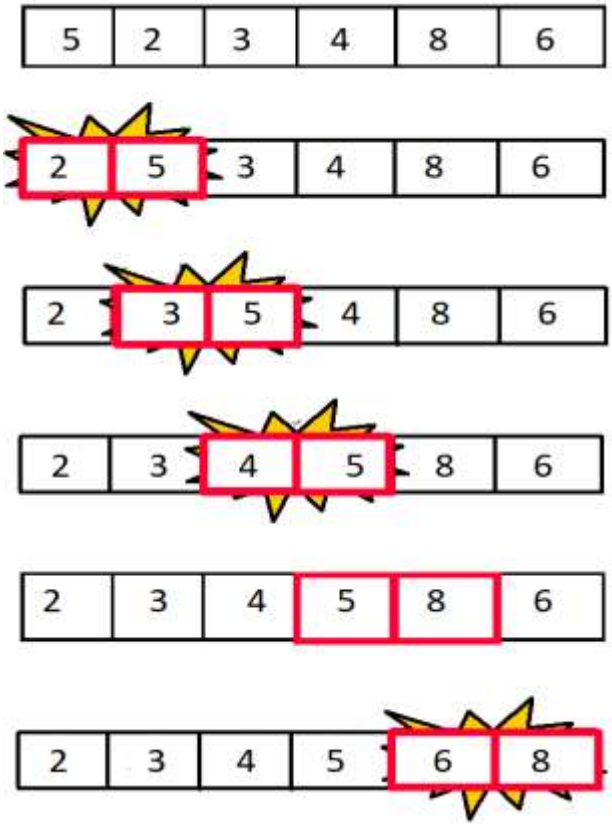
```
BEGIN:
    FOR i=1 TO N-1 DO
      FLAG =1
        FOR j= 1 To N-i DO
            IF A[j] >A[j+1] THEN
                Exchange (A[j], A[j+1])
                FLAG=0
        IF FLAG ==1 THEN
            RETURN
END;
```

Optimized Bubble Sort Analysis

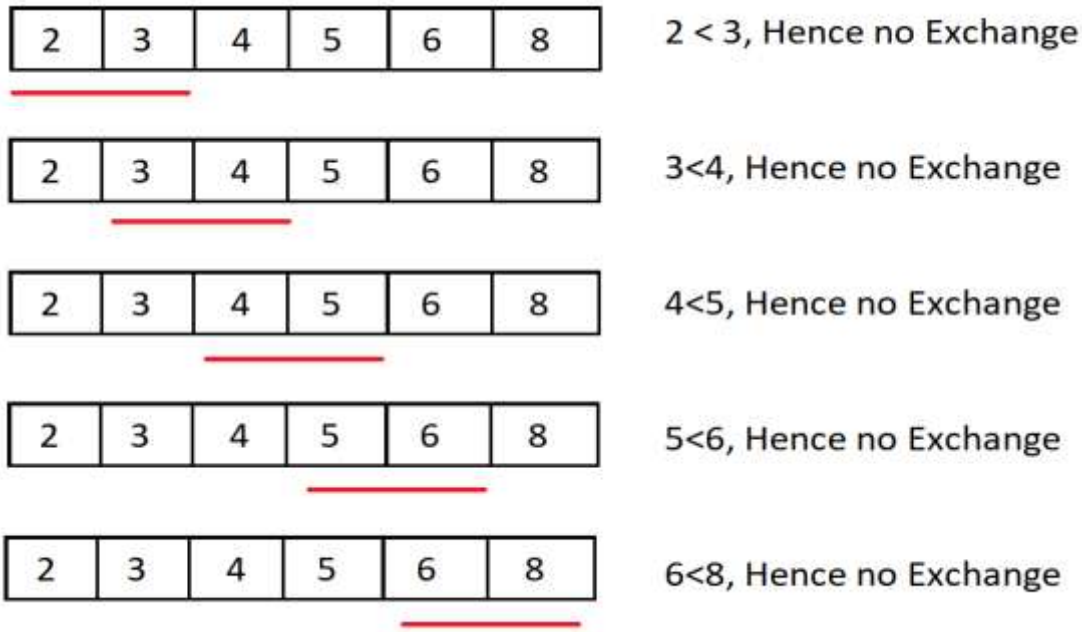
If elements are already sorted then it takes $\Omega(N)$ time because after the first pass flag remains unchanged, meaning that no swapping occurs. Subsequent passes are not performed. A total of N-1 comparisons are performed in the first pass and that is all.

If elements become sorted after k-1 passes, k^{th} pass finds no exchanges. It takes $\theta(N*k)$ effort.

Optimized Bubble Sort (First Pass)



Optimized Bubble Sort (Second Pass)



2.2.3. Sort the link list using optimized Bubble Sort (case study)

Normally we use sorting algorithms for array because it is difficult to convert array algorithms into Linked List.

There are two reasons for this -

1-Linked List cannot be traversed back.

2- Elements cannot be accessed directly (traversal is required to reach to the element).

Bubble Sort Algorithm can be easily modified for Linked List due to two reasons -

1 - In Bubble Sort no random access needed

2 - In Bubble Sort move only forward direction.

ALGORITHM Bubble Sort Link list(START)

BEGIN:

```
Q=NULL
FLAG=TRUE
WHILE TRUE DO
        FLAG=FALSE
        T=START
        WHILE T→Next ! = Q DO
                IF T→Info>T→Next→info THEN
                        Exchange (T, T→Next)
                        FLAG=TRUE
                T=T→Next
        Q=T
```

END;

Algorithm Complexity

If elements are already sorted then it takes Ω(N) time because after the first pass flag remains unchanged, meaning that no swapping occurs. Subsequent passes are not performed. A total of N-1 comparisons are performed in the first pass and that is all.

If elements become sorted after k-1 passes, k^{th} pass finds no exchanges. It takes θ(N*k) effort.

2.3. Selection Sort:

2.3.1. Selection Sort as a variation of Bubble Sort

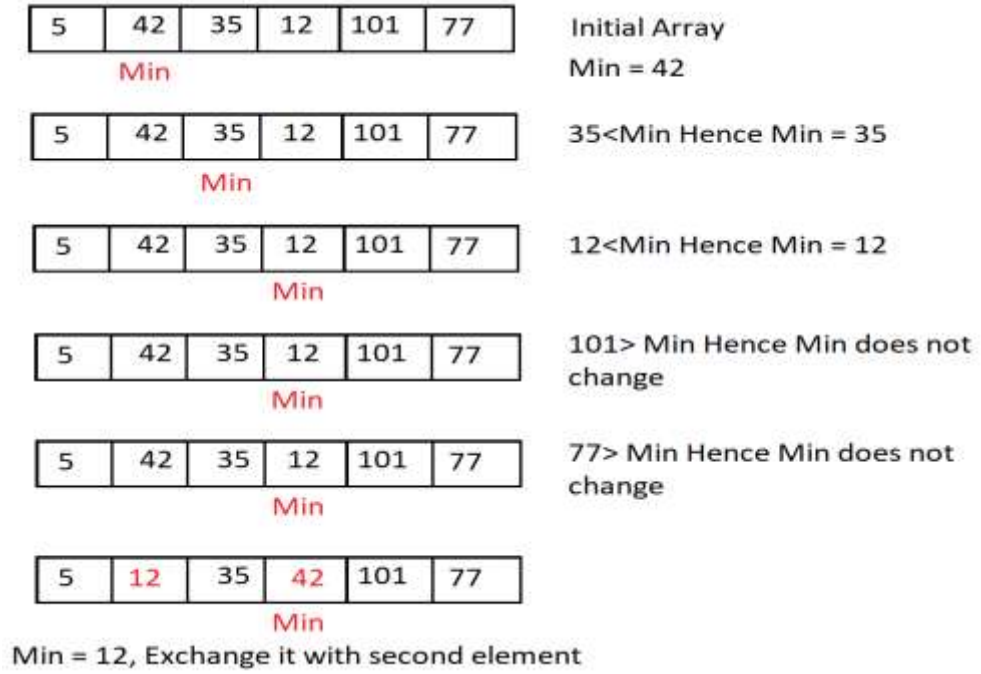
Selection sort is nothing but a variation of Bubble Sort because in selection sort swapping occurs only one time per pass.

In every pass, choose largest or smallest element and swap it with last or first element. If the smallest element was taken for swap, position of first element gets fixed. The second pass starts with the second element and smallest element is found out of remaining N-1 Elements and exchanged with the second element. In the third pass the smallest element is found out of N-2 elements (3^{rd} to N^{th} element) and exchanged with the third element and so on so forth. The same is performed for N-1 passes.

Number of comparisons in this Algorithm is just the same as that of Bubble Sort but number of swaps in this 'N' (as compared to N*(N-1)/2 Swaps in Bubble Sort.

The First Pass

The Second Pass



In first pass

```
Min =1;
FOR j=2 TO N DO
        IF A[j]<A[min] THEN
                Min = j;
Exchange (A[1], A[Min])
```

Total N-1 passes of similar nature

ALGORITHM Selection Sort(A[ ],N)

```
BEGIN:
    FOR i=1 TO N-1 DO
            Min=i
             FOR j= i+1 TO N DO
                    IF A[j] <A[Min] THEN
                            Min = i
            Exchange(A[i],A[Min])
END;
```

Time Analysis (Total (N-1) iterations)

N-1 comparisons in first iteration

N-2 comparison in second iteration

N-3 comparison in third iteration

…

1 Comparison in $(N-1)^{st}$ iteration

Total = (N-1) + (N-2) + (N-3) + … +1

= N(N-1)/2 = $N^2/2 – N/2$

1 Exchange per iteration hence total exchanges are N-1. Total effort for N-1 iterations=3 *(N-1)

Total Effort = 3*(N–1) + $N^2/2 – N/2$ = $N^2/2 + 5/2*N - 3$

As there is no best case possible as all iterations are compulsory, Complexity should be written in Theta notation i.e. $\boldsymbol{\theta}(N^2)$

Space Complexity remains θ(1) as only 3 variables (i, j, Min) are used in the logic.

Selection Sort Scenario

As selection sort takes only O(N) swaps in worst case, it is the best suitable where we need minimum number of writes on disk. If write operation is costly then selection sort is the obvious choice. In terms of write operations the best known algorithm till now is cycle sort, but cycle sort is unstable algorithm.

2.3.2. Cycle Sort

Cycle sort is an in-place, unstable, comparison sort that is theoretically optimal in terms of the total number of writes to the original array. It is optimal in terms of number of memory writes. It minimizes the number of memory writes (Each value is either written zero times, if it's already in its correct position, or written one time to its correct position.)

It is based on the idea that array to be sorted can be divided into cycles. Cycles can be visualized as a graph. We have N nodes and an edge directed from node i to node j if the element at i^{th} index must be present at j^{th} index in the sorted array.

Example- Arr[]={10,5,2,3}

10	5	2	3

Step1- let us suppose that

Cycle=0

Item=arr[0]

Now our aim to find out the position where we put the item

Pos=cycle

I=pos+1

WHILE i<n DO //n is number of item

IF arr[i]<item THEN

Pos++

Swap(arr[pos],item)

Here after execution of while loop we get the value of pos=3

Swap(arr[3],10)

10	5	2	10

Step2-

Cycle =0

Item=3

Pos=cycle

I=pos+1

WHILE i<n DO //n is number of item

IF arr[i]<item THEN

Pos++

Swap(arr[pos],item)

Here after execution of while loop we get the value of pos=1

Swap(arr[1],3)

10	3	2	10

Step 3-

Cycle =0

Item=5

Pos=cycle

I=pos+1

WHILE i<n DO //n is number of item

IF arr[i]<item THEN

Pos++

Swap(arr[pos],item)

Here after execution of while loop we get the value of pos=2

Swap(arr[2],5)

10	3	5	10

Step 4-

Cycle =0

Item=2

Pos=cycle

I=pos+1

WHILE i<n DO //n is number of item

IF arr[i]<item THEN

Pos++

Swap(arr[pos],item)

Here after execution of while loop we get the value of pos=0

Swap(arr[0],2)

2	3	5	10

2.3.3. Competitive Coding– Problem Statement-

A company is coming in college campus for recruitment. College has N students and there are exactly N vacancies in the company. Company policy is that to fill all open position from a single college so that they all know each other before joining them and they do not invest extra money for team building activities. Each student is graded and a number is given to every student based on their knowledge. All open position has minimum criteria (number) and a student need to above number to fill the open post. There is no restriction in position allotment and any student can be hired for any one of the N position. You have to find all students are hired or not.

Input format

First line contains number of students N

Second line contains N integers representing knowledge grade of N students

Third line contains N integers represents knowledge criteria for N open position

Output format

Print single line hired or not hired

1-Identify problem statement

Read the story and try to convert it into technical form. For this problem reframes as-

Given two integer array of same size (let arr1, arr2), check if number can be arranged in a way that

Arr 1[i] > Arr2[i] for all values of i

2-Identify Problem Constraints

Knowledge of students and criteria must satisfy

1<N<100

Sample input

5

13 46 34 84 49

10 39 48 79 22

sample output hire

3. Design Logic

1 - Sort both arrays in non-decreasing order.

2 - In a loop compare all elements of Arr1 and Arr2 and if at any step

Get Arr1[i] <Arr[i] print not hire otherwise hire.

4. Implementation-

ALGORITHM CheckHire(Arr1[ ], Arr2[ ], N)

BEGIN:

```
BubbleSort(Arr1)
BubbleSort(Arr2)
FOR i=0 TO N DO
        IF Arr1[i ] < Arr2[i] THEN
                WRITE("Not hire")
                RETURN
        WRITE("Hire")
```

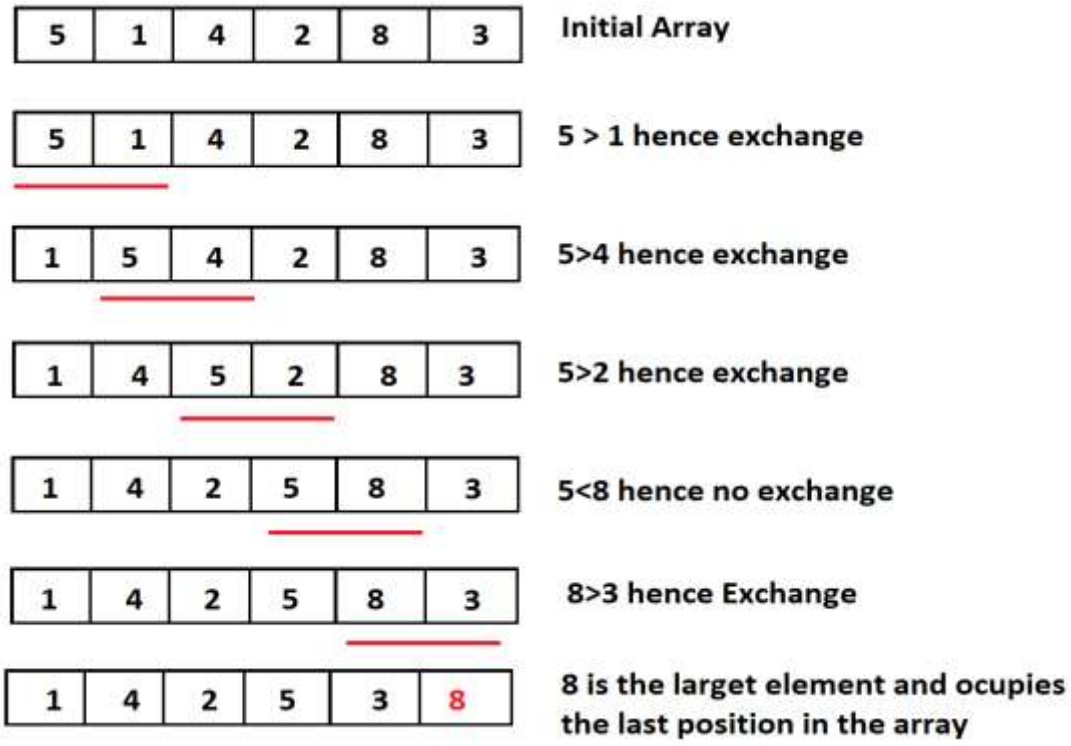
END;

Complexity-$O(N^2)$

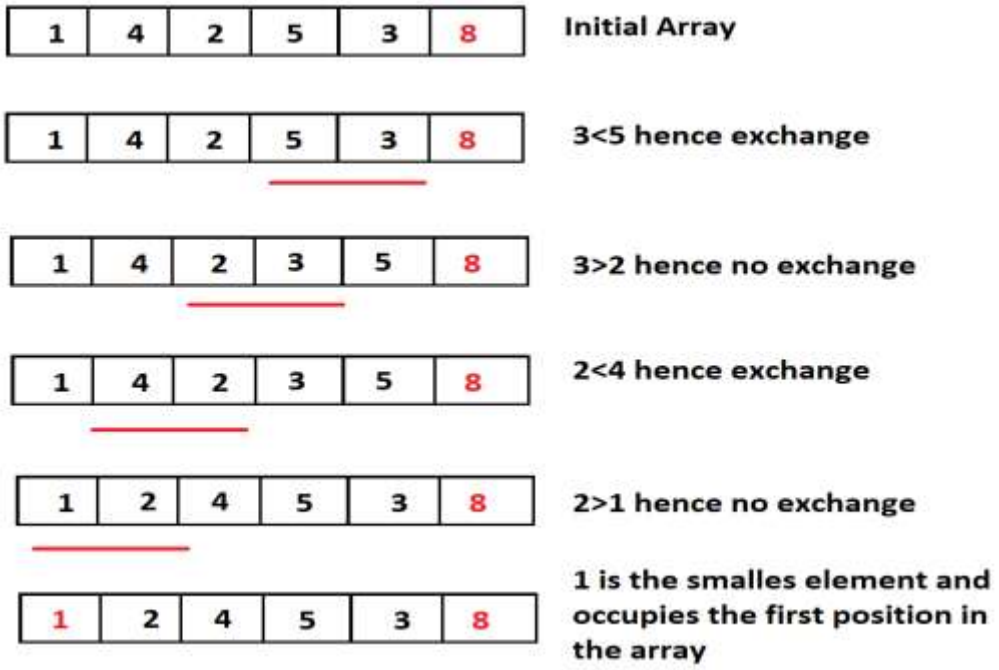
2.3.4. Cocktail Sort-

Cocktail Sort is a variation of Bubble sort in which instead of moving only in forward direction for finding the largest element, the array is traversed alternatively in forward direction (for finding the largest element) and in backward direction (for finding the smallest element).

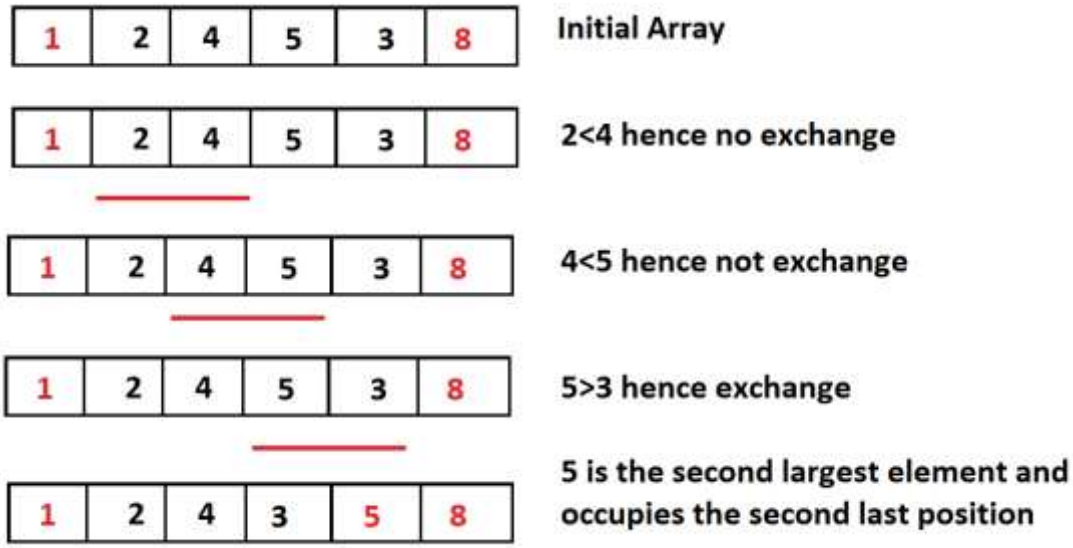
First Forward Pass:



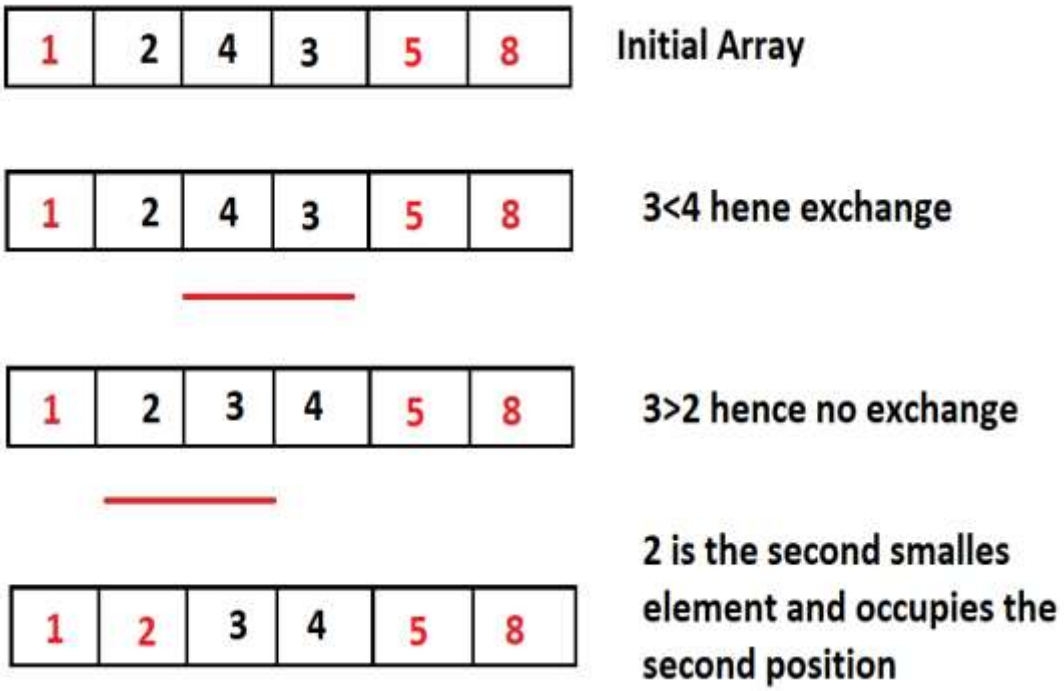
First Backward Pass:



Second Forward Pass:



Second Backward Pass:



ALGORITHM CocktailSort(A[ ], N)

BEGIN:

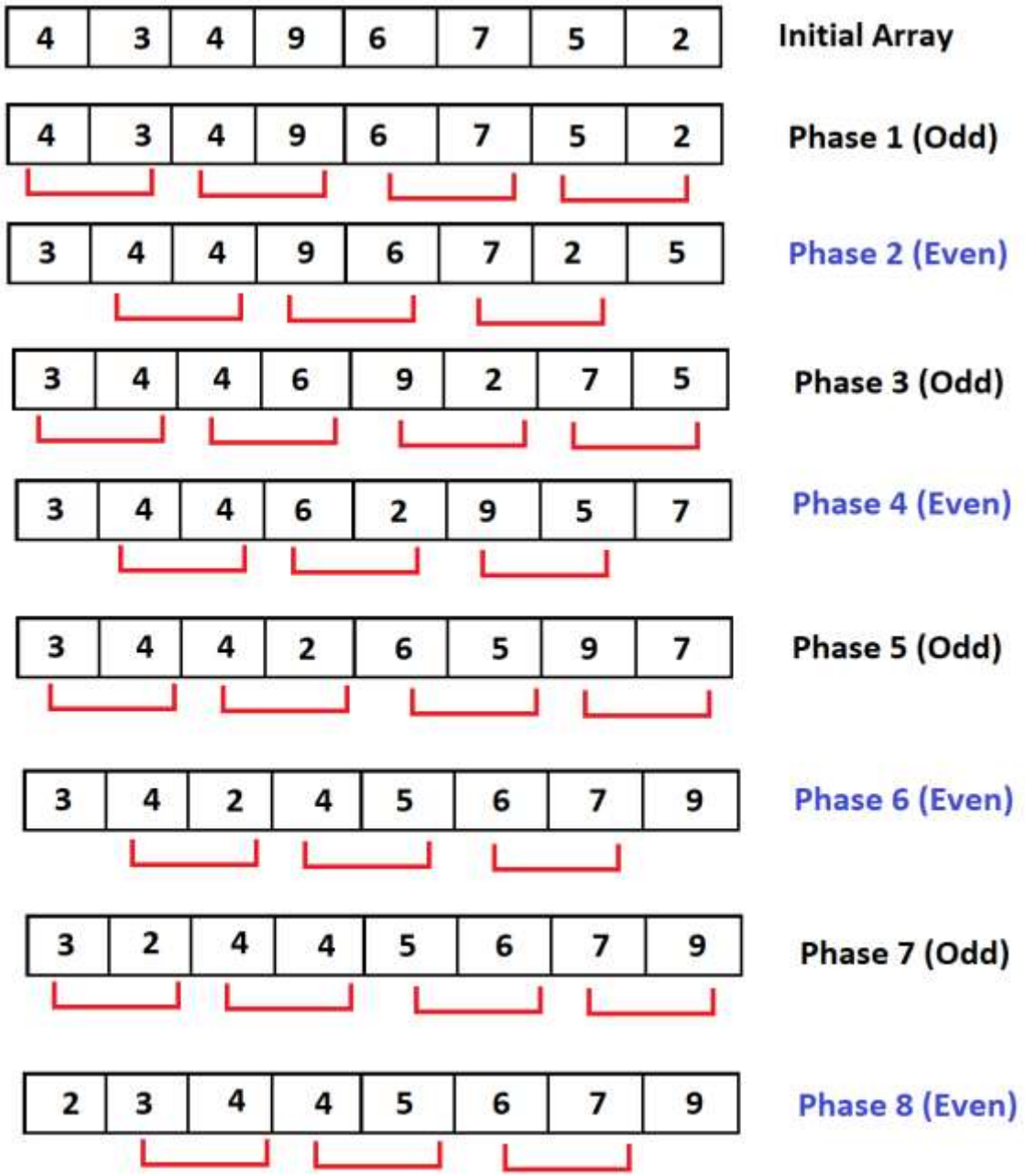
```
j=1, END=N-1, I
WHILE J < END DO
        FLAG=FALSE
        FOR I=J TO END DO
                IF A[i] > A [i+1]
                        Exchange (A[I], A[I+1])
                        FLAG = TRUE
        END = END -1
        FOR I=END-1 TO J STEP - 1 DO
                IF A [i] < A [i-1]
                        Exchange (A [I], A [I-1])
                        FLAG = TRUE
        J=J+1
        IF !FLAG THEN
                RETURN
```

END;

Time Complexity $O(N^2)$ in worst case but in best case $\Omega(N)$

2.3.5.Odd-Even Sort (Brick Sort)

In this sorting each pass has two loops one for even and other for odd indexes. Total cycles to be performed is equal to the number of Elements in the set or When No exchanges take place in pair of odd and even cycle.


4 3 4 9 6 7 5 2
Initial Array
4 3 4 9 6 7 5 2
Phase 1 (Odd)
3 4 4 9 6 7 2 5
Phase 2 (Even)
3 4 4 6 9 2 7 5
Phase 3 (Odd)
3 4 4 6 2 9 5 7
Phase 4 (Even)
3 4 4 2 6 5 9 7
Phase 5 (Odd)
3 4 2 4 5 6 7 9
Phase 6 (Even)
3 2 4 4 5 6 7 9
Phase 7 (Odd)
2 3 4 4 5 6 7 9
Phase 8 (Even)

ALGORITHM Odd Even Sort(A[ ], N)

```
BEGIN:
    FLAG=TRUE
    WHILE FLAG DO
        FLAG=False
        FOR I=1 TO N-1 STEP + 2 DO
            IF A [I] > A [I+1] THEN
                Exchange(ARR[I],ARR[I+1])
                FLAG=TRUE
        FOR I=2 TO N-2 STEP+2 DO
            IF ARR[I] > ARR[I+1] THEN
                Exchange (ARR[I],ARR[I+1])
                FLAG=TRUE
END;
```

2.3.6 Objective Type Questions:

1	The number of swapping needed to sort the numbers 8, 22, 7, 9, 31, 5, 13 in ascending order, using Bubble Sort is
A	13
B	12
C	11
D	10
AN	**C**

2	Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the number of swap operations is minimized in general?
A	Selection
B	Merge
C	Heap
D	Quick
AN	**C**

3	How many comparisons are needed to sort an array of length 5 if a straight selection sort is used and array is already in the opposite order?
A	1
B	5
C	10
D	20
AN	**C**

4	void Sort(intarr[], int n)
	{
	int i, j;
	bool swapped;
	for (i = 0; i < n-1; i++)
	{
	swapped = false;
	for (j = 0; j < n-i-1; j++)
	{
	if (arr[j] >arr[j+1])
	{
	swap(&arr[j], &arr[j+1]);
	swapped = true;
	}
	}
	What is the time complexity of function in best case?

A	O(n)
B	O(1)
C	O(nlog n)
D	O(log n)
AN	**C**

2.3.7. Practice Competitive coding problems:

1. Problem Statement:

•Joey loves to eat Pizza. But he is worried as the quality of pizza made by most of the restaurants is deteriorating. The last few pizzas ordered by him did not taste good :(. Joey is feeling extremely hungry and wants to eat pizza. But he is confused about the restaurant from where he should order. As always he asks Chandler for help.

•Chandler suggests that Joey should give each restaurant some points, and then choose the restaurant having **maximum points**. If more than one restaurant has same points, Joey can choose the one with **lexicographically smallest** name.

•Joey has assigned points to all the restaurants, but can't figure out which restaurant satisfies Chandler's criteria. Can you help him out?

Input format

•First line has N, the total number of restaurants.

•Next N lines contain Name of Restaurant and Points awarded by Joey, separated by a space.

Restaurant name has **no spaces**, all lowercase letters and will not be more than 20 characters.

Output format

Print the name of the restaurant that Joey should choose.

Constraints

•$1 <= N <= 10^5$

•$1 <= Points <= 10^6$

2. Problem Statement

There are N people in city, who wants to visit Kingdom of Dreams. The road of reaching of Kingdom is not so safe. So, they go to kingdom only in a security vehicle which can accommodate at most 2 people.(There is only one security vehicle available in the town as it is quite costly and unique).People started to hire this vehicle to reach safely by driving it by themselves. Every part of journey from town to kingdom or from kingdom to town has some cost associated with it which is given by an array A[] elements. Array A[] has n elements, where A(i) represents the cost i^{th} person has to pay if they travel alone in the vehicle. If two people travel in vehicle, the cost of travel will be the maximum of cost of two people.Calculate the minimum total cost so that all N people can reach Kingdom safely.

Input format

•The first line contains, T, denoting the number of test cases. Each test case contains 2 lines each. The first line has an integer N, denoting the number of persons. Next line contains N space separated distinct integers denoting the cost of i^{th} person.

Output format

For each test case, print the minimum cost required so that all people can reach kingdom.

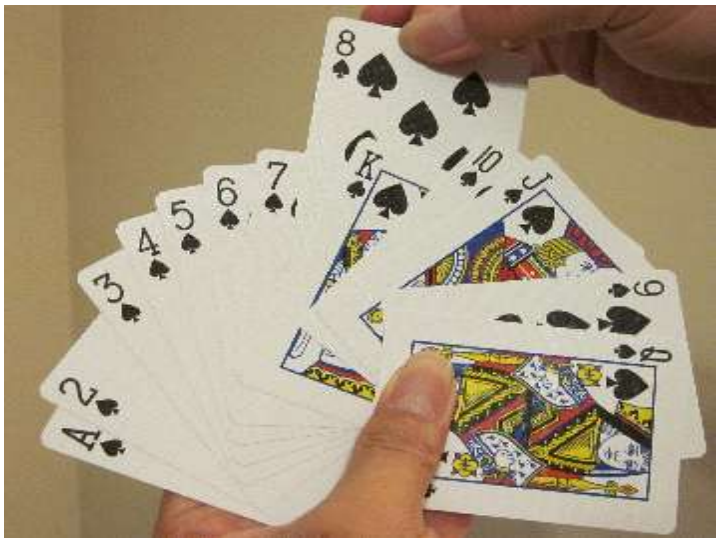
2.4. Insertion Sort

2.4.1. Analogy (Card Game)

Consider a situation where playing cards are lying on the floor in the arbitrary manner. In case we want these cards to be sorted, we can choose one of the cards and place that in hand. Every time we pick a card from pile, we can insert that at the right place in the hand. This way we will have sorted cards in the hand and a card arbitrarily chosen from lot will be inserted at the right place in the cards in hand.

A, 2, 3, 4, 5, 6, 7 | K, 10, J, 8, 9, Q

Red colored card set is sorted, Black Colored card set unsorted. Try inserting a card from unsorted set in the sorted set.



A, 2, 3, 4, 5, 6, 7, 8 | K, 10, J, 9, Q

Red colored card set is sorted, Black Colored card set unsorted. Try inserting a card from unsorted set in the sorted set.

A, 2, 3, 4, 5, 6, 7, 8, 9 | K, 10, J, Q

Red colored card set is sorted, Black Colored card set unsorted. Try inserting a card from unsorted set in the sorted set.

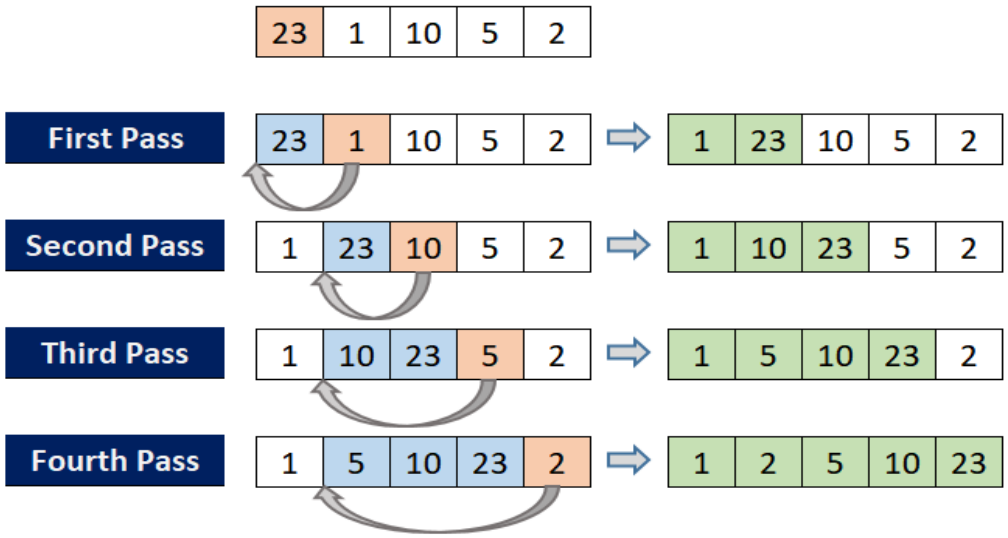
The process like this continues and finally we can have all the cards sorted in the hand.

2.4.2. Concept of Insertion Sort:

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists.

It considers two halves in the same array: Sorted and unsorted. To start with only one element is taken in the sorted list (first element) and N-1 elements in the unsorted list (2^{nd} to N^{th} element). It works by taking elements from the unsorted list one by one and inserting them in their correct position into the sorted list. See the example below wherein we have taken a small set of numbers containing 23, 1, 10, 5, 2. In the first pass we will consider the sorted parts to contain only 23 and unsorted part containing 1, 10, 5, 2. A number (1) is picked from unsorted part and inserted in sorted part at the right place. Now sorted part contains 2 elements. A number from unsorted part (10) is picked and inserted in the sorted part. And so on so forth until all the elements from the

unsorted part have been picked and inserted in the sorted part. The diagram below shows the step by step process:



In last (N-1)st pass

```
J = N;
Key = A[N]
WHILE J >= 1 AND key <A[J-1] DO
        A[j] =  A[j-1];
        J=J-1
A[J+1]=Key;
```

Total N - 1 passes of similar nature

ALGORITHM Insertion Sort (A[ ], N)

BEGIN:

```
FOR i=2 TO N DO
        J = i;
        Key = A[N]
        WHILE J >= 1 AND key <A[J-1] DO
                A[j] =  A[j-1];
                J=J-1
        A[J+1]=Key;
```

END;

Worst Case (N-1 iterations, all comparisionsin each iteration)

1 comparison in first iteration

2 comparisons in second iteration

3 comparisons in third iteration

…

N-1comparisons in $(N-1)^{st}$ iteration

Total = 1+2+3+ … +(N-1)

= N(N-1)/2

= $N^2/2 – N/2 = O(N^2)$

Best Case (All iteration takes place, only one comparisons per iteration, if numbers are Sorted)

Total comparisons = 1+1+1+ … (N - 1) times

= (N-1) = Ω (N)

Insertion sort is-Comparison based sorting algorithm, Stable sorting algorithm, In place sorting algorithm, Uses incremental approach

Recursive Approach

ALGORITHM Insertion Sort(A[ ], N)

BEGIN:

```
IF N<=1 THEN
        RETURN;
Insertion Sort(A, N-1)
Key = A [N-1]
J=N
WHILE J >= 1 AND key <A[J-1] DO
        IF A [j] > Key THEN
                A [j+1] = A [j]
                J = J -1
        A [j+1] = Key
END;
```

2.4.3. Scenarios where Insertion Sort can be applied

Scenario–1

Insertion sort is best suited for online algorithms because it sorts the elements as it receives it (if numbers of elements are known). If not known then it is required to store those elements into link list.

Insertion sort performs better than even Quick sort where we sort small amount of data.

Scenario–2

When numbers of elements are almost sorted

Let us suppose that out of N elements N-1 elements sorted and only 1 element is out of place. The Insertion sort takes O(N) time in this scenario.

2.4.4. Problems

Problem–1

Sort the given elements and compute time complexity needed. Suggest which sorting algorithm will take minimum time to sort these elements.

Arr[]={43,33,64,54,86,75}

Solution-

From above it is clear that alternate elements are not sorted.

Outer loop runs n-1 times (Number of iterations)

Inner loop runs for 1 time or 2 time.

It means total time taken O(n)

Problem–2

Sort the given elements and compute time complexity needed. Suggest which sorting algorithm will take minimum time to sort these elements.

Arr[]={5,5,5,5,5,5,5,5,5,5}

Solution-

From above it is clear that all the elements are same.

Outer loop runs n-1 times

Inner loop runs for one time per iteration.

It means total time taken O(n)

Inversion in insertion sort:

Inversion is defined as

A pair of elements (arr[i], arr[j]) is said to be inversion if arr[i]>arr[j] for i<j

If array sorted in ascending order – 0 inversion

If array sorted in descending order – n (n+1)/2 inversion

Problem–3

If the number of elements is n and it is given that number of inversion is O(n) then how much time insertion sort takes to sort the elements.

Solution-

Control goes in inner loop only n times so insertion sort takes O(n) times.

Binary Insertion Sort

Use binary search logic in order to find the position where element to be inserted. It takes O(logk) times in place of k times.

It is used where cost of comparison is more than cost of swapping. Consider the scenario of sorting the names.

ALGORITHM insertionSort(ARR, N)

BEGIN:

```
FOR i=2 to N DO
        j=i-1
        temp = ARR[i]
        loc = BinarySearch (ARR, temp, 0, j)
        WHILE j>=loc DO
                ARR[j+1] =ARR[j]
                J = J – 1
        ARR[j+1]=temp
```

END;

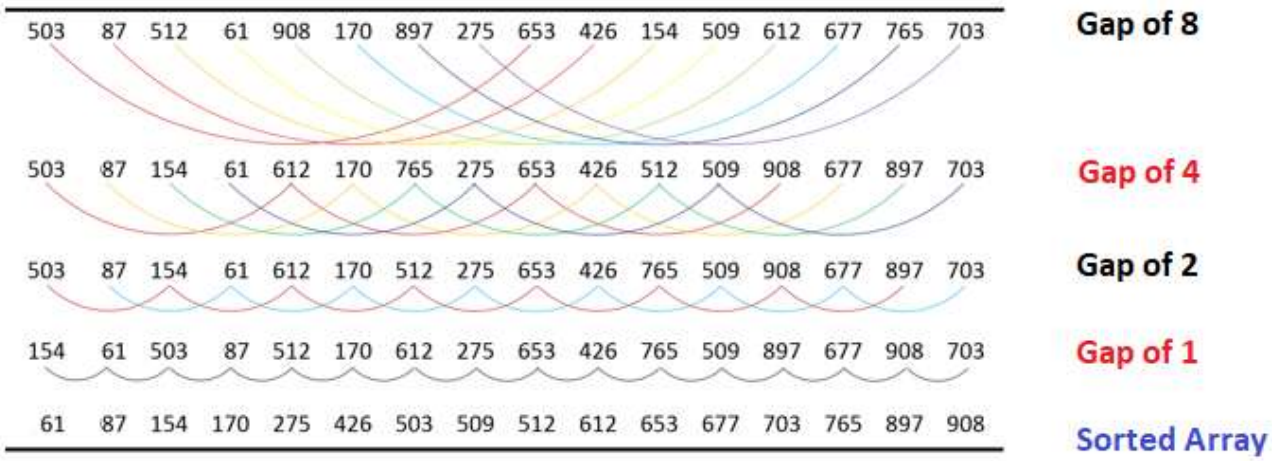
2.4.5. Variation of Insertion Sort (Shell Sort)

Shell sort is a variation of insertion sort. It improves complexity of insertion by dividing it into number of parts and then apply insertion sort.

It works on two facts about insertion sort-

1-Works better for less number of elements.

2-Elements are less distant towards their final positions.



Shell Sort Algorithm (gapped insertion sort)

ALGORITHM Shell Sort(ARR, N)

BEGIN:

```
FOR i=N/2 TO 1 STEP/2 DO
        FOR j=i TO j<=N DO
                temp=ARR[j]
                FOR k = j TO k > = I STEP -1 DO
                        IF ARR[k-i] > temp THEN
                                ARR[k]=ARR[k-i]
                ARR[k]=temp
```

END;

Suggested Gaps to be taken for Shell Short

1-Hibbard Gap Sequence

There is a comparison between Hibard Shell Sort($n^{1.5}$) and Donald Shell Sort takes O(n^2). It is most widely gap sequence suggested by Thomas Hibard which is (2^{k-1} ……. 31 .. 15 .. 7 .. 3 .. 1) which gives time complexity O(n √n)

2-Donald Knuth Gap Sequence- (1, 4, 13, …)

3- A gap sequence like (64, 32, 16, …) is not good because it increases time complexity.

2.4.6. Competitive Coding– Problem Statement-

There is a given Binary Search Tree in which any two of the nodes of the Binary Search Tree are exchanged. Your task is to fix these nodes of the binary search tree.

Note: The BST will not have duplicates.

Examples:

Input Tree:

```
      15
     / \
    12   10
   / \
  4   35
```

In the above tree, nodes 35 and 10 must be swapped to fix the tree.

Following is the output tree

```
      15
     / \
    12   35
   / \
  4   10
```

Input format: Given nodes of binary search tree

Output format: Print the nodes in inorder

1-Identify problem statement: Read the story and try to convert it into technical form. For this problem reframes as- Given a BST in which two nodes are interchanged so it's lost the property of BST.

Design Logic:

1-ttraverse the node in inorder and store it in an array.

2-Sort it using insertion sort and store it into another array.

3-Compare both arrays and find out exchanged nodes and fix these node.

Implementation-

ALGORITHM bst(ROOT)

BEGIN:

```
ARRAY A[]
Inorder(ROOT,A)
Copy(B,A)
Insertion_sort(A,N)
FOR i=0 TO N DO
        IF A[i] !=  B[i] THEN
                Find(ROOT,A[i],B[i])
                BREAK
        RETURN ROOT
        Find(ROOT→left,x,y)
        IF !root THEN
                RETURN
        Find(ROOT->left, x, y)
        IF ROOT ->data == x THEN
                ROOT ->data = y
        ELSE IF ROOT ->data == y
                ROOT ->data = x
        Find(ROOT ->right, x, y)
```

END;

Complexity-O(n)

2.4.7. Unsolved Coding problems on Insertion Sort:

1. Competitive Coding Problem-(Hackerrank)

Complete the **insert()** function which is used to implement Insertion Sort.

Example 1:

Input:

N = 5

arr[] = { 1,3,2,4,5}

Output: 1 2 3 4 5

Example 2:

Input:

N = 7

arr[] = {7,6,5,4,3,2,1}

Output: 7 6 5 4 3 2 1

Task: Read no input and don't print anything. Your task is to complete the function **insert()** and **insertionSort()** where **insert()** takes the array, it's size and an index i and **insertion_Sort()** uses insert function to sort the array in ascending order using insertion sort algorithm.

Expected Time Complexity: O(Nlog n).

Expected Auxiliary Space: O(1).

Constraints:

1 <= N <= 1000

1 <= arr[i] <= 1000

2. Competitive Coding Problem Statement (code chef)

Given an array sort it using insertion sort and binary search in it to find the position to place the element. Additionally you need to count the number of times recursive calls done by binary search function to find the position.

Input:

- First line will contain NN, size of the array.
- Next Line contains n array elements a[i]a[i].

Output:

For each test case, output sorted array and binary search calls counter modulo 1000000007.

Constraints

- 1≤N≤100001≤N≤10000
- 0≤M≤1090≤M≤109

Sample Input:

5

5 4 3 2 1

Sample Output:

Sorted array: 1 2 3 4 5

7

3. Competitive Coding Problem Statement (code chef)

Say that a string is ***binary*** if it consists solely of the symbols 00 and 11 (the empty string is binary too). For binary string ss let's define two functions:

- The function rev(s) reverses the string ss. For example, rev(010111)=111010, and rev(01)=10.
- The string flip(s) changes each character in ss from 00 to 11 or from 11 to 00. For example, flip(010111)=101000 and flip(11)=00.

If s=rev(s) then we say that ss is a ***palindrome***. If s=rev(flip(s)) then we say that ss is an ***antipalindrome***.

Given a binary string $s= s_1s_2…s_{|s|}$, divide it into a palindrome and an antipalindrome. Formally, you should find two sequences $i_1,i_2,…,i_k$, and $j_1,j_2,…,j_m$, such that:

- k, m≥0
- $|s|=k+m$
- All indices $i_1,i_2,…,i_k$, $j_1,j_2,…,j_m$ are distinct integers satisfying $1≤i_x,j_x≤|s|$.
- $i_1<i_2<…<i_k$ and $j_1<j_2<…<j_m$
- The string $s_{i1}s_{i2}…s_{ik}$ is a palindrome.
- The string $s_{j1}s_{j2}…s_{jm}$ is an antipalindrome.

Input:

The first line contains a single integer, tt - the number of test cases. The next tt lines describe test cases.

The only line for each test case contains binary string ss.

Output:

In the first line for each test case, print two integers k and m.

In the second line for each test case, print k integers $i_1,i_2,…,i_k$.

In the third line for each test case, print m integers $j_1,j_2,…,j_m$.

All required conditions should be satisfied.

It can be shown that an answer always exists. If there exists multiple answers you can print any.

Constraints

- 1 ≤ t ≤ 105
- 1 ≤ |s|≤ 100000
- the sum of lengths of all strings does not exceed 300000

Subtasks

Subtask #1:

- t≤1000
- |s|≤10

Subtask #2: original constraints

Sample Input:

4

0

10111001011

1100100001

11000111

Sample Output:

1 0

1

5 6

1 4 6 8 11

2 3 5 7 9 10

6 4

1 3 4 7 8 10

2 5 6 9

6 2

1 2 4 5 7 8

3 6

Explanation:

In the first test case, the string 00 is a palindrome and the empty string is an antipalindrome. In the second test case, we can use indices [1,4,6,8,11] to create the palindrome $s_1s_4s_6s_8s_{11}$=11011 and indices [2,3,5,7,9,10] to create the antipalindrome $s_2s_3s_5s_7s_9s_{10}$=011001.

2.4.8. Objective Type Questions:

1	Which of the following searching techniques is used in insertion sort algorithm?
A	Linear Search
B	Binary Search
C	Hashing
D	None of these
AN	**A**
DL	**E**

2	The best case complexity in insertion Sort algorithm is ?
A	$O(n^2)$
B	O(nlog n)
C	O(n)
D	None of these
AN	**C**
DL	**E**

3	In the best case scenario while implementing Insertion sort algorithm, How many comparisons occur in inner loop?
A	n-1
B	n/2
C	1
D	None of these
AN	**C**

4	In the worst case scenario the elements in the Insertion Sort algorithm will be?
A	Strictly Increasing order
B	Strictly Decreasing Order
C	ZigZag order
D	None of these
AN	**B**
DL	**M**

5	In the best case algorithm the elements in Insertion Sort algorithm will be?
A	Strictly Increasing order
B	Strictly Decreasing Order
C	ZigZag order
D	None of these5
AN	**A**
DL	**E**

6	What would be the space complexity and auxiliary space complexity when we implement Insertion Sort Algorithm?
A	O(n), O(1)
B	O(1), O(n)
C	O(1), O(1)
D	None of these
AN	**C**

7	Which of the following properties is satisfied by Insertion Sort algorithm?
A	It is in place algorithm
B	It has best case complexity of O(n)
C	Overall complexity is n log n
D	Both A and B
AN	**D**
DL	**M**

8	Consider the following array 34 15 29 8. How many number of comparisons required in the third pass to sort the array using bubble sort?
A	1
B	2
C	0
D	3
AN	**1**
DL	**M**

9	Consider the following array 10 9 11 6 15 2. What would be the worst case complexity of the sorting operation on the array using bubble sort?
A	O(n^2)
B	O(n)
C	O(1)
D	O(log n)
AN	**A**
DL	**E**

10	Consider the following array 10 9 11 6 15 2. How the array looks like after the second pass if applies bubble sort on the array.
A	9 10 6 11 2 15
B	9 6 10 2 11 15
C	6 9 2 10 11 15
D	9 6 10 2 11 15
AN	**B**
DL	**M**

11	Consider the following array 10 9 11 6 15 2. How the array looks like after the third pass if applies bubble sort on the array.
A	9 10 6 11 2 15
B	9 6 10 2 11 15
C	6 9 2 10 11 15
D	9 6 10 2 11 15
AN	**C**
DL	**M**

Scenario	Consider the following code given below, answer the following questions from 12 to 15
	Insertion_sort(A)
	{
	for j = 2 to A.length
	key = A[j]
	// insert A[j] into sorted sequence of A[i..... j-1]
	i = j-1;
	while(i >0 and A[i] > key)
	A[i+1] = A[i];
	i = i -1;
	A[i+1] = key;
	}

12	How many numbers of swapping is required in the worst case scenario in the above algorithm? Assuming there are n elements?
A	1
B	n^2
C	n-1
D	None of these
AN	**B**
DL	**M**

13	How many numbers of comparisons are required in the worst case scenario in the above algorithm? Assuming there are n elements?
A	n^2
B	0
C	1
D	n-1
AN	**A**
DL	**M**

14	How many numbers of swapping is required in the best case scenario in the above algorithm? Assuming there are n elements?
A	n^2
B	0
C	1
D	n-1
AN	**B**
DL	**M**

15	How many numbers of comparisons are required in the best case scenario in the above algorithm? Assuming there are n elements?
A	n^2
B	0
C	1
D	n-1
AN	**D**
DL	**M**

16	If we use Binary Search instead of linear search while implementing Insertion sort ,then total number of swapping involved will be?
A	O(log n)
B	O(n)
C	$O(n^2)$
D	None of these
AN	**C**
DL	**M**

17	If we use Binary Search instead of linear search while implementing Insertion sort ,then total number of comparisons and time complexity will be?
A	O(log n) and O(nlog n)
B	O(n) and O(nlog n)
C	O(log n) and $O(n^2)$
D	None of these
AN	**A**
DL	**M**

18	If we use a Doubly Linked list instead of linear search while implementing Insertion sort , then total number of comparisons involved will be?
A	O(log n)
B	O(n)
C	$O(n^2)$
D	None of these
AN	**C**
DL	**M**

19	If we use a Doubly Linked list instead of linear search while implementing Insertion sort, then total number of swapping involved will be?
A	O(log n)
B	O(n)
C	$O(n^2)$
D	None of these
AN	**B**
DL	**M**

20	Which of the following is not an example of a non comparison algorithm?
A	Counting Sort
B	Radix Sort
C	Bucket Sort
D	Bubble sort
AN	**D**
DL	**M**

21	Which of the following algorithms will in its typical implementation give best performance when applied on an array that is sorted or almost sorted? Assume there to be one or two misplaces?
A	Counting Sort
B	Insertion Sort
C	Bucket Sort
D	Quick Sort
AN	**B**
DL	**M**

22	Bubble sort can be categorized into which of the following?
A	Greedy Algorithm
B	Dynamic Programming
C	Brute force technique
D	Divide and conquer
AN	C
DL	**E**

2.5. Heap Sort

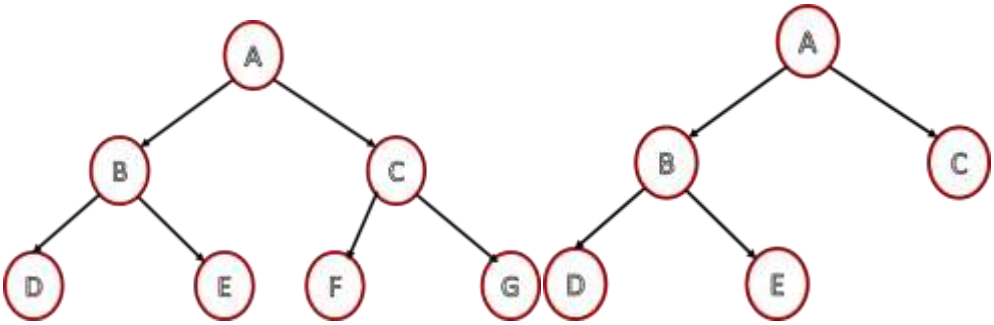
2.5.1. Heap Analogy

A Knock out tournament is organized where first round is the quarter final in which there are 8 teams CSK, Mumbai Indians, Delhi Capitals , Kolkata Knight Riders, Punjab Kings, Rajasthan Royals, RCB, Sunrisers Hyderabad participated.

After the first round four teams will reach in semi final round where two semi finals will be played.
From the semi Final round, two team will reach in final. Now the winner of the final will take
This analogy resemble the concept of Heap(Max-Heap).

2.5.2. Pre requisite:- Complete Binary Tree

It is a type of binary tree in which all levels are completely filled except possibly the last level .
Also last level might or might not be filled completely. If last level is not full then all the nodes should be filled from the left.

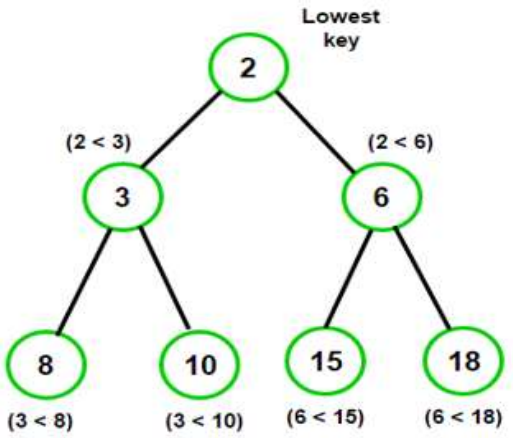


Note:- In the above diagram, nodes are filling from left to right and level via level.

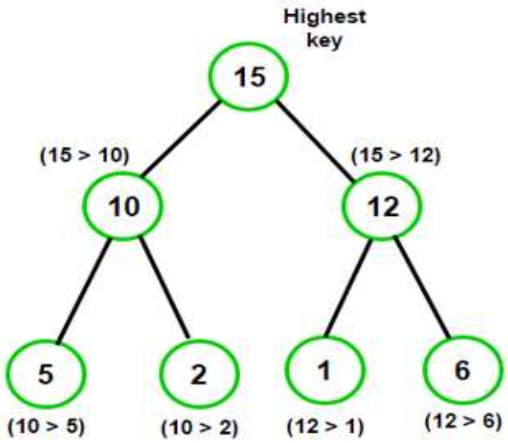
Application: To implement Heap Data structure.

2.5.3.Heap:

A Binary heap is a complete Binary Tree which makes it suitable to be implemented using array.
A Binary Heap is categorized into either Min-Heap or Max-Heap.
In a Min Binary Heap, the value at the root node must be smallest among all the values present in Binary Heap. This property of Min-Heap must be true repeatedly for all nodes in Binary Tree.
In a Max Binary Heap the value at the root node must be largest among all the values present in Binary Heap. This property of Max-Heap must be true repeatedly for all nodes in Binary Tree.
As heap is a complete binary tree therefore height of tree having N nodes will always O(log n).



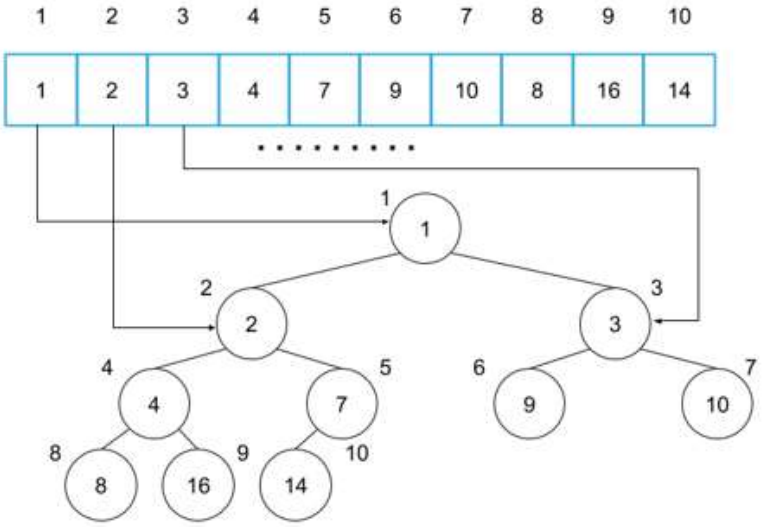
Min Heap
(Parent key is less than or equal to (≤) the child key)



Max Heap
(Parent key is greater than or equal to (≥) the child key)

2.5.3.1 Implementation of Heap:

If Heap can be implemented using Array. Assume that Array indexes start from 1.



You can access a parent node or a child nodes in the array with indices below.

A root node | $i = 1$, the first item of the array

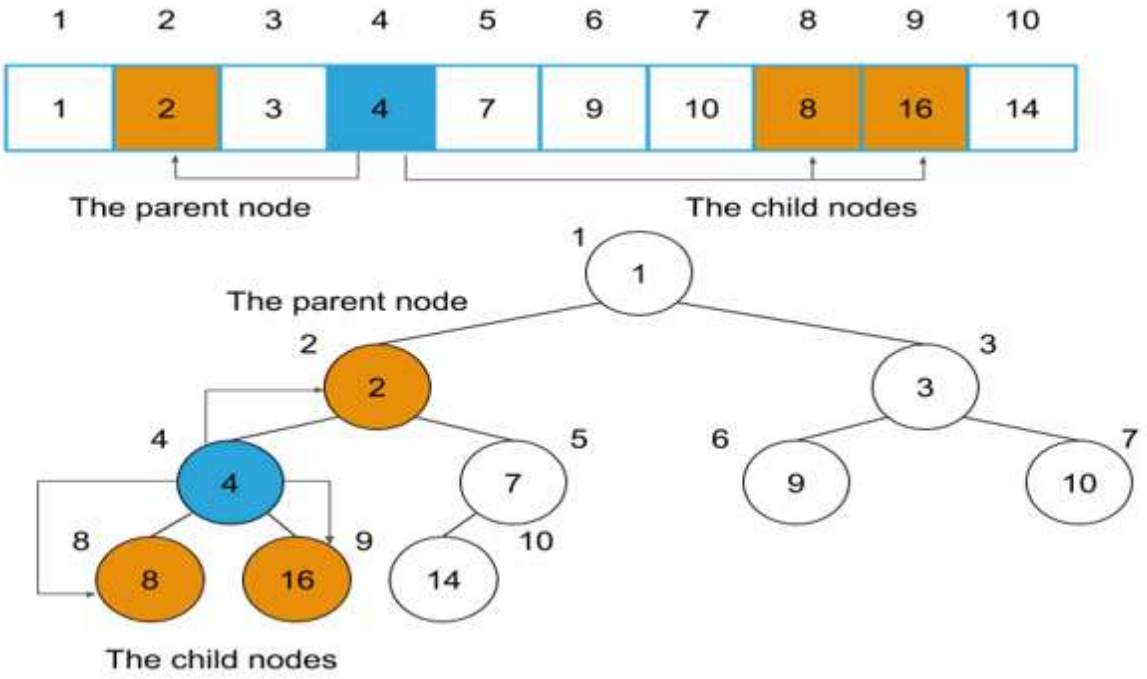
A left child node | left(i) = 2*i

A right child node | right(i)=2*i+1

A parent node | parent(i) = i / 2

When you look at the node of index 4, the relation of nodes in the tree corresponds to the indices of the array below. If i = 4, Left Child will be at 2 * 4 that is 8^{th} position and Right Child will be at (2*4 + 1) 9^{th} position.

Also if the index of left child is 4 then index of its parent will be 4/2 =2.



2.5.3.2 Heap Operations

1. Construct max Heap:

Following two operations are used to construct a heap from **an arbitrary array**:

a) **MaxHeapify**–In a given complete binary tree if a node at index k does not fulfill max-heap property while its left and right subtree are max heap, MaxHeapify arrange node k and all its subtree to satisfy maxheap property.

b) **BuildMaxHeap**–This method builds a Heap from the given array. So BuildMaxHeap use MaxHeapify function to build a heap from the

MaxHeapify(A,i,N) is a subroutine.

When it is called, two subtrees rooted at Left(i) and Right(i) are max-heaps, but A[i] may not satisfy the max-heap property.

MaxHeapify(A,i,N) makes the subtree rooted at A[i] become a max-heap by letting A[i] “float down”.

Example:

a) Given an arrays as below

b) First construct a complete binary tree from the array

c) Start from the first index of non-leaf node whose index is given by n/2 where n is the size of an array.

d) Set current element having index k as largest.

e) The left child index will be 2*k and the right child index will be 2*k + 1 (when array index starts from 1).

If left Child value is greater than current Element (i.e. element at kth index), set leftChildIndex as largest index.

If rightChild value is greater than element at largest index, set rightChildIndex as largest index.

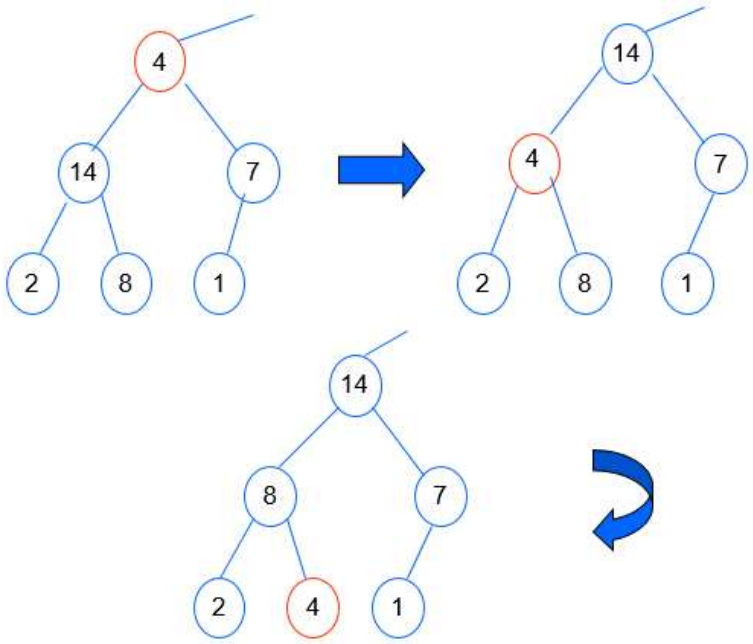
f) Exchange largest with currentElement(i.e. element at kth index).

g) Repeat the steps from (c) to (f) until the subtrees are also get heapify.

ALGORITHM MaxHeapify(A[ ], k, N)
BEGIN:

```
L = Left(k)                          //Left(k)=2*k
R = Right(k)                         //Right(k)=2*k+1
//Initialize Largest index
IF L ≤ heap-size(A) and A[L] > A[ k ] THEN
    largest = L
ELSE
    largest = k
IF R ≤ heap-size(A) and A[ R ] > A[ largest ] THEN
    largest ← R
IF largest != k THEN
    Exchange A[ k ] with A[ largest ]
MaxHeapify (A, largest,N)
```

END;



Left child of orange marked node is 2 and right child is 8.

Steps:

1) Is 4 > 2 , greater is 4 stored in temp.
2) Is 4 > 8, greater is 8, now largest is 8. Temp is having index of 8.
3) Swap the element at index of marked node and temp node.

ALGORITHM BuildMaxHeap(A[ ], N)
BEGIN:

FOR i = N/2 TO 1 STEP – 1 DO

MaxHeapify(A, i, N)

END;

Analysis

As we know that time complexity of Heapify operation depends on the height of the tree i.e. H and H should be log N when there are N nodes in the heap.

The height 'h' increases as we move upwards along the tree. Build-Heap runs a loop from the index of the last internal node N/2 with height=1, to the index of root(1) with height = lg(n). Hence, Heapify takes different time for each node, which is θ(h).

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

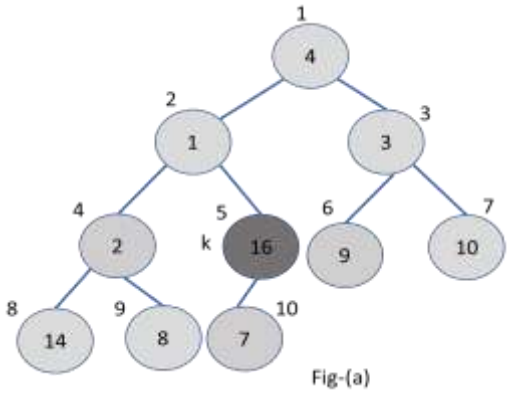


Fig-(a)

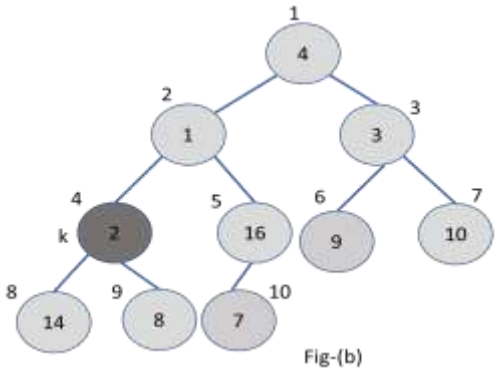


Fig-(b)

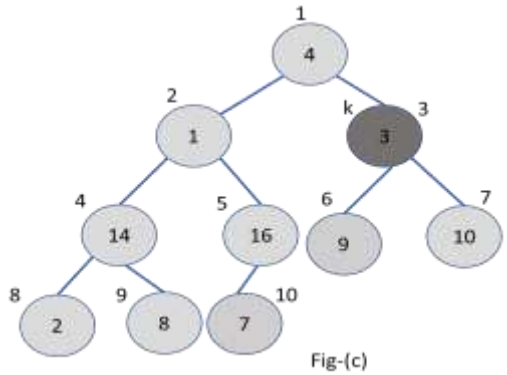


Fig-(c)

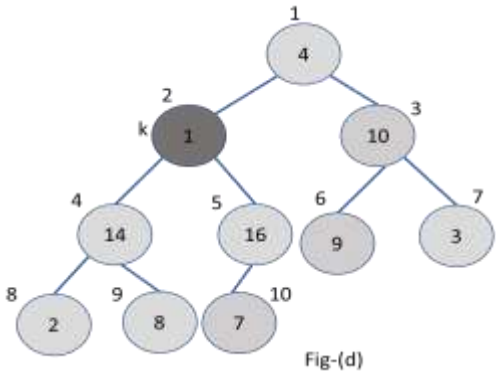


Fig-(d)

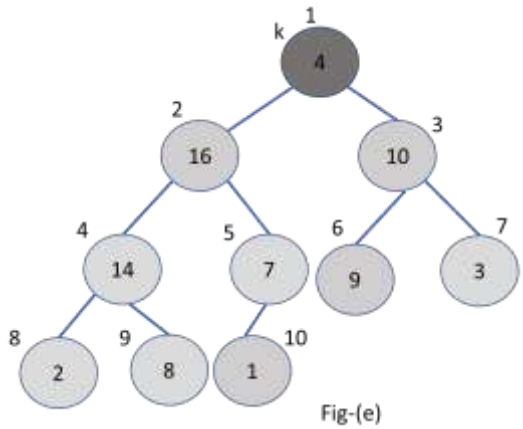


Fig-(e)

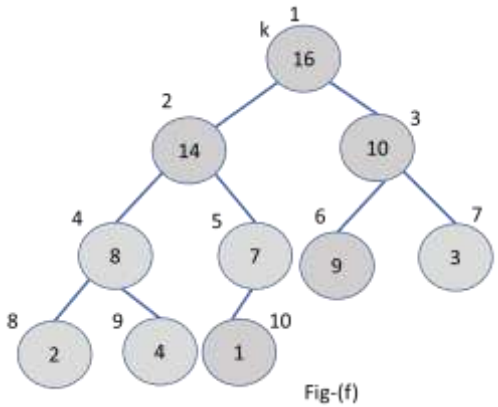


Fig-(f)

	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1

Finally, we have got the max-heap in fig.(f).

2. Insert operation:

To insert an element in Max Heap.

Following are the steps to to insert an element in Max Heap.

a) First update the size of the tree by adding 1 in the given size.
b) Then insert the new element at the end of the tree.
c) Now perform Heapify operation to place the new element at its correct position in the tree and make the tree either as max heap or min heap.

Example: To show how insert operation works.

	1	2	3	4	5	6	7	8	9	10	11
A	16	14	10	8	7	9	3	2	4	1	20

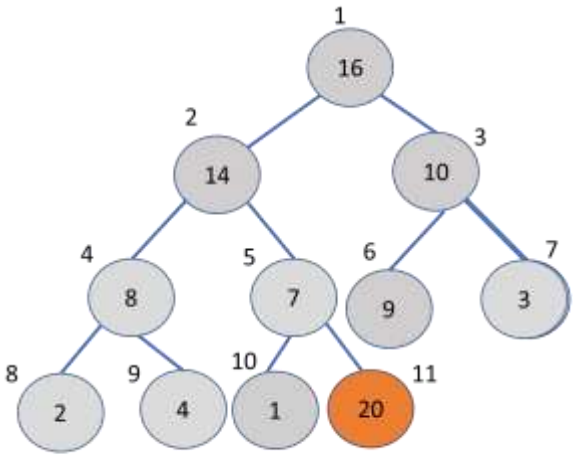


Fig-(g)

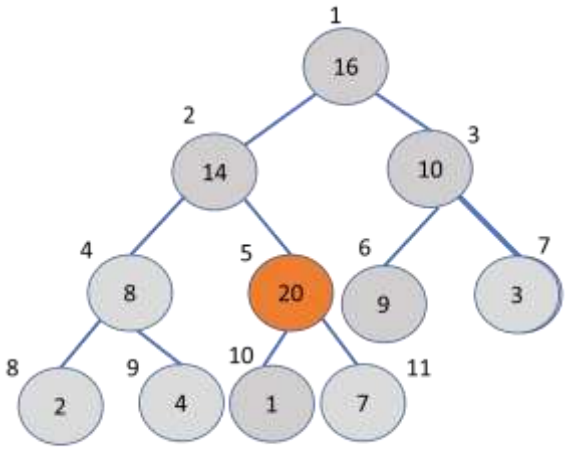


Fig-(h)

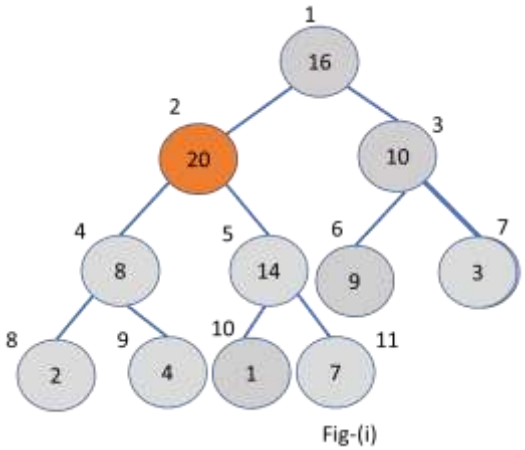


Fig-(i)

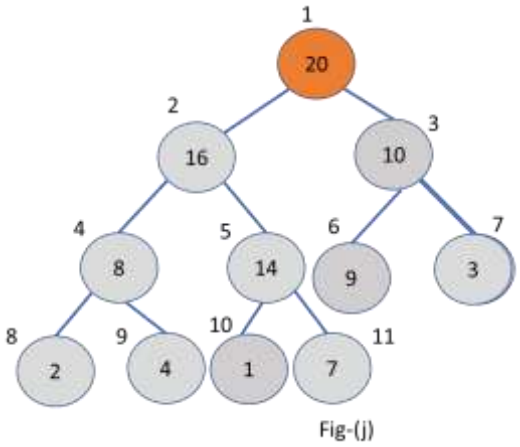


Fig-(j)

	1	2	3	4	5	6	7	8	9	10	11
A	20	16	10	8	14	9	3	2	4	1	7

This is the array representation of the given max-heap.

Algorithm: InsertNode(A[ ], N, item)

BEGIN:

```
N = N + 1;
A[ N ] = item;
ReHeapifyUp( A[ ], N, k)
```

END;

Algorithm: ReHeapifyUp(A[ ], N, k)

BEGIN:

```
parentindex = k/ 2;
IF parentindex> 0 THEN
        IF A[k]>A[parentindex] THEN
                Exchnage(A[k], A[parentindex])
                ReHeapifyUp(A, N, parentindex)
```

END;

Analysis:

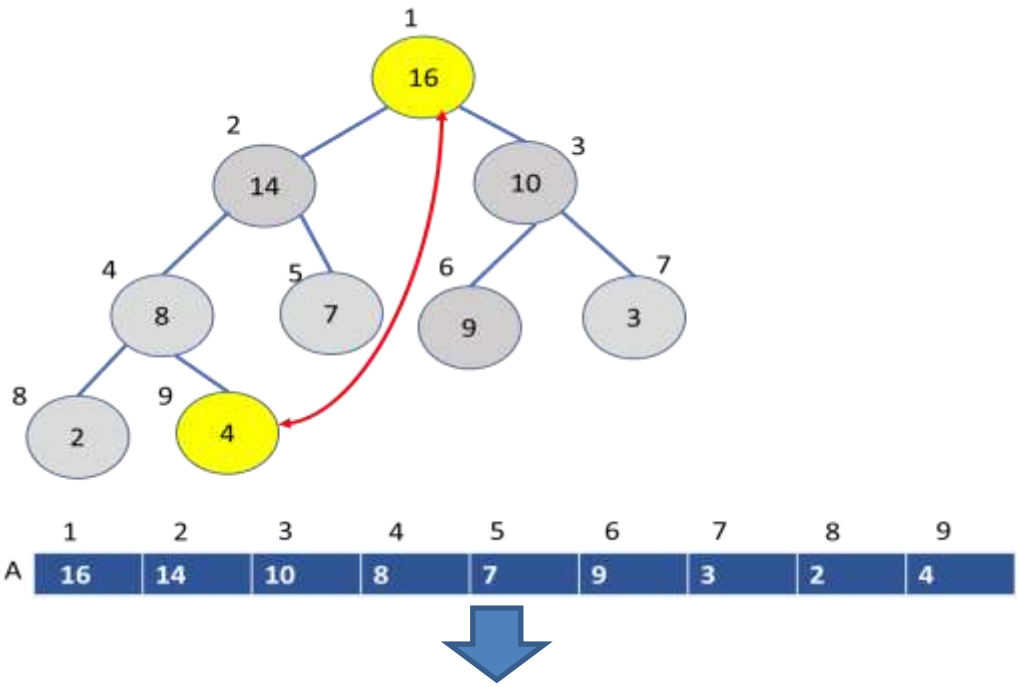
Time Complexity: θ(log n)

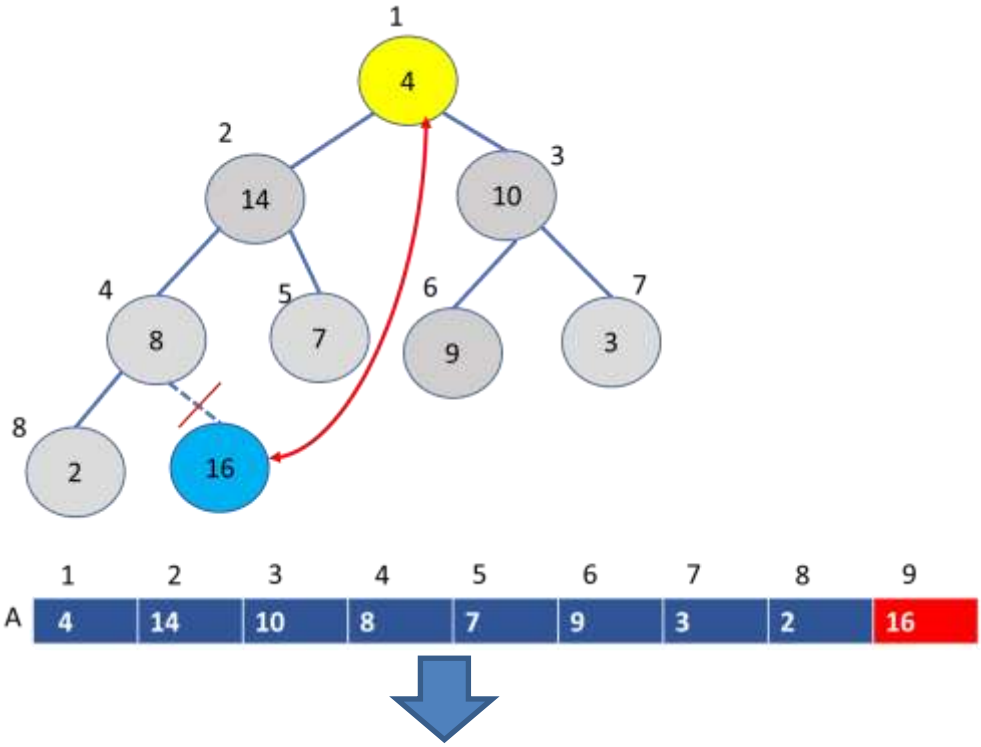
3 Delete Operation:

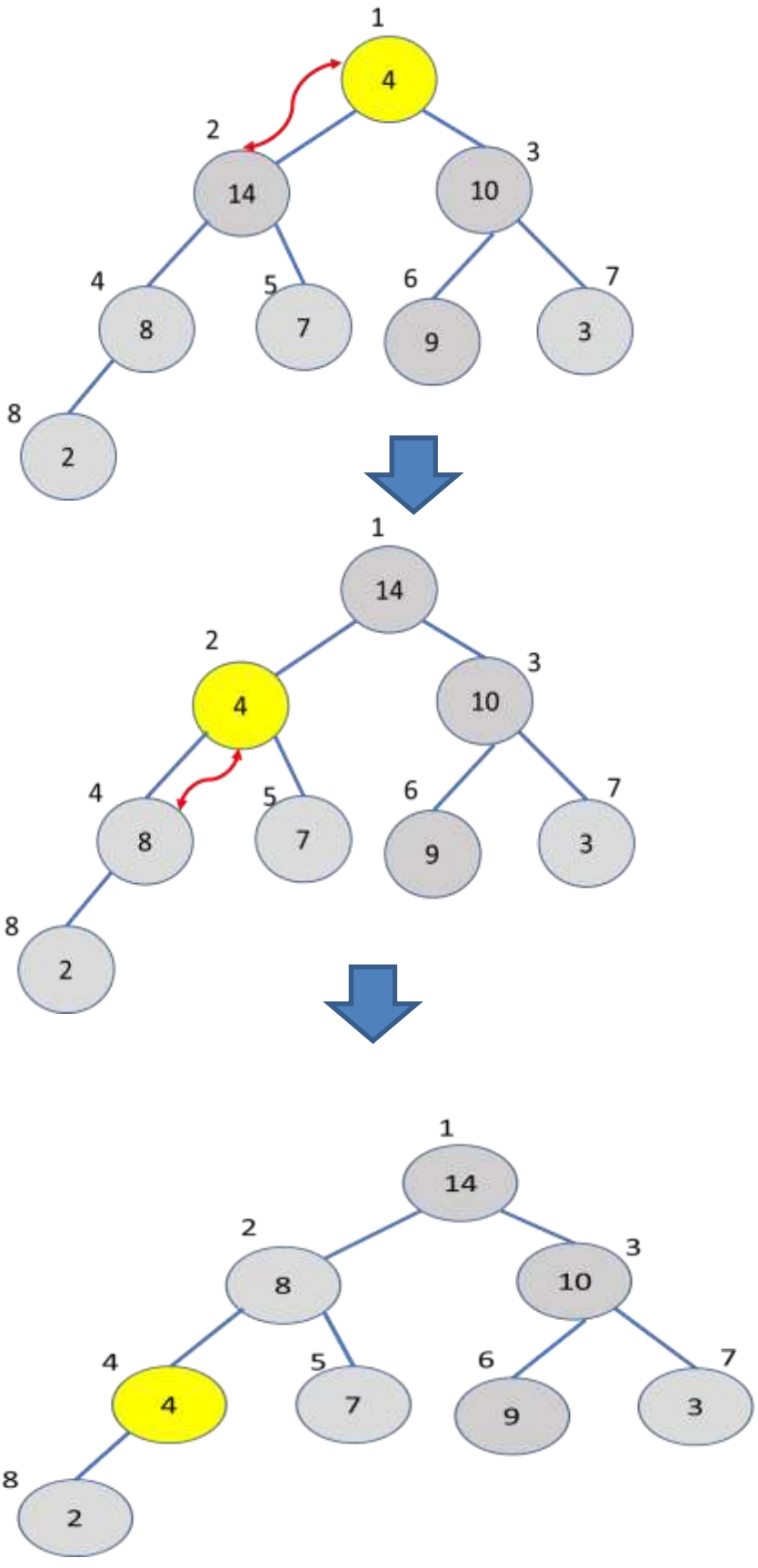
Method-1 To delete root element from Max Heap.

Following are the steps to delete an element from Max Heap.

a) First exchange theroot element with the last element in the heap..
b) Then remove the last element by decreasing the size of the tree by 1.
c) Now perform Heapify operation to make the tree either as max heap or min heap.


1
16
2
14
3
10
4
8
5
7
6
9
7
3
8
2
9
4
1 2 3 4 5 6 7 8 9
A
16 14 10 8 7 9 3 2 4


1
4
2
14
3
10
4
8
5
7
6
9
7
3
8
2
16
1 2 3 4 5 6 7 8 9
A
4 14 10 8 7 9 3 2 16


1
4
2
14
3
10
4
8
5
7
6
9
7
3
8
2
1
14
2
4
3
10
4
8
5
7
6
9
7
3
8
2
1
14
2
8
3
10
4
4
5
7
6
9
7
3
8
2

Method-2 To delete an element at any position from Max Heap.

Following are the steps to delete an element from Max Heap.

a) First pick the element to be deleted.
b) Now exchange it with the last element.
c) Then remove this element by decreasing the size of the tree by 1.
d) Now perform Heapify operation to make the tree either as max heap or min heap.

Algorithm: DelHeap(A[ ], N)

BEGIN:

```
lastitem = A[N]          // Get the last element
A[1] = lastitem;         // Replace root with first element
N = N - 1;               // Decrease size of heap by 1
MaxHeapify(A,1,N);       // heapify the root node
```

END;

Analysis:

Time Complexity: θ(log n)

2.5. 4. Heap Sort

Heap Sort is a popular and efficient sorting algorithm to sort the elements.

Step1: In the Max-Heap largest item is always stored at the root node.

Step 2: Now exchange the root element with the last element in the heap.

Step 3: Decrese the size of the heap by 1.

Step 4: Now performMax-Heapify operation so that highest element should arrive at root. Repeat these steps until all the items are sorted.

Example:

Step1: In the Max-Heap largest item is always stored at the root node.

Step 2 Now exchange the root element with the last element.

Step1: Replace A[1] with A[last].

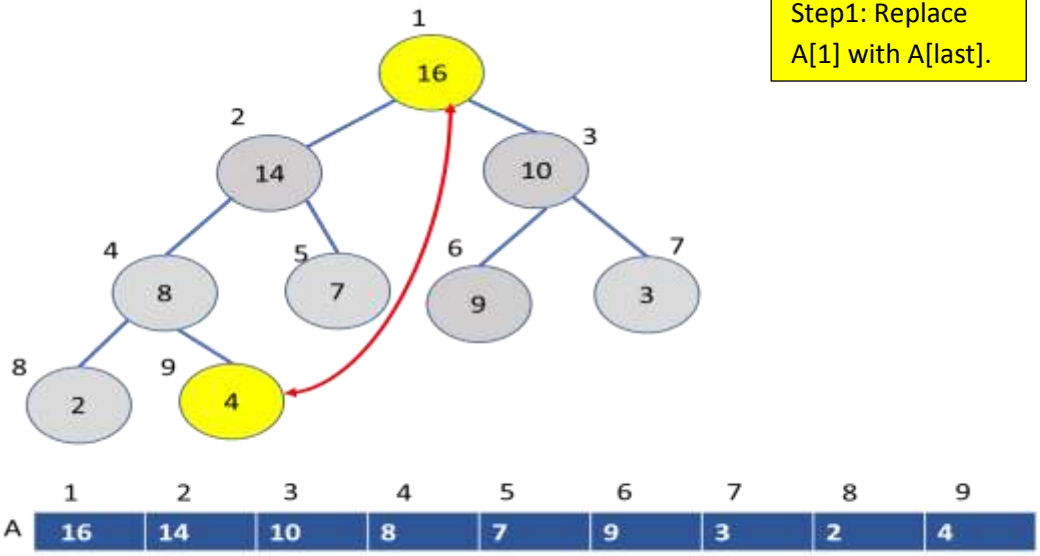


Fig-(k)

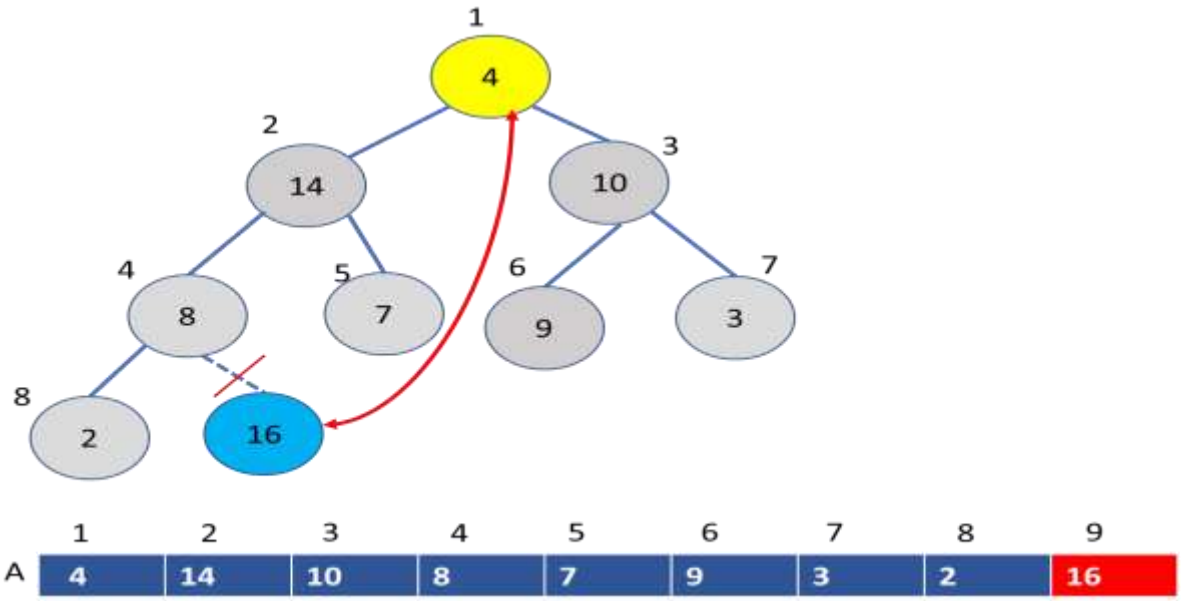


Fig-(l)

Step3 Decrese the size of the heap by 1 .

Step4 Now perform Max-Heapify operation so that highest element should arrive at root.

L = 2*k

R = L+1

Call Max-Heapify()

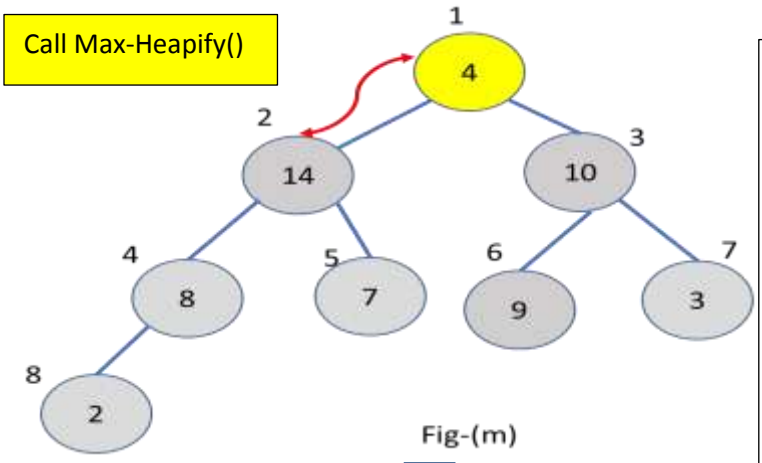


Fig-(m)

```
IF L ≤ heap-size(A) AND A[L] > A[ k ] THEN
        largest = L
else
        largest = k
IF R ≤ heap-size(A) AND A[ R ] > A[ largest ]
THEN
        largest = R
```

```
IF largest != k THEN
        Exchange A[ k ] with A[ largest ]
```

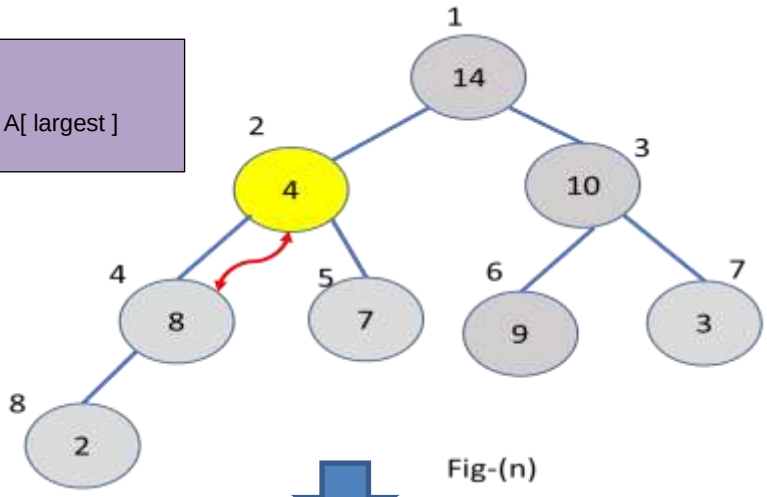


Fig-(n)

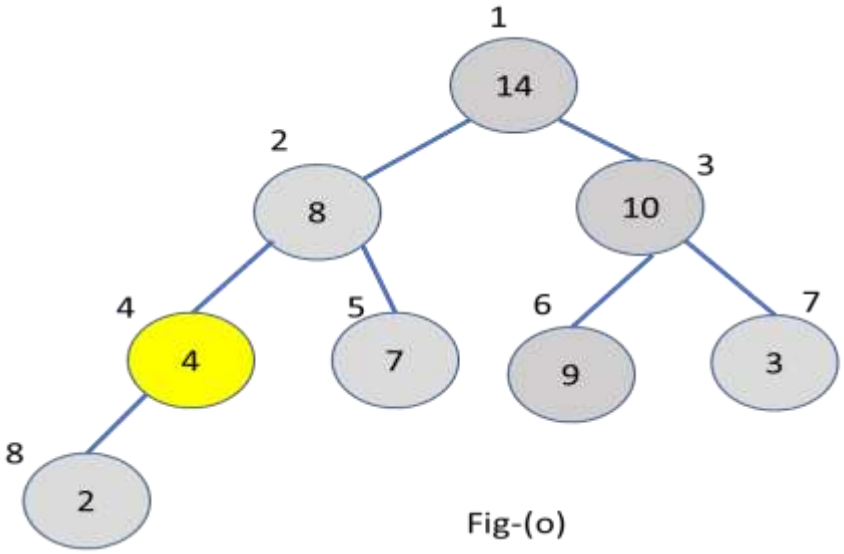


Fig-(o)

	1	2	3	4	5	6	7	8	9
A	14	8	10	4	7	9	3	2	16

Now this is the Max-heap after performing first deletion of root element.

Repeat these steps until all the items are sorted.

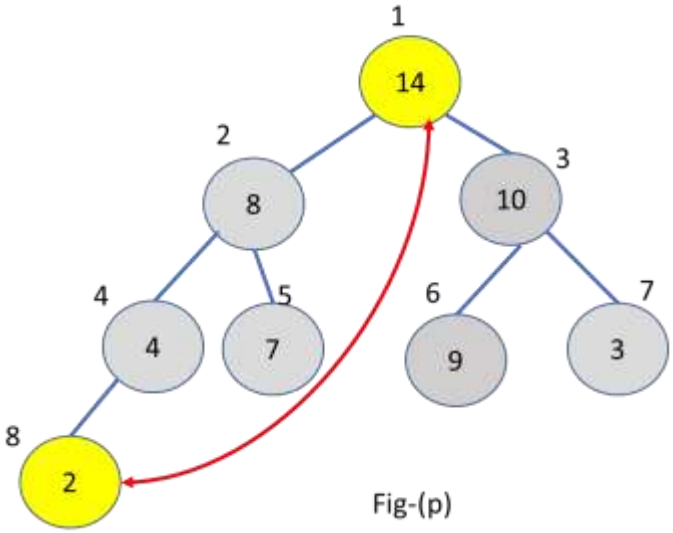


Fig-(p)

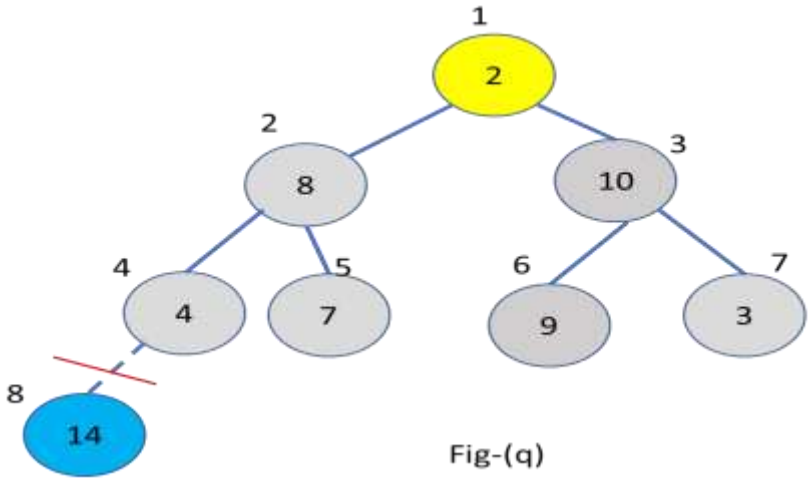


Fig-(q)

	1	2	3	4	5	6	7	8	9
A	2	8	10	4	7	9	3	14	16

Now, the last two fields of the array are sorted.

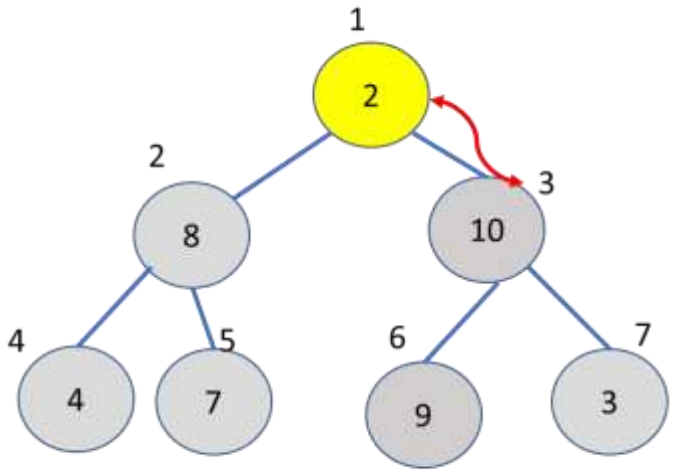


Fig-(q)

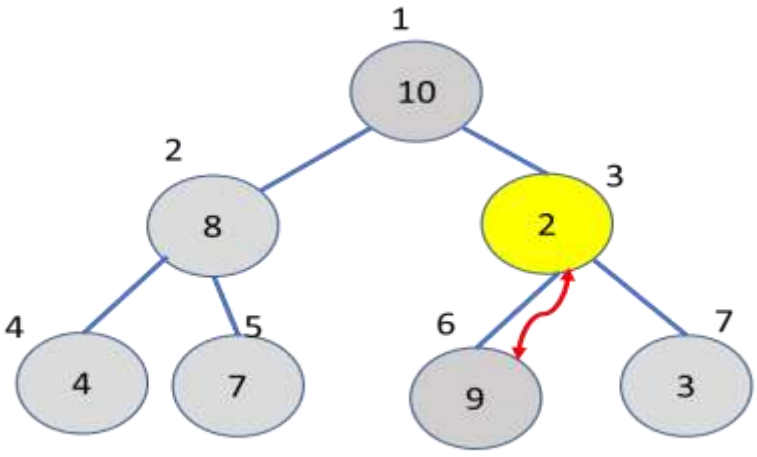


Fig-(r)

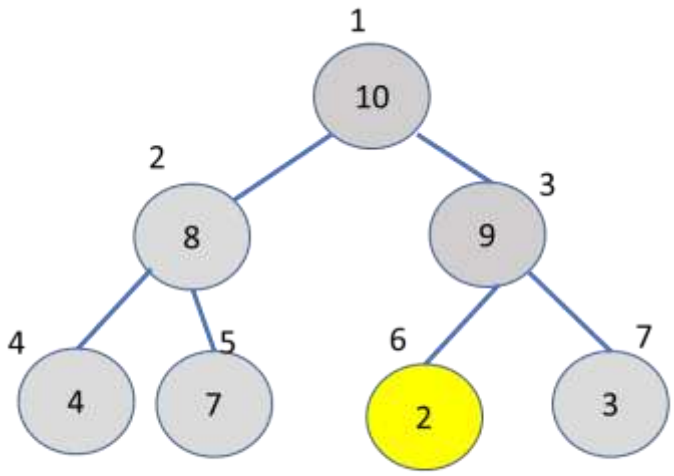
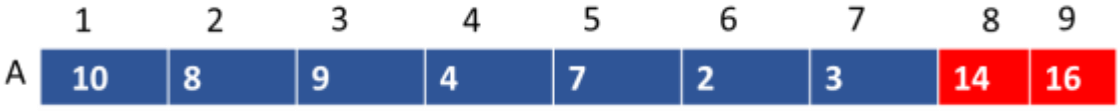


Fig-(s)



We repeat the process until there is only one element left in the tree:

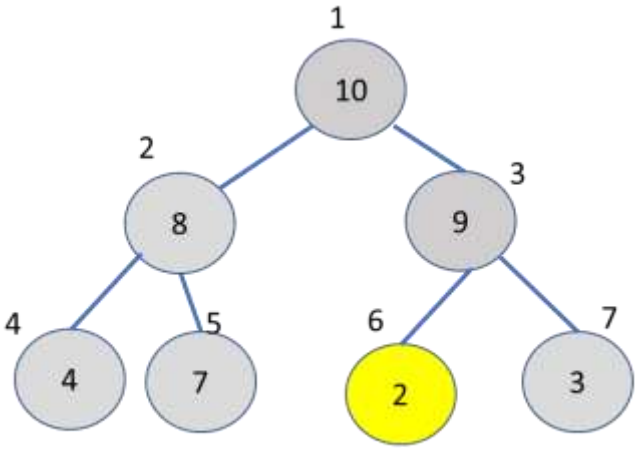


Fig-(t)

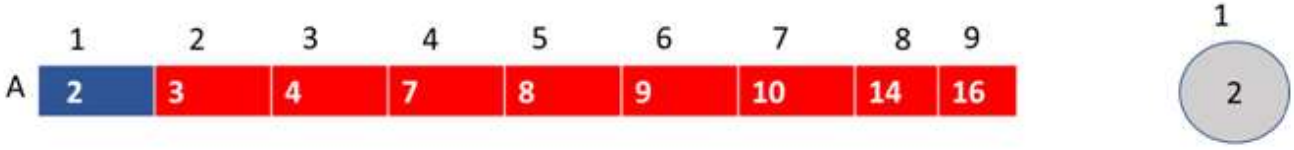
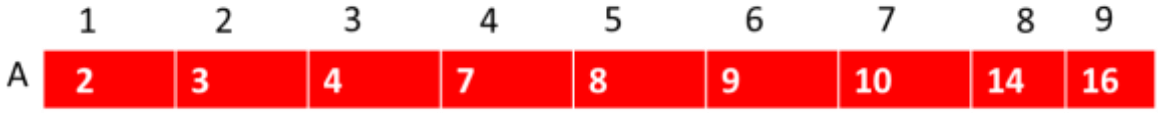


Fig-(u)

This last element is the smallest element and remains at the beginning of the array. The algorithm is finished, as the array is sorted:



ALGORITHM HeapSort(A[ ], N)
BEGIN:

```
BuildMaxHeap(A, N)

FOR j = N to 2 STEP – 1 DO

    Exchange(A[j], A[1])      /*Exchange root and last Node in the Array*/

    MaxHeapify(A, 1, j-1)     /*Readjust the Heap starting from root node*/
```

END;

Analysis

Heap Sort has **θ(nlog n) time complexities** for all the cases (best case, average case, and worst case).
As heap is the complete binary tree so the height of a complete binary tree containing n elements is log n. During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takes log n worst time because we might have to bring the element all the way from the root to the leaf. Since we repeat this n times, the heapsort step is also nlog n.

It performs sorting in **θ(1) space complexity** as it is in-place sorting.

2. 5.5 Uses of Heap Sort

1. **Heapsort**: One of the best sorting methods being in-place and log(N) time complexity in all scenarios.
2. **Selection algorithms**: Finding the min, max, both the min and max, median, or even the kth largest element can be done in linear time (often constant time) using heaps.
3. **Priority Queues**: Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority. Schedulers, timers
4. **Graph algorithms**: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal
5. Because of the lack of pointers, the operations are faster than a binary tree. Also, some more complicated heaps (such as binomial) can be merged efficiently, which is not easy to do for a binary tree.

2.5.6. Variant of Heap Sort

First, we will see some Variant of Heap sort:- Some variants of heapsort are as follows:-

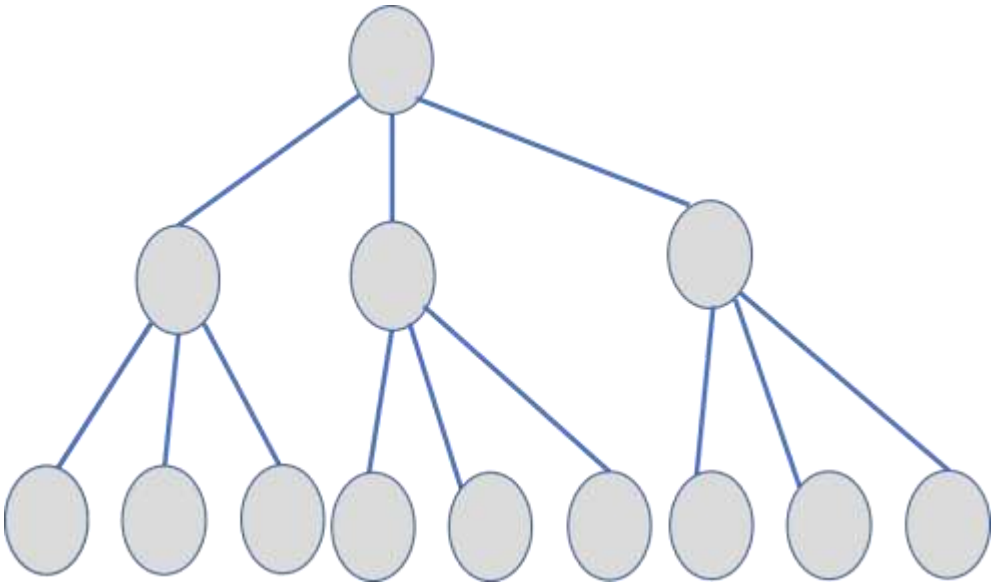
1. **Ternary Heap Sort**:- Here we use ternary heap instead of binary heap. A ternary heap is one where each node is having three children. It uses fewer numbers of comparison and swapping as compare to Binary Heap Sort.

   A ternary heap is a data structure which is part of the heap family. It inherits the properties from ternary tree and the heap data structure.

   Types: There are two kinds of ternary heaps:

   a) **Min ternary heap**
   A min ternary heap has the smallest element as its root. Parent node has to be smaller than or equal to its children.

   b) **Max ternary heap**
   A max ternary heap has the biggest element as its root. Each node has to be greater than or equal to its children.



In both cases, a ternary heap has to satisfy the heap property, which is every level of tree has to be filled, or if the last level is not filled, the leaves are distributed from left to right.

You can access a parent node or a child nodes in the array with indices below.

A root node | $i = 1$, the first item of the array

A left child node | left(i) = 3*i-1

A middle node | mid(i) = 3*i

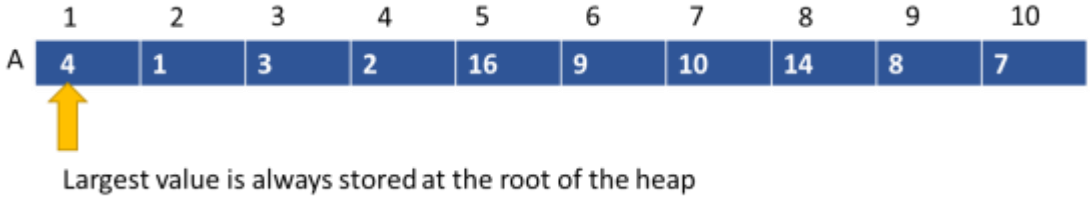
A right child node | right(*i*)=3**i*+1

A parent node | parent(*i*) = (*i+1)* / 3

2. Implementation of Priority Queue using Heap

Priority queue is a type of queue in which every element has a key associated to it and the queue returns the element according to these keys.SO this queue does not follow the concept of FCFS(First Come First Serve).

Thus, a max-priority queue returns the element with maximum key first whereas, a min-priority queue returns the element with the smallest key first.



Largest value is always stored at the root of the heap

2.5.7 Uses of Priority Queue:

Heap Implemented priority queues are used in Graph algorithms like Prim's Algorithm and Dijkstra's algorithm. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority ex-Schedulers, timers etc.

Heap data structure is best to implement priority queue because in max-heap largest value is present at root while in min-heap smallest value is present at root. So Max-Heap can be used to implement Max-Priority Queue and Min-Heap can be used to implement Min-Priority queue.

2.5.8 Operations of the priority queue

Therefore, with the help of heap following operations of the priority queue can be implemented:

a) Insert(): To insert or add new element in the priority queue.

b) PrintMax/PrintMin(): To print or get maximum or minimum element from the priority queue.

c) DeleteMax/DeleteMin(): To delete maximum or minimum element from the priority queue.

d) UpdatePriority(): To increase or decrease the priority of any element in the priority queue.

a) Implementation of Insert operation:

To insert an element in Max Heap.

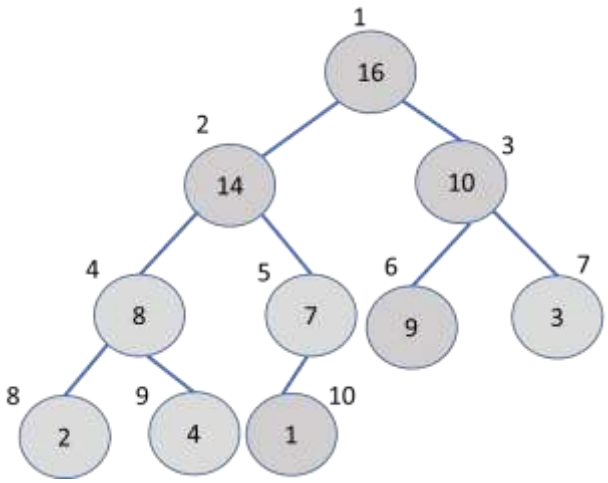
Following are the steps to to insert an element in Max Heap.

a) First update the size of the tree by adding 1 in the given size.
b) Then insert the new element at the end of the tree.
c) Now perform Heapify operation to place the new element at its correct position in the tree and make the tree either as max heap or min heap.

Time Complexity: O(log N) where N is the number of elements

b) Implementation of PrintMax()/PrintMin():

As we know that the largest element is present at the root node of max heap and smallest element is present at root in min heap. So simply print the root node to get either lagest element(i.e element with highest priority or key) or smallest element (i.e element with lowest priority or key)



	1	2	3	4	5	6	7	8	9	10
A	16	14	10	8	7	9	3	2	4	1

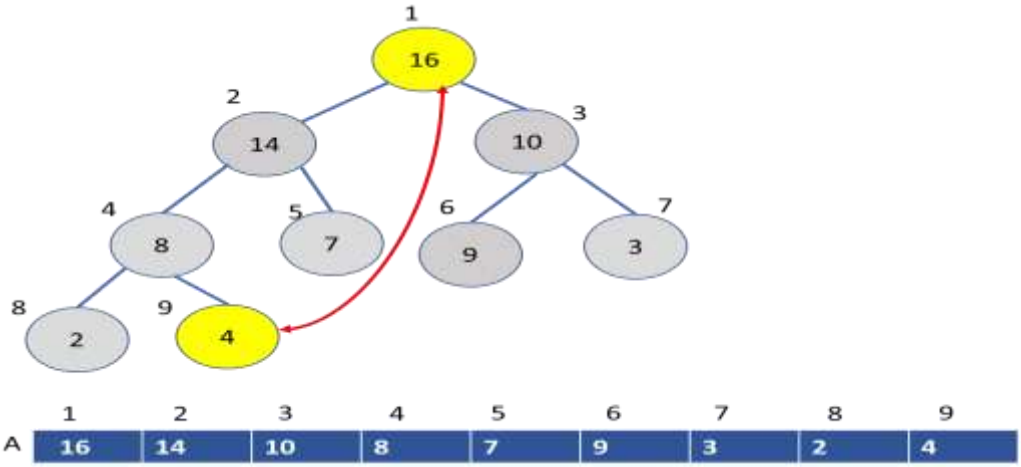
Time Complexity: O(1)

c) Implementation of DeleteMax/DeleteMin():

To delete root element from Max Heap.

Following are the steps to delete an element from Max Heap.

d) First exchange theroot element with the last element in the heap..
e) Then remove the last element by decreasing the size of the tree by 1.
f) Now perform Heapify operation to make the tree either as max heap or min heap.



Time Complexity: O(log N) where N is the number of elements in heap.

d) Implementation of UpdatePriority():

Using this operation, we can

i) Either decrease or increase the priority or key of any element then

ii) Using the heapify procedure place that update priority element at its correct position.

Time Complexity: O(log N) where N is the number of elements in heap.

Advantages of Using heap to implement Priority Queue:

To maintain the priority queue using array, it take O(Nlog N) time while O(log N) using Heap.

a. **Smooth Sort algorithm**:- This comparison-based algorithm is a variant of Heapsort algorithm, which is not a stable sort algorithm and requires O(nlog n) but may come closer to O(n) time if the input would be sorted to some extent.

2.5.9. Heap Sort Gate Questions

1	Consider a max heap, represented by the array: 40, 30, 20, 10, 15, 16, 17, 8, 4. (Gate 2015) Array Index: 1 2 3 4 5 6 7 8 9 Value: 40 30 20 10 15 16 17 8 4
A	40, 30, 20, 10, 15, 16, 17, 8, 4, 35
B	40, 35, 20, 10, 30, 16, 17, 8, 4, 15
C	40, 30, 20, 10, 35, 16, 17, 8, 4, 15
D	40, 35, 20, 10, 15, 16, 17, 8, 4, 30
AN	**B**
DL	**E**

2	An operator delete (i) for a Binary Heap Data Structure is to be designed to delete the item in the i^{th} node. Assume that the heap is implemented in an array and i refers to the i^{th} index of the array. If the heap tree has depth d (number of edges on the path from the root to the farthest leaf), then what is the time complexity to re-fix the heap efficiently after the removal of the element? (Gate 2016)
A	*O* (1)
B	*O* (d) but not *O* (1)
C	*O* (2d) but not *O* (d)
D	*O* (d^2d) but not *O* (2d)
AN	**B**
DL	**E**

3	Consider the following array of elements <89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100> The minimum number of interchanges needed to convert it into a max-heap is (Gate 2015)
A	4
B	5
C	2
D	3
AN	**D**
DL	**E**

4	Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap? (Gate 2009)
A	{25,12,16,13,10, 8,14}
B	{25,14,13, 16,10, 8,12}
C	{25,14,16,13,10, 8,12}
D	{25,14,12,13,10, 8,16}
AN	**C**
DL	**E**

4	Consider a binary max-heap implemented using an array. What is the content of the array after two delete operations on the correct answer to the previous question? (Gate 2009)
A	{14, 13,12,10, 8}
B	{14,12,13, 8,10}
C	{14,13, 8, 12,10}
D	{14,13,12, 8,10}
AN	**C**
DL	**E**

6	Consider a max heap, represented by the array: 41, 31, 21, 11, 16, 17, 18, 9, 5. Now consider that a value 35 is inserted into this heap. After insertion, the new heap is
A	41, 31, 21, 11, 16, 17, 18, 9, 5, 35
B	41, 35, 21, 11, 31, 17, 18, 9, 5, 16
C	41, 31, 21, 11, 35, 17, 18, 9, 5, 16
D	41, 35, 21, 11, 16, 17, 18, 9, 5, 31
AN	**A**
DL	**E**

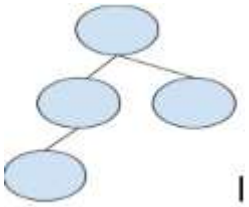
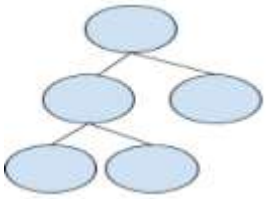
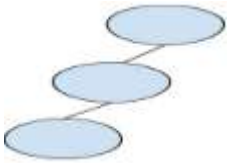
7	Consider the following array of elements 90, 20, 51, 18, 13, 16, 3, 6, 8, 12, 7, 10, 101 The minimum number of interchanges needed to convert it into a max-heap is ?
A	2
B	4
C	3
D	6
AN	**C**

8	Which of the following sequences of array elements forms a heap?
A	24, 18, 15, 7, 14, 11, 2, 13, 8, 6
B	24, 18, 15, 7, 14, 11, 2, 6, 8, 13
C	24, 18, 15, 8, 14, 11, 2, 6, 7, 13
D	24, 18, 15, 8, 14, 11, 2, 13, 6, 8
AN	**C**

9	Given a binary-max heap. The elements are stored in an array as 26, 15, 17, 12, 11, 9, 13. What is the content of the array after two delete operations?
A	15,14,9,13,11
B	15,13,14,11,9
C	15,14,13,9,11
D	15,14,13,11,9
AN	**C**
DL	**E**

10.	Consider a max heap, represented by the array: 42, 32, 22, 12, 17, 18, 19, 10, 6. Now consider that a value 36 is inserted into this heap. After insertion, the new heap is
A	42, 32, 22, 12, 17, 18, 19, 10, 6, 36
B	42, 36, 22, 12, 33, 18, 19, 10, 6, 17
C	42, 32, 22, 12, 36, 18, 19, 10, 6, 17
D	42, 36, 22, 12, 17, 18, 19, 10, 6, 32
AN	**A**

11	In order to perform heap sort , element satisfy ?
A	Strictly binary tree property.
B	AVL tree property.
C	almost complete Binary tree property
D	None of these
AN	**C**

12	Which of the following is not a complete binary tree?
A	
B	
C	
D	None of these
AN	**C**

2. 5. 10. Practice Questions on Heap Sort:

Assignment Questions Heap Sort

Q-1) What are the minimum and maximum number of elements in a heap of height h?

Q-2) Is there a min-heap with seven distinct elements so that the preorder traversal of

it gives the elements in sorted order?

Q-3) Show the steps for deleting an arbitrary element from min heap.

Q-4) What could be the height of a heap with n elements?

Q-5) Is there any optimized method to improve the time complexity for finding the Kth smallest element in min heap.

Assessment Questions Heap Sort

Q-1) Show a min-heap/max-heap with seven distinct elements so that the inorder traversal of it gives the elements in sorted order?

Q-2) Show the steps to find the Kth smallest element in min heap. Also calculate the time complexity.

Q-4) Given a sequence of numbers: 1, 2, 3, 4, 5, 6, 7, 8, 9

a. Draw a binary Min-heap by inserting the above numbers one by one

b. Also draw the tree that will be formed after calling Dequeue() on this heap

Q-5) Given an array: [3, 9, 5, 4, 8, 1, 5, 2, 7, 6]. Apply heapify over this to make a min heap and sort the elements in decreasing order?

Q-6) In Heap-Sort once a root element has been put in its final position, how much time does it take to re-heapify the structure so that the next removal can take place?

Reference: DATA STRUCTURE AND ALGORITHM MADE EASY NARASIMHA KARUMANCHI

Assignment Questions Heap Sort Variants

Q-1) Given a big file containing 10 billion of numbers, how can you find the 20 maximum numbers from that file?

Q-2) Implementation of Smooth Sort algorithm? (Given that it uses bottom up strategy, depth first and post order traversal).

Q-3) Implement the Max-Heapify operation ,insert operation and delete operation for ternary heap.

Q-3) How do we implement Queue using heap?

Q-4) How do we implement stack using heap?

2.5.11. Coding Problems on Heap Sort

1) Cube Change Problem

   Chandan gave his son a cube with side N. The N X N X N cube is made up of small 1 X 1 X 1 cubes. Chandan's son is extremely notorious just like him. So he dropped the cube inside a tank filled with Coke. The cube got totally immersed in that tank. His son was somehow able to take out the cube from the tank. But sooner his son realized that the cube had gone all dirty because of the coke. Since Chandan did not like dirty stuffs so his son decided to scrap off all the smaller cubes that got dirty in the process. A cube that had coke on any one of its six faces was considered to be dirty and scrapped off. After completing this cumbersome part his son decided to calculate volume of the scrapped off material. Since Chandan's son is weak in maths he is unable to do it alone.

   Help him in calculating the required volume.
   Practice Problem (hackerearth.com)

2) Raghu Vs Sayan Problem

Raghu and Sayan both like to eat (a lot) but since they are also looking after their health, they can only eat a limited amount of calories per day. So when Kuldeep invites them to a party, both Raghu and Sayan decide to play a game. The game is simple, both Raghu and Sayan will eat the dishes served at the party till they are full, and the one who eats maximum number of distinct dishes is the winner. However, both of them can only eat a dishes if they can finish it completely i.e. if Raghu can eat only 50 kCal in a day and has already eaten dishes worth 40 kCal, then he can't eat a dish with calorie value greater than 10 kCal.

Given that all the dishes served at the party are infinite in number, (Kuldeep doesn't want any of his friends to miss on any dish) represented by their calorie value(in kCal) and the amount of kCal Raghu and Sayan can eat in a day, your job is to find out who'll win, in case of a tie print “Tie” (quotes for clarity).

Practice Problem (hackerearth.com)

3) Divide Apples Problem

N boys are sitting in a circle. Each of them have some apples in their hand. You find that the total number of the apples can be divided by N. So you want to divide the apples equally among all the boys. But they are so lazy that each one of them only wants to give one apple to one of the neighbors at one step. Calculate the minimal number of steps to make each boy have the same number of apples.

Practice Problem (hackerearth.com)

2.6. Merge Sort:

2.6.1.Introduction of Merge Sort: Analogy

Merge sort is based on divide and conquer approach.It kept on dividing the elements present in array until it cannot be divided further and then we combine all these elements in order to get elements in sorted order.

Analogy 1: King Conquering an entire territory: Suppose you are a king and you wish to conquer an entire territory, your army is awaiting your orders. So, for a feasible approach rather than attacking the entire territory at once, find the weaker territory and conquer them one by one. So, ultimately we take a large land, split up into small problems. Capturing smaller territories is like solving small problems and then capturing as a whole.

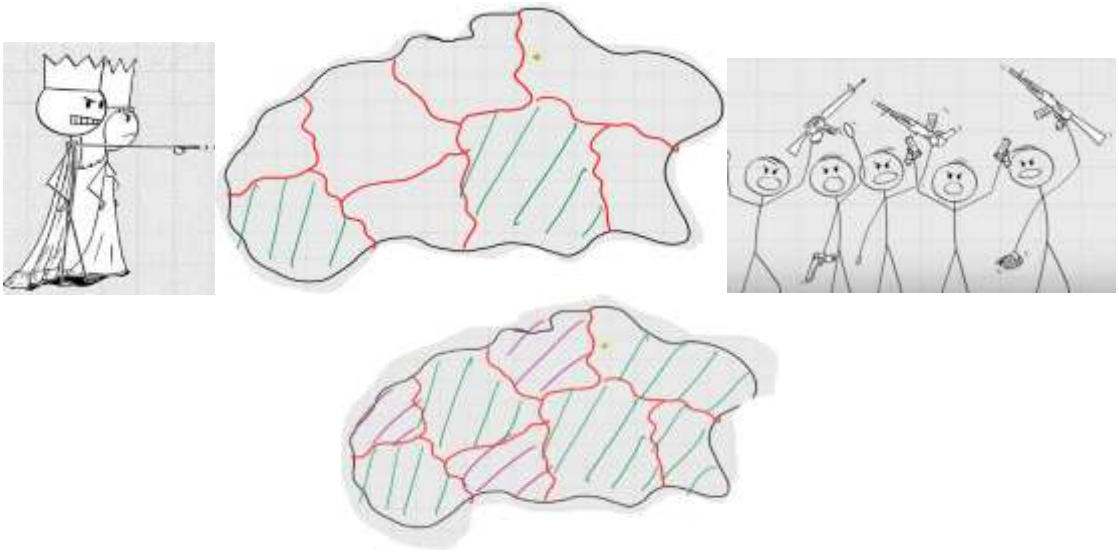


Figure 1: Elaboration about the King Conquering the Territory.

Analogy 2.Searching a page number in book: To straightaway go to any page number of a book let say page 423. I do know the fact that page numbers are serialized from 1 to 710. Initially, I would just randomly open the book at any location. Suppose page 678 opens, I know my page won't be found there. So, my search would be limited to the other half of the book. Now, if I pick any page from other half let say 295. That means that portion of the book is eliminated so I am left with portion of 296- 677 pages. In this way we are reducing our search space. If for the same brute force is used the navigation from page 1 till end would lead to many iterations. So, this gives an idea how merge sort can be applied by the same concept of splitting into smaller sub problems.

Figure 2: Elaboration on searching a page number in book

Analogy 3: Given a loaf of bread, you need to divide it into 1/8th1/8th1/8th pieces, without using any measuring tape:

Figure 3: Elaboration on loaf of bread

One day a man decided to distribute a loaf of bread to eight beggars, he wanted to distribute it in equal proportion among them. He was not having any knife with him so he decided to divide it in two equal halves, now he divided the first ½ into two equal halves that is into two ¼ equal halves, further dividing these ¼ pieces will result in getting four equal 1/8 pieces. Similarly, the second ½ bread loaf can be further divided into two ¼ pieces and then further it can be divided into four 1/8 equal pieces. Hence, total eight equal size pieces is divided among 8 beggars :

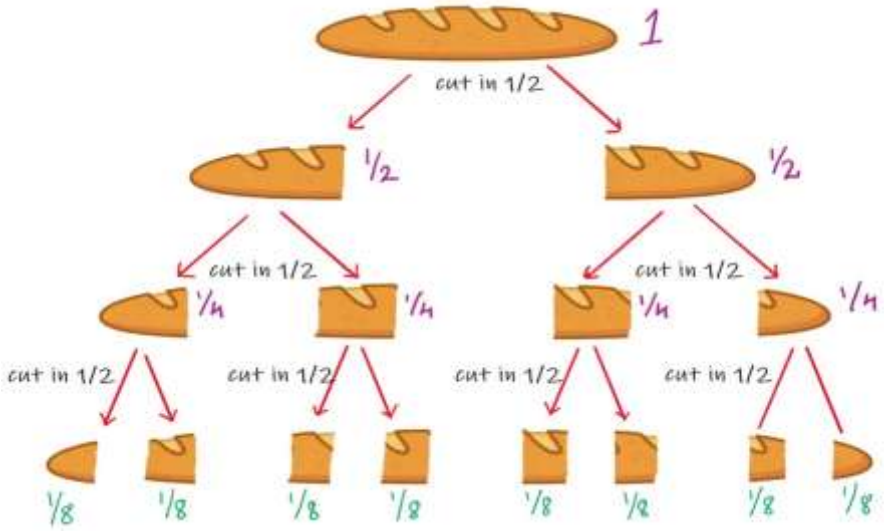


Figure 4: Divide the array into sub-arrays

Conclusion:-The manner inwhich the bread loaf was broken down in subparts is termed as a Divide procedure. The moment when it cannot be further sub divided is termed as a conquer and when we join together in order to receive the original piece is known as a merging procedure. Merge Sort, thereby work on this Divide and Conquer technology.

2.6.2.What is Divide and Conquer Approach and its Categorization:

In Divide and Conquer approach, we break a problem into sub-problems. When the solution to each sub-problem is ready, we merge the results obtained from these subproblems to get the solution to the main problem . Let's take an example of sorting an array A. Here, the sub problem is to sort the sub problem after dividing it into two parts. If q, is the mid index, then the array will be recursively divided into two halves that is from A[p……q] and then from A[q+1……r]. We can merge these and combine the list in order to get data in sorted order.

2.6.3.Why Divide and Conquer over Comparison based sorting:

In various comparison based sorting algorithm, the worst case complexity is O (n^2). This is because an element is compared multiple times in different passes. For example, In Insertion sorts, while loop runs maximum times in worst case. If array is already sorted in the decreasing order, then in worst case while loop runs 2+3+4+------n times in each iteration. Hence, we can say that the running time complexity of Insertion Sort in worst case is O (n^2).

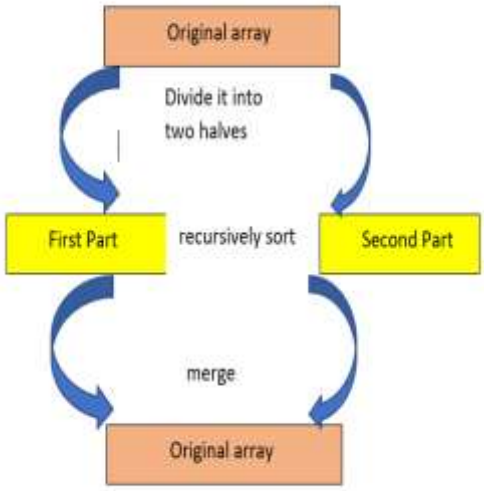


Figure 6: Divide and Conquer

Using Divide and Conquer approach, we are dividing the array into two sub-arrays. Due to the division focus shift on the half of the elements. Now, number of comparison reduced in context to

comparison based sorting algorithm. Hence, we can say that the complexity of the merge sort will be less than the complexity of comparison based sorting algorithms. In Merge sort, we divide the elements into two sub-arrays on the basis of the size of the array. In Quick Sort, division is done into two sub-arrays on the basis of the Pivot Element.

2.6.4.Merge Sort Approach:

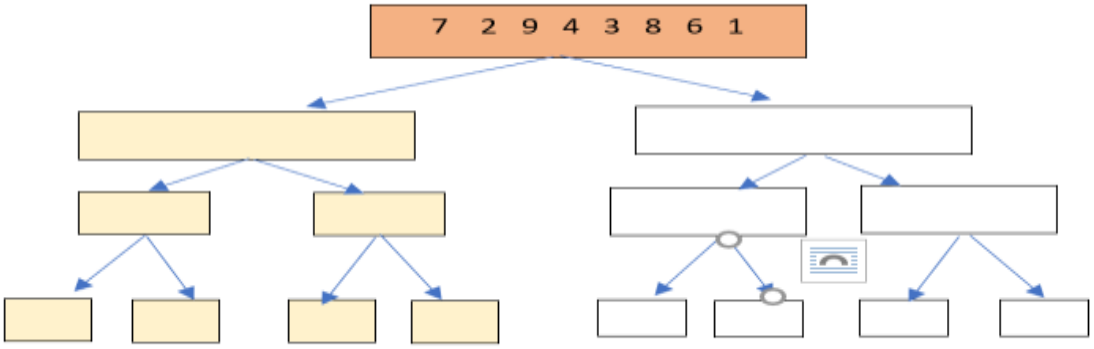
Theoretical View along with the Diagrammatic representation:

Merge sort is a sorting technique based on divide and conquer technique.

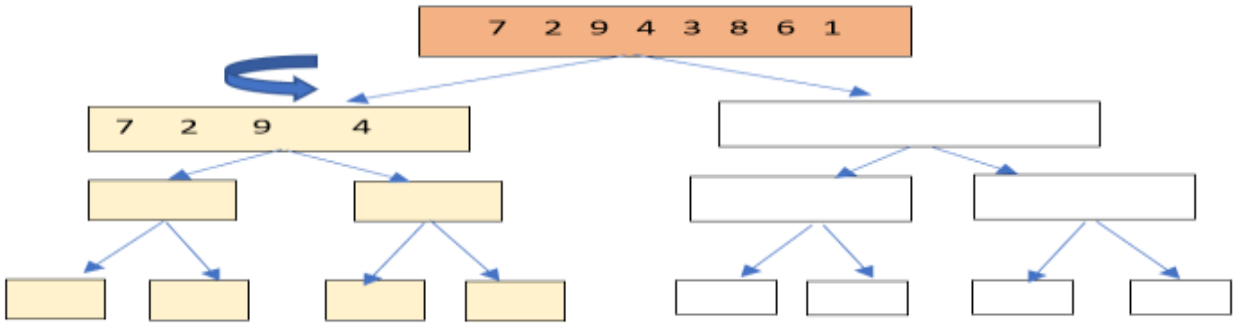
How Merge Sort Works: The Merge Sort algorithm has two procedures. These are calling of Merge function and merging procedure. Let's us understand the merging procedure before going towards calling procedure of Merge Sort.

Flow of Merge Sort:

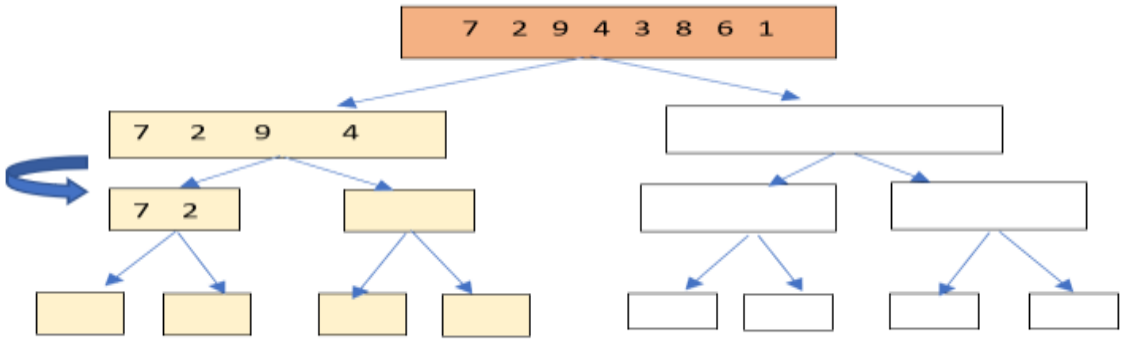
Step 1: Divide the Array of 8 Elements into two equal halves.



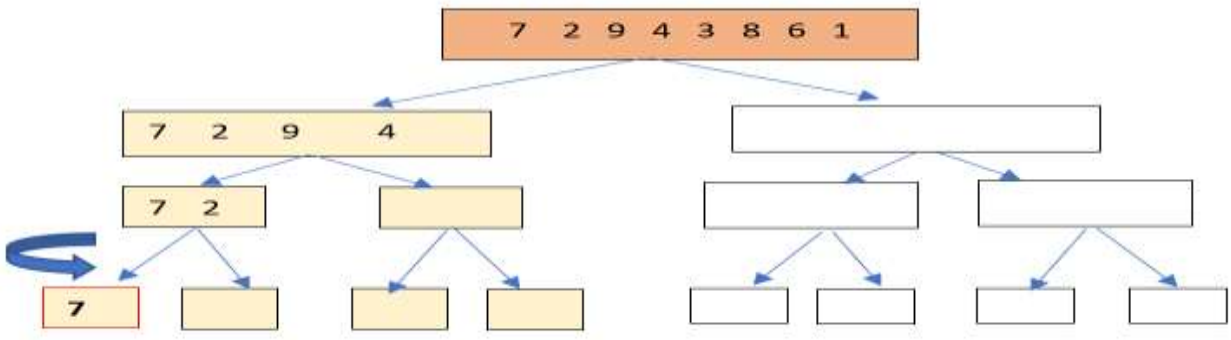
Step 2: Recursive Call and Partition:



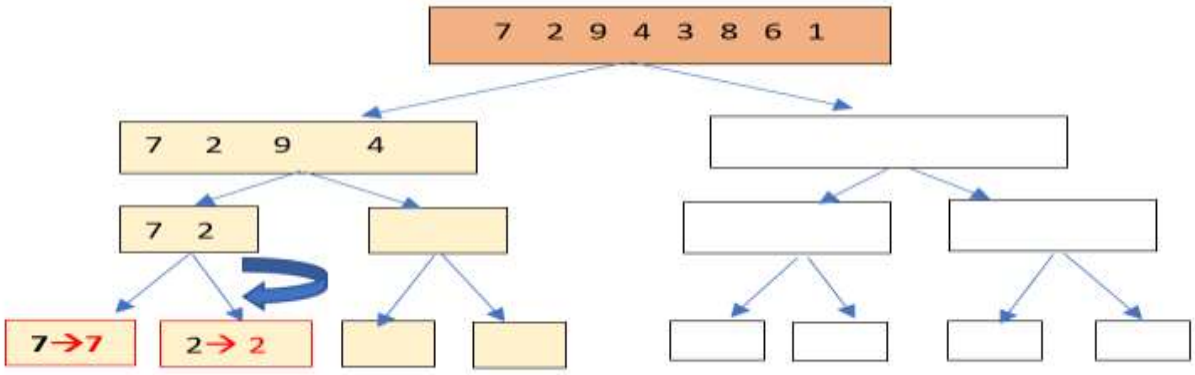
Step 3: Recursive Call and Partition:



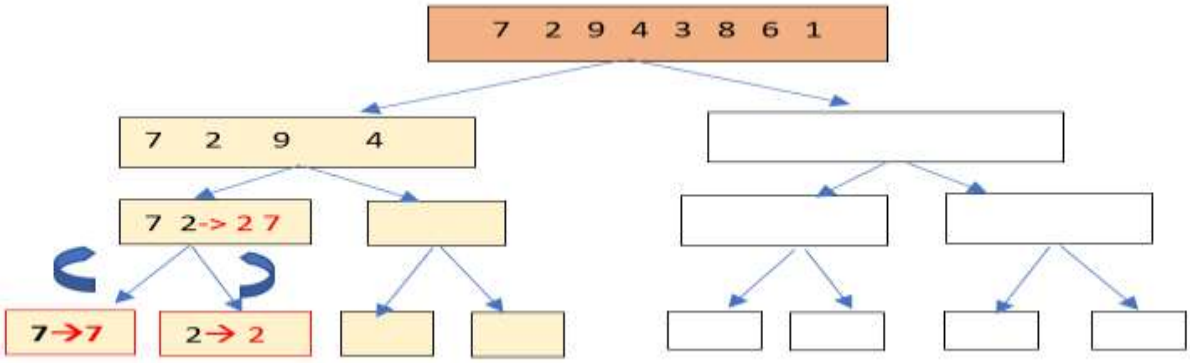
Step 4: Recursive Call and Base Case:



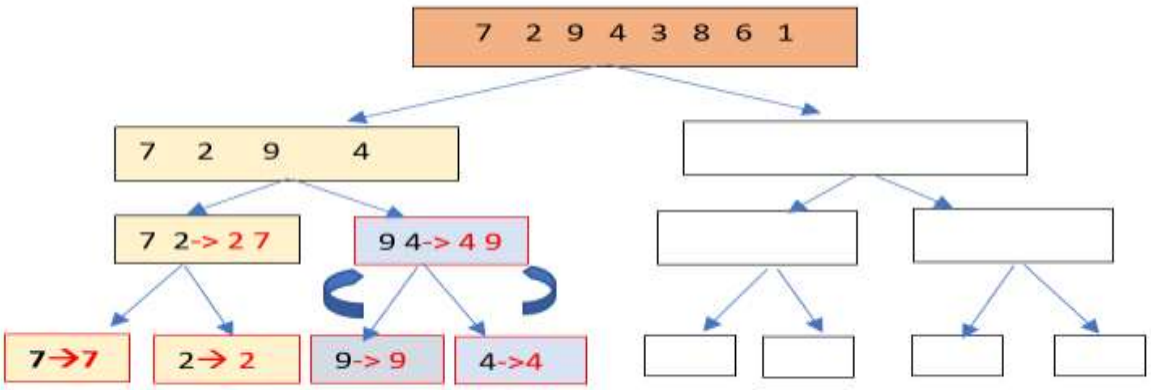
Step 5: Recursive Call and Base Case:



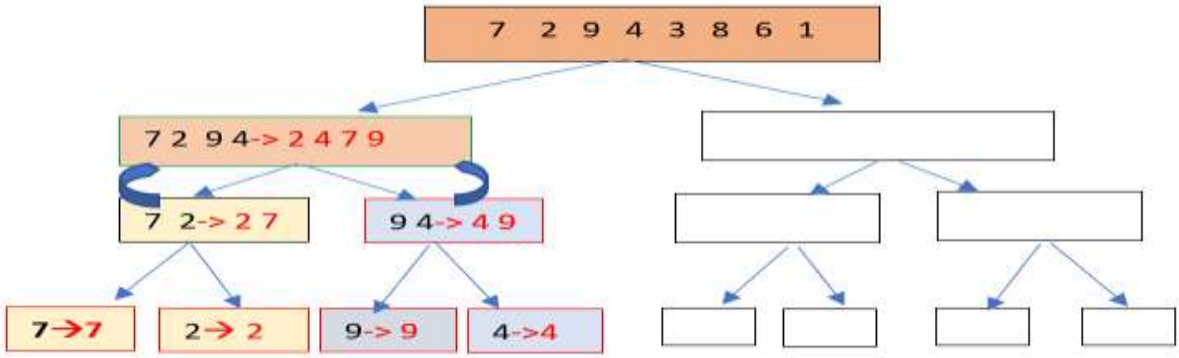
Step 6: Merge:



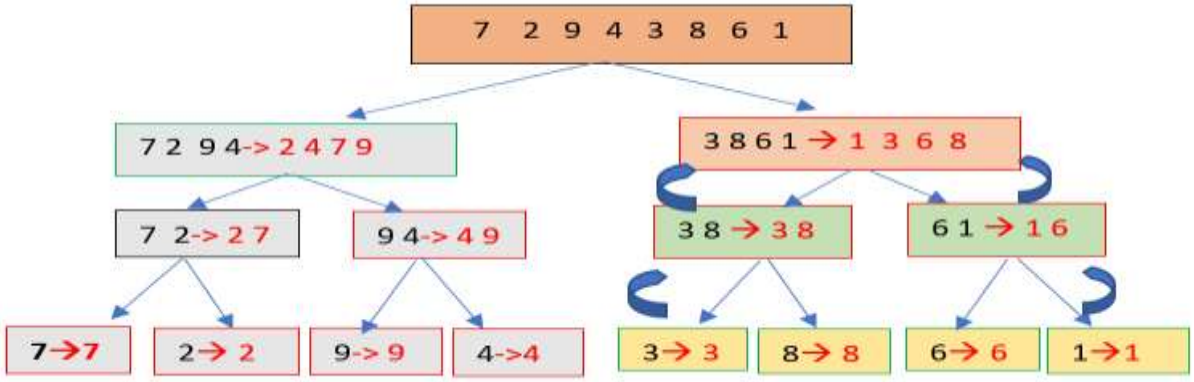
Step 7: Recursive Call, Base Case, Merge:



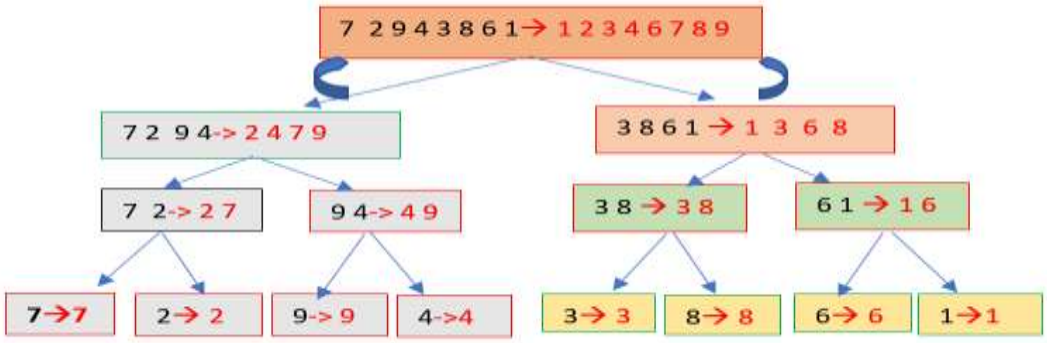
Step 8: Merge:



Step 9: Recursive Call, Base Case, Merge:



Step 10: Merge:



2.6.5 Algorithmic View of Merge Sort along with the Example Flow:

Merge Sort Algorithm works in the following steps-

- The first step is to divide the array into two halves- left and right sub arrays.
- Repeative recursive calls are made in second steps.
- This division will be done till the size of array is 1.
- As soon as each sub array contain 1 element, merge procedure is called.
- The merge procedure combines these subs sorted arrays to have a final sorted array.

Merging of two Sorted Arrays:

The heart of the merge sort algorithm is the Merging procedure. In merge procedure, we use an auxiliary array. The MERGE (A, p, q, r) procedure has following tuples :- A is an array and p, q, and r are indices into the array such that $p \leq q < r$.

The process considers following constraints:-

Input:- Two sub-arrays A (p…. q) and A (q+1…… r) are in sorted order.

Output:- Single array which is sorted and having elements ranging from p……q and q+1…… r.

Pseudo code for performing the merge operation on two sorted arrays is as follows:

ALGORITHM Merge(A[ ], p, q, r)

BEGIN:

```
1.  n1 = q-p+1
2.  n2 = r-p
3.  Create Arrays L[n1+1], R[n2+1]
4.  FOR i = 1 TO n1 DO
5.        L[i] = A[p+i-1]
6.  FOR j = 1 TO n2 DO
7.        R[j] = A[q+j]
8.  L[n1+1] = ∞
9.  R[n2+1] = ∞
10. i = 1
11. j = 1
12. FOR k=p TO r DO
13.    IF L[i] <= R[j] THEN
14.         A[k]=L[i]
15.         i = i + 1
16.    ELSE A[k] = R[j]
17.         j = j + 1
```

END;

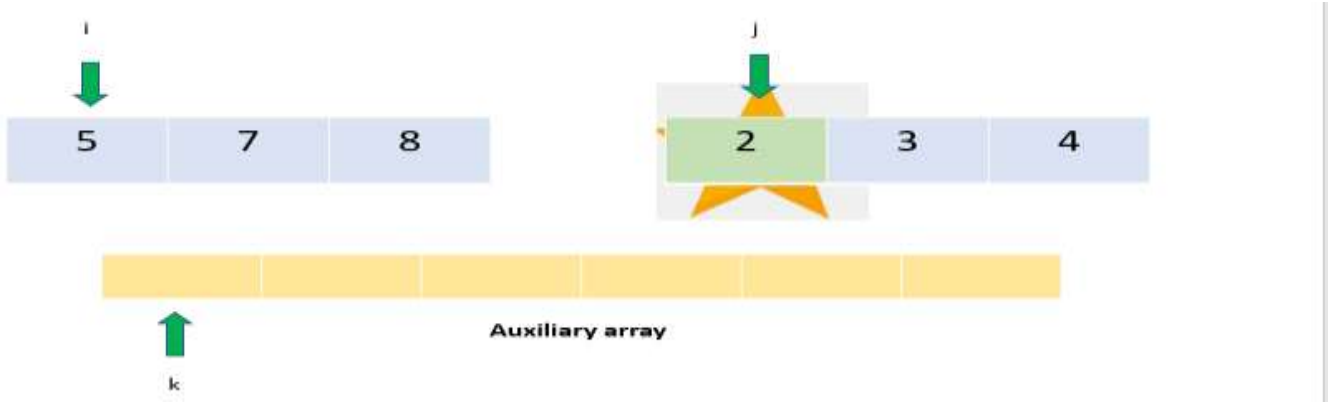
Step 1 and Step 2 is used to divide the array into sub-arrays. In Step 3, we will create left and right sub-arrays along with the memory allocation. From First three steps, running time complexity is

constant θ(1). In Step 4 and 5, Independent for loop is used for assigning Array elements to the left sub-array.In Step 6 and 7, Independent for loop is used for assigning Array elements to the right sub-array. Running Time Complexity of step 4 to step 7 is θ($n_1 + n_2$). Step 8 and Step 9 is used for storing maximum element in left and right sub-array. Step 10 and Step 11 is used to initialize the variables i and j to 1. Step 12 to Step 17 are used to compare the elements of the left and right sub-array, keeping the track of i and j variables. After comparison, left and right sub-array elements are further stored in original array. After this, running time complexity of Merge Procedure is θ(n).

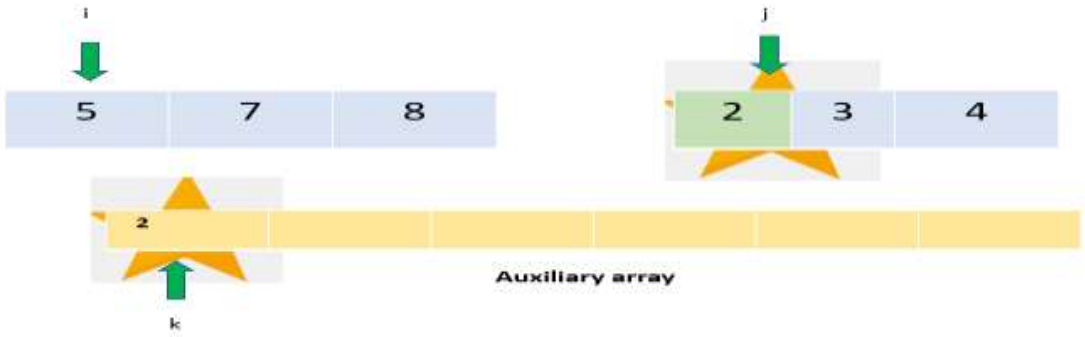
Let's take an example where we are sorting the array using this algorithm. Here there would be different steps which we have shown later. Here we are using one step in order to show how merging procedure take place**:**

Now let's go through step by ;step:-

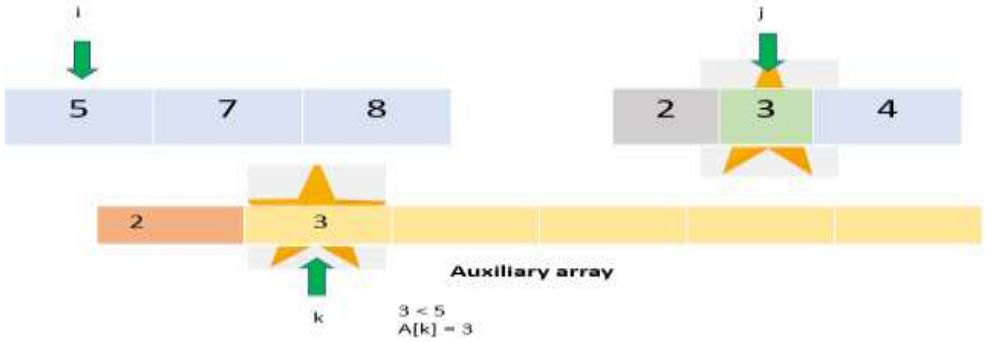
Step-01: Two sub array Left and Right is created, number of elements contained in Left sub array is given by index i and number of elements contained in Right sub array is given by index j . Number of elements contained in Left array will be L[i] = q – p + 1 and number of elements in right most array will be R[j] = r – q.



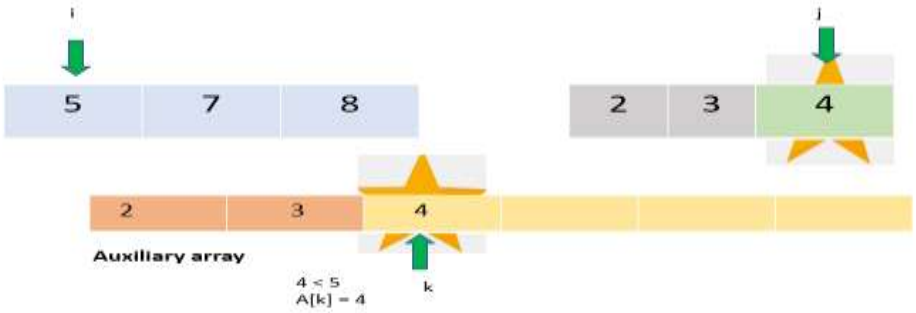
Step-02: Initiallyi , j and k will be 0.As element in right array R[0] is 2 which is than L[0] that is 5. So, A[k] will hold 2.



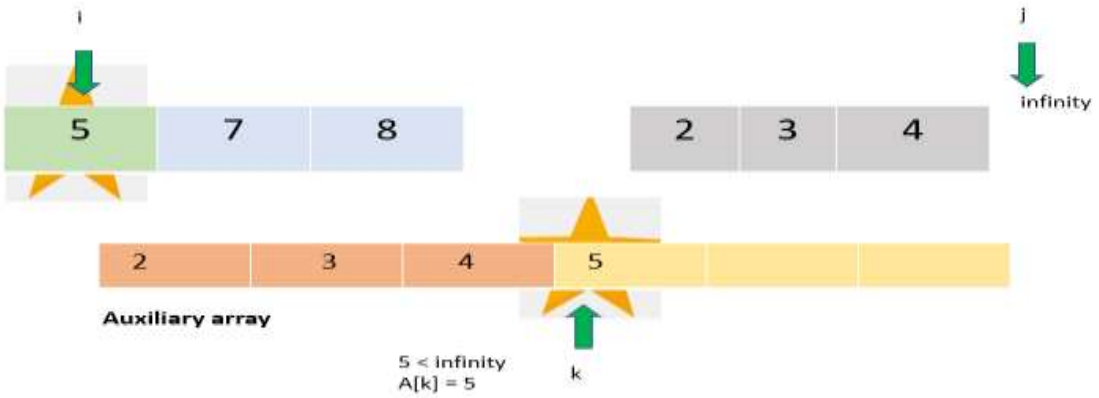
Step-03: Now j = 1 , i = 0 and k = 1. Now R[j] will point to 3 and L[i] will point to 5, since 3 is less than 5 A[k] will hold 3.



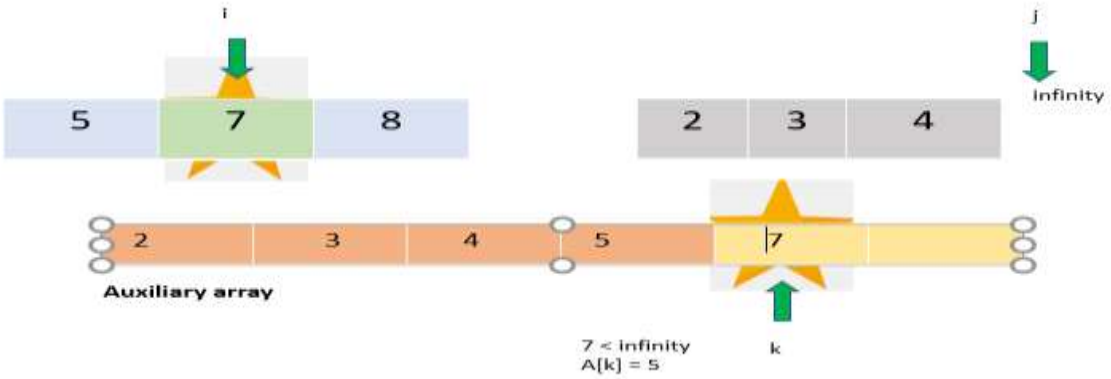
Step 4: Nowj = 2, k = 2, i = 0. Since R[j] will point to 4 and L[i] is still at 5 so B[2] will now contains 4.



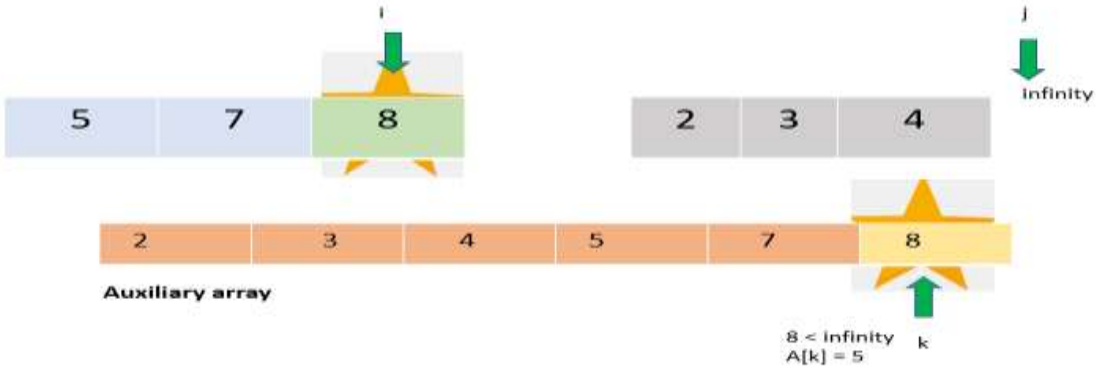
Step-05: Now i = 1, j points to senitel which stores infinity , k = 3. Now A[k] will be 5.Thus, we have-



Step 6: Now i = 2, j points to senitel which stores infinity , k is at 4th index. Now A[k] will be holding 7.Thus, we have-



Step-07:Now i = 3, j points to senitel which stores infinity , k is at 5th index. Now A[k] will be holding 8. Thus, we have--



Hence this is how the merging takes place.

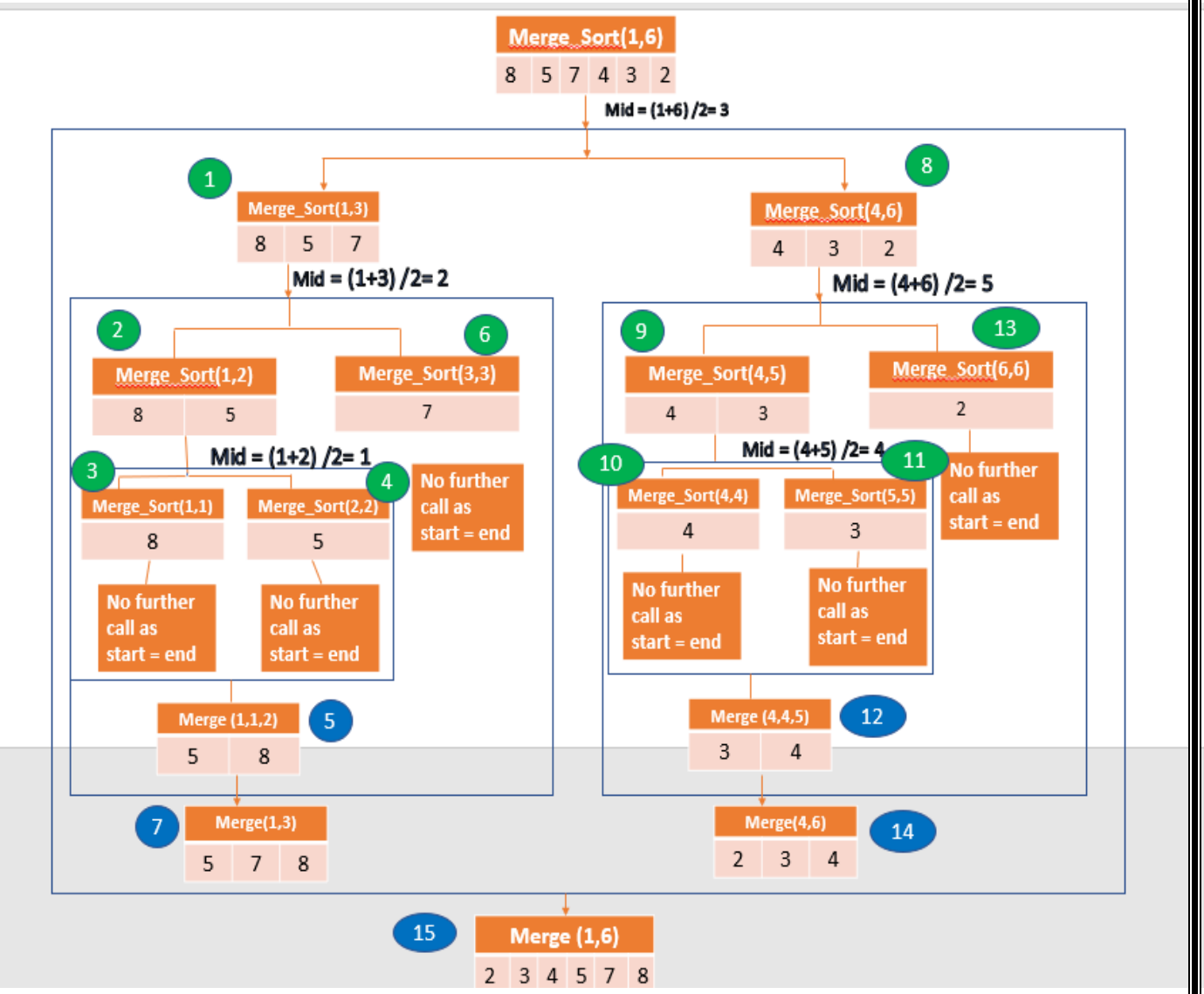
Merge_Sort Calling

The procedure MERGE-SORT (*A*, *p*, *r*) recursively calls itself until each element reaches to one that is it cannot be further subdivided. The recursive calling of this procedure is shown by the algo given below:-

```
ALGORITHM MergeSort(A[ ],p,r)
BEGIN:
        IF p<r THEN
                q=⌊(p + r)/2⌋
                MergeSort(A, p, r)
                MergeSort(A,q+1,r)
                Merge(A,p,q,r)
END;
```

To sort the entire sequence *A* = _*A*[1], *A*[2], . . . ,*A*[*n*], we make the initial call MERGESORT(*A*, 1, *length*[*A*]), where once again *length*[*A*] = *n*.

In this, p is the first element of the array and r is the last element of the array. In Step 1, we will check the number of the elements in an array. If number of element is 1 then, it can't be further divided.. If number of element is greater than 1 then, it can be further divided. q is the variable which specifies from where the array is to be divided. Merge sort function (A, p, r) is divided into two sub function Merge Sort Function(A, p, q) and Merge Sort Function(A,q+1,r). In this, n size problem is divided into two sub problems of n/2 size. This recursive calling will be done and then perform the merge operation in order to unite two sorted arrays into a single array.



2.6.6. Complexity Analysis of Merge Sort along with Diagrammatic Flow:

The complexity of merge sort can be analyzed using the piece of information given below:-

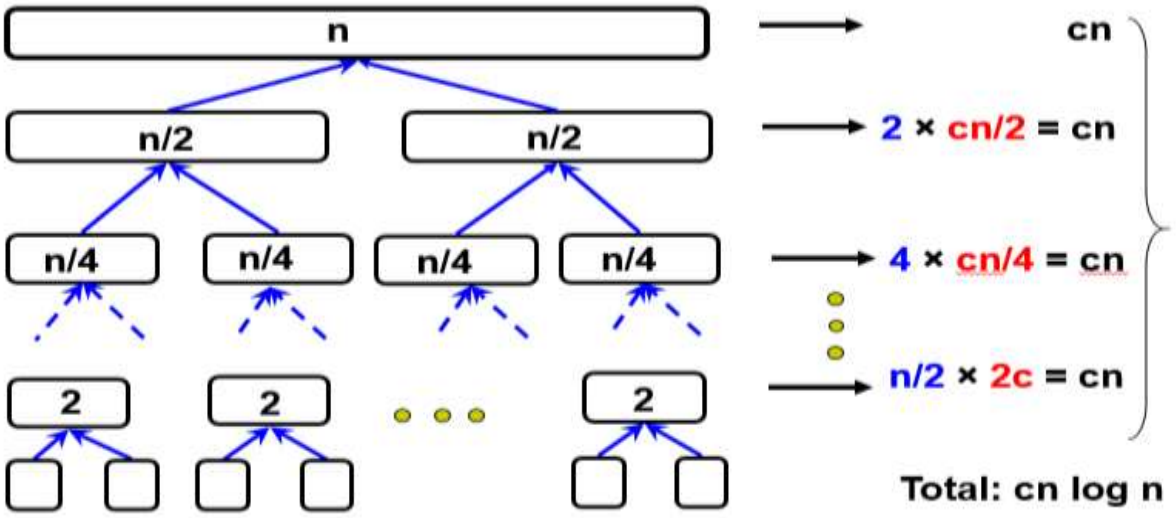
• **Divide:** When we find the middle index of the given array and it is a recursive call. So, finding the middle index each time will take $D(n) = \Theta(1)$ operations.
• **Conquer:** Whenever we solve two sub-problems, of size $n/2$ Here n varies from 1 to n/2 as it is a recursive function.
• **Combine:** Here we are sorting n elements present in array , so $C(n) = \Theta(n)$.
Hence the recurrence for the worst-case running time $T(n)$ of merge sort will be:

T (n) = c If n=1

= 2T (n/2) + c n If n is greater than 1

Where, the constant *c* represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.

By applying Case 2 of the master method, we simply say that the running time complexity of Merge Sort is O (n log n).



Here, recursion tree method is used to calculate the complexity of Merge Sort. In the above diagrammatic representation, we can see how the array is divided in to two sub-arrays. Cost at each level is calculated, which is same as cn. Hence, we can say that there are (log n+1) level in binary tree. At each and every level the cost is same as cn.

Therefore, Total Cost= Cost at each level * number of levels in the binary tree

= c n* (log n+1)

= c n log n+ c n

=c n log n

= θ (n log n)

Hence, running time complexity is θ (n log n).

The **time complexity of Merge Sort** is order of (n*Log n) in all the 3 cases (worst, average and best) as the **merge sort** always divides the array into two halves and takes linear **time** to **merge** two halves.

2.6.7. Comparison of Merge Sort Running Time Complexity with other sorting algorithm:

1. **Comparison of various sorting algorithms on the basis of Time and Space Complexity:**

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

2. **Comparison of various sorting algorithms on the basis of its stability and in place sorting:**

Sorting Algorithms	In - Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Quick Sort	Yes	No
Merge Sort	No (because it requires an extra array to merge the sorted subarrays)	Yes
Heap Sort	Yes	No

2.6.7. Variants on Merge Sort:

1. Three Way Merge Sort and so on:

Introduction to Three Way Merge Sort: Generally, in merge sort we divide the array into two equal halves and then merge them together to form a sorted array. This process of dividing the array into two halves is called as a 2 way Merge sort. Here, a question arises in the mind that can we divide the array into 3 sub-arrays, 4 sub-arrays or k sub-arrays? Then, we come with an answer that we can divide the arrays into 3 sub-arrays, 4 sub-arrays or k sub-arrays. Process of dividing the array into three sub-arrays and then merging them together to form a sorted array is called as a 3 Way Merge Sort. Process of dividing the array into four sub-arrays and then merging them together to form a sorted array is called as a 4 Way Merge Sort.

Similarly, we can define the K way Merge sort in which we can divide the array in to K sub-arrays and then merge them together to form a sorted array.

Example 1: Number of Elements are 10. Division is 4, 3 and 3

Input:45, -2, -45, 78, 30, -42, 10, 19, 73, 93
Output:-45, -42, -2, 10, 19, 30, 45, 73, 78, 93
Example 2: Number of Elements are 2. Division is 1, 1 and 0

Input:24,-18
Output:-18, 24

Pseudo Code: Three way merge Sort:

Merge sort 3(A [1….n]):

1. Ifn less than 1, then return A[1….n])
2. Let K =n/3 and m = 2n/3(considering Ceiling Values)
3. Return Merge3(Mergesort3(A[1….K], Mergesort3(A[K+1….m], Mergesort3(A[m+1….n])

Time Complexity: In 2 way merge sort, the recurrence obtained are: T(n) = 2T(n/2) + θ (n). By applying master method on above mentioned recurrence we can compute the running time complexity of 2 way merge sort which is **θ (n log n).** This is obtained with the help of Case 2 of the Master Method. Similarly, in case of 3-way Merge sort the recurrence obtained is: T(n) = 3T(n/3) + O(n) by solving it using Master method Case 2, we get its complexity as **O(n log $_3$n).** Similarly, in case of 4-way Merge sort the recurrence obtained is: T (n) = 4 T (n/4) + θ(n) by solving it using Master method Case 2, we get its complexity as **θ(n log $_4$n).** Similarly, in case of K-way Merge sort the recurrence obtained is: T (n) = K T (n/K) + θ(n) by solving it using Master method Case 2, we get its complexity as **θ(n log $_k$ n)**

2.Tim Sort:

Introduction to Tim Sort: Tim Peter in 2002 derived a sorting algorithm with his name. In this sorting algorithm, Tim proposed the blended model of two sorting algorithm working together i.e. Insertion and Merge Sort in an optimal way on real world data. It is an adaptive sorting algorithm which requires O (n log n) comparisons to sort an array of n elements in average case and worst case. But in the best case the blended mode of Merge and Insertion sort will take O (n) running time to sort an array of n elements.

Example 1:

Input Elements is:-2, 7, 15, -14, 0, 15, 0, 7, -7, -4, -13, 5, 8, -14, 12

Output Elements is: -14, -14,-13,-7 ,-4 ,-2 ,0 ,0 ,5 ,7 ,7 ,8 ,12 ,15 ,15

Technique used in Tim Sort: Let us assume that we are given an array of n elements and we have to sort the given array using Tim Sort. In Tim Sort, division of array is done into several parts on the basis of the run. These runs will further be sort one by one using insertion Sort and then, in turn will be combined together in order to form a sorted array using Merge function. Basic idea behind this sorting algorithm is that Insertion sort works more optimally on the short arrays in comparative to large arrays.

3	10	15	20	21	3	5	10	2	4	5	10	14	16	20
Run 1					Run 2			Run 3						

Step by Step Algorithm for Tim Sort:

1. Divide the given array into the number of sub-arrays on the basis of the runs.
2. Next Step is to consider the size of the run which can either be 32 or 64.
3. Next Step is to sort each run one by one using Insertion Sort.
4. Then, Merge the sorted runs one by one in such a way using the merge function of the Merge Sort.
5. After performing Merge function of Merge Sort, we will then combine the sorted sub-arrays in to the sorted array.

Time Complexity: Time complexity of Tim Sort in best case is better than the running time complexity of Merge Sort. This is only because of the blended mode used by the Tim sort to sort elements. Blended way means, Tim Sort is basically a combination of Insertion sort and Merge Sort.

Complexity	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Time Complexity	Ω(n)	θ(n log n)	O(n log n)

Space Complexity			O(N)

3. Iterative Merge Sort (Bottom up Merge Sort):

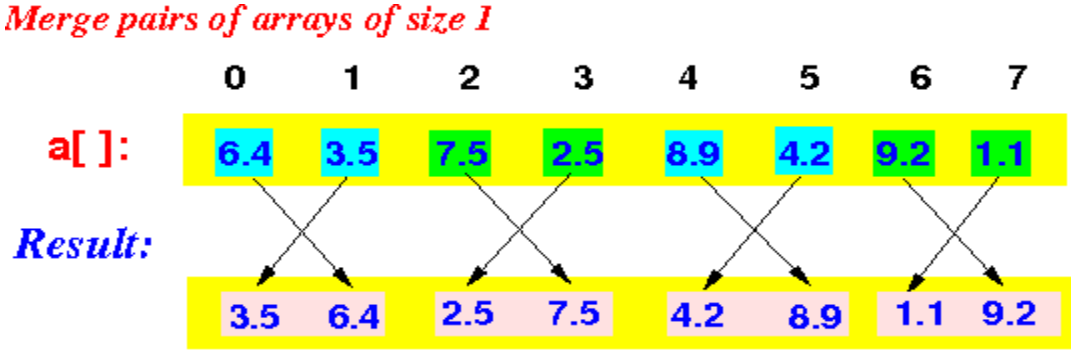
Introduction to Iterative Merge Sort (Bottom up Merge Sort):

In Bottom-up Merge Sort, we will consider each element of an array as a separate sub-array. Then, we will merge the pairs of adjacent arrays of 1 element each. Then, merges pairs of adjacent arrays of 2 elements, next merges pairs of adjacent arrays of 4 elements and so on, until the whole array is merged.

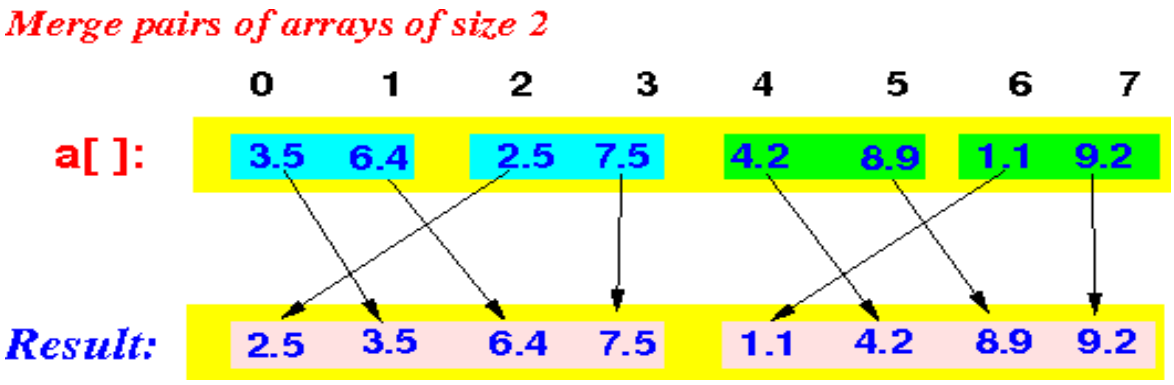
Example on Iterative Merge Sort (Bottom up Merge Sort):

Input array: 6.4 3.5 7.5 2.5 8.9 4.2 9.2 1.1

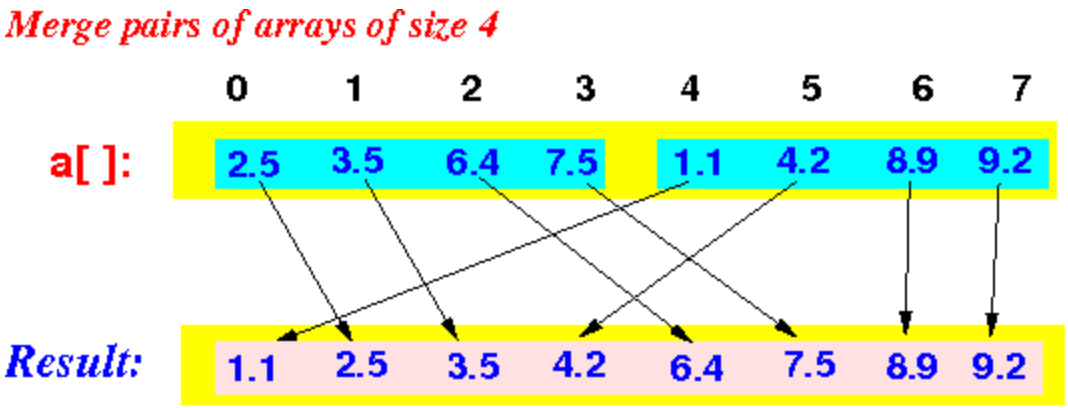
Iteration 1: Merge pairs of adjacent arrays of size = 1



Iteration 2: Merge pairs of adjacent arrays of size = 2



Iteration 3: Merge pairs of adjacent arrays of size = 4



Time Complexity: Running Time Complexity of Iterative Merge sort will be same as the recursive merge sort that is O (n log n).

2.6.8. Real life scenarios where Merge sort can be used:

There are various real life scenarios where merge sort can be used. In this, we will discuss about the two scenarios in order to have a depth understanding of the concept.

1. **E-commerce website:** Most of us might see a section in various e-commerce website" You might like" in which we found some stuff of our choice .Do you know how it actually work? E-commerce website maintains an array of its all users with their activities on the website and when they have some latest products in their products list, which also store them in another array, will in turn divide those latest products in array according to user's array of choice. Another mega example of using merge sorting is two of India's biggest e-commerce website policybazar.com and trivago.com who use merge sorting for creating best price and stuff for its users.
2. **Division of Long Loaf of Breads:** If you want to divide a long loaf of bread in 8 or 16 equal pieces, generally people cut it into two equal halves first and then cut each half into two equal halves again, repeating the process until you get as many pieces as you want – 8, 16, 32, or whatever. Almost nobody tries to divide the loaf into 8 pieces all at once – people can guess halves much better than eighths.

2.6.9. Animation of Merge Sort: Step by Step Guide:

Following are the two links for better understanding of Merge Sort Concept with the Example:

1. **Merge Sort Concept:** https://www.youtube.com/watch?v=dxuIwsPKRts
2. **Merge Sort Example Step by Step:** https://www.youtube.com/watch?v=b2z_hp7Q-wU

2.6.10. Objective Type Questions on Merge Sort: Solved

1	Merge sort uses which of the following technique to implement sorting?
A	Backtracking
B	Greedy Algorithm
C	Divide and Conquer
D	Dynamic Programming
ANS	**C**

2	What is the average case time complexity of merge sort?
A	O(n log n)
B	$O(n^2)$
C	$O(n^2 \log n)$
D	$O(n \log n^2)$
ANS	**A**

3	What is the auxiliary space complexity of merge sort?
A	O(1)
B	O(log n)
C	O(n)
D	O(n log n)
ANS	**C**

4	Which of the following method is used for sorting in merge sort?
A	Merging
B	Partitioning
C	Selection
D	Exchanging
ANS	**A**

5	Which of the following is not a variant of merge sort?
A	in-place merge sort
B	bottom up merge sort
C	top down merge sort
D	linear merge sort
ANS	**D**

6	Which of the following sorting algorithm does not use recursion?
A	quick sort
B	merge sort
C	heap sort
D	bottom up merge sort
ANS	**D**

Company Based Objective Type Questions on Merge Sort:Solved

1	Which of the following is not stable sorting in its typical implementation?
A	Insertion
B	Merge
C	Quick
D	Bubble
ANS	**C**

2	Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the number of swap operations is minimized in general?
A	Heap
B	Selection
C	Insertion
D	Merge
ANS	**B**

3	You have to sort 1 GB of data with only 100 MB of available main memory. Which sorting technique will be most appropriate?
A	Heap
B	Merge
C	Quick
D	Insertion
ANS	**B**

4	**In a modified merge sort, the input array is splitted at a position one-third of the length (N) of the array. Which of the following is the tightest upper bound on time complexity of this modified Merge Sort?**
A	N(log N base 3)
B	N(log N base 2/3)
C	N(log N base 1/3)
D	N(log N base 3/2)
ANS	**D**

5	**A list of n string, each of length n, is sorted into lexicographic order using the merge-sort algorithm. The worst case running time of this computation is**
A	O (n log n)
B	O (n^2 log n)
C	O (n^2 + log n)
D	O (n^2)
ANS	**B**

6	**Which of the following is true about merge sort?**
A	Merge Sort works better than quick sort if data is accessed from slow sequential memory.
B	Merge Sort is stable sort by nature
C	Merge sort outperforms heap sort in most of the practical situations.
D	All of the above.
ANS	**D**

GATE/PSUs Objective Type Questions on Merge Sort:Solved

1	**Assume that a Merge sort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?GATE 2015**
A	256
B	512
C	1024
D	2048
ANS	**B**

2	**If one uses straight two-way merge sort algorithm to sort the following elements in ascending order: 20, 47, 15, 8, 9, 4, 40, 30, 12, 17 then the order of these elements after second pass of the algorithm is:ISRO 2015**
A	8, 9, 15, 20, 47, 4, 12, 17, 30, 40
B	8, 15, 20, 47, 4, 9, 30, 40, 12, 17
C	15, 20, 47, 4, 8, 9, 12, 30, 40, 17
D	4, 8, 9, 15, 20, 47, 12, 17, 30, 40
ANS	**B**

3	**Given two sorted list of size m and n respectively. The number of comparisons needed the worst case by the merge sort algorithm will beISRO 2018**
A	m * n
B	Minimum of m and n
C	Maximum of m and n
D	m + n -1
ANS	**D**

4	**Of the following sorting algorithms, which has a running time that is least dependent on the initial ordering of the input? ISRO 2018**
A	Merge
B	Insertion
C	Selection
D	Quick
ANS	**A**

5	**For merging two sorted lists of size m and n into sorted list of size m + n, we require comparisons of GATE 1995**
A	O (n)
B	O (m)
C	O (n +m)
D	O (log n + log m)
ANS	**C**

6	**Given Log n sorted list each of size n/ Log n. What is the total time required to merge them into a single list of size n GATE 2016**
A	O (n)
B	O (log n)
C	O (n log n)
D	O (n log log n)
ANS	**D**

2.6.11. Competitive Coding Problems on Merge Sort:Solved

Coding Problem 1:Count of larger elements on right side of each element in an array

Asked in:(Hack-withInfy First Round 2019)

Difficulty Level: Hard

Step 1: Problem Statement: Given an array **arr[]** consisting of **N** integers, the task is to count the number of **greater elements** on the **right side** of each array element.

Step 2: Understanding the Problem with the help of Example:
Input:arr[] = {3, 7, 1, 5, 9, 2}
Output: {3, 1, 3, 1, 0, 0}
Explanation:
For arr[0], the elements greater than it on the right are {7, 5, 9}.
For arr[1], the only element greater than it on the right is {9}.
For arr[2], the elements greater than it on the right are {5, 9, 2}.
For arr[3], the only element greater than it on the right is {9}.
For arr[4], no greater elements exist on the right.
For arr[5], no element exist on the right.

Step3: Naive Approach: The simplest approach is to iterate all the array elements using two loops and for each array element, count the number of elements greater than it on its right side and then print it.
Time Complexity: $O(N^2)$ *and* **Auxiliary Space:** *O(1)*

Optimized Approach: The problem can be solved using the concept of **Merge sort** in descending order. Follow the step by step algorithm to solve the problem:

1.Initialize an array **count[]** where **count[i]** store the respective count of greater elements on the right for every **arr[i]**

2.Take the indexes **i** and **j**, and compare the elements in an array.

3:If higher index element is greater than the lower index element then, the entire higher index element will be greater than all the elements after that lower index.

4:Since the left part is already sorted, add the count of elements after the lower index element to the **count[]** array for the lower index.

5:Repeat the above steps until the entire array is sorted.

6:Finally print the values of **count[]** array.

Time Complexity: O(N*log N) and **Auxiliary Space:** O(N)

Coding Problem 2:Count pairs (i, j) from given array such that i < j and arr[i] > K * arr[j]

Asked in:(Goldman Sachs 2016)

Difficulty Level: Moderate

Step 1: Problem Statement: Given an array **arr[]** of length **N** and an integer **K**, the task is to count the number of pairs **(i, j)** such that **i < j** and **arr[i] > K * arr[j]**.

Step 2: Understanding the Problem with the help of Example:

Input:arr[] = {5, 6, 2, 5}, K = 2
Output: 2
Explanation: The array consists of two such pairs:
(5, 2): Index of 5 and 2 are 0, 2 respectively. Therefore, the required conditions (0 < 2 and 5 > 2 * 2) are satisfied.
(6, 2): Index of 6 and 2 are 1, 2 respectively. Therefore, the required conditions (0 < 2 and 6 > 2 * 2) are satisfied.

Step3: Naive Approach: The simplest approach to solve the problem is to traverse traverse an array and for every index, find numbers having indices greater than it, such that the element in it when multiplied by **K** is less than the element at the current index.

Steps to solve the problem using Naïve Approach:

1. Initialize a variable, say **count**, with **0** to count the total number of required pairs.
2. Traverse the array from left to right.
3. For each possible index, say **i**, traverse the indices **i + 1** to **N – 1** and increase the value of **count** by **1** if any element, say arr[j], is found such that **arr[j] * K** is less than **arr[i]**.
4. After complete traversal of the array, print **count**as the required count of pairs.

Time Complexity: O(N^2) and **Auxiliary Space:** O(N)

Optimized Approach: The idea is to use the **concept of merge sort**and then count pairs according to the given conditions.

Steps to solve the problem Optimized Approach:

1. Initialize a variable, say **answer**, to count the number of pairs satisfying the given condition.
2. Repeatedly partition the array into two equal halves or almost equal halves until one element is left in each partition.
3. Call a recursive function that counts the number of times the condition **arr[i] > K * arr[j]** and **i < j** is satisfied after merging the two partitions.
4. Perform it by initializing two variables, say **i** and **j**, for the indices of the first and second half respectively.
5. Increment **j** till **arr[i] > K * arr[j]** and **j < size** of the second half. Add **(j – (mid + 1))** to the **answer** and increment **i**.
6. After completing the above steps, print the value of **answer** as the required number of pairs.

Time Complexity:O (N*log N) **and Auxiliary Space:**O (N)

***Coding Problem 3:*Count sub-sequences for every array element in which they are the maximum**

Asked in:(Hashed-In 2019)

Difficulty Level: Moderate

Step 1: Problem Statement: Given an array **arr[]** consisting of **N** unique elements, the task is to generate an array **B[]** of length **N** such that **B[i]** is the number of subsequencesin which **arr[i]** is the maximum element.

Step 2: Understanding the Problem with the help of Example:

Input:arr[] = {2, 3, 1}
Output: {2, 4, 1}
Explanation: Subsequences in which arr[0] (= 2) is maximum are {2}, {2, 1}.
Subsequences in which arr[1] (= 3) is maximum are {3}, {1, 3, 2}, {2, 3}, {1, 3}.
Subsequence in which arr[2] (= 1) is maximum is {1}.

Step 3: Optimized Approach: The problem can be solved by observing that all the subsequences where an element, **arr[i]**, will appear as the maximum will contain all the elements less than **arr[i]**. Therefore, the total number of distinct subsequences will be $\mathbf{2^{(Number\ of\ elements\ less\ than\ arr[i])}}$.

Steps to solve the problem Optimized Approach

1. Sort the array **arr[]** indices with respect to their corresponding values present in the given array and store that indices in array **indices[]**, where**arr[indices[i]] <arr[indices[i+1]]**.
2. Initialize an integer, **subsequence** with **1** to store the number of possible subsequences.
3. Iterate **N** times with pointer over the range **[0, N-1]** using a variable, **i**.
    a. **B[indices[i]]** is the number of subsequences in which **arr[indices[i]]** is maximum i.e., $\mathbf{2}^{\mathbf{i}}$, as there will be **i** elements less than **arr[indices[i]].**
    b. Store the answer for **B[indices[i]]** as **B[indices[i]] = subsequence**.
    c. Update **subsequence**by multiplying it by **2**.
4. Print the elements of the array**B[]** as the answer.

Time Complexity: O(Nlog n) and **Auxiliary Space:** O(N)

2.6.12. Competitive Coding Problems on Merge Sort: unsolved

Coding Problem 1:Count sub-sequences which contains both the maximum and minimum array element.

Difficulty Level: Moderate

Problem Statement: Given an array **arr[]** consisting of **N** integers, the task is to find the number of subsequences which contain the maximum as well as the minimum element present in the given array.

Coding Problem 2: Partition array into two sub-arrays with every element in the right sub-array strictly greater than every element in left sub-array

Difficulty Level: Moderate

Problem Statement: Given an array **arr[]** consisting of **N** integers, the task is to partition the array into two non-empty sub-arrays such that every element present in the right sub-array is strictly greater than every element present in the left sub-array. If it is possible to do so, then print the two resultant sub-arrays. Otherwise, print **"Impossible".**

***Coding Problem 3*: Count Ways to divide C in two parts and add to A and B to make A strictly greater than B**

Difficulty Level: Moderate

Problem Statement: Given three integers **A, B** and **C**, the task is to count the number of ways to divide **C** into two parts and add to **A** and **B** such that **A is strictly greater than B**.

2.7. Quick Sort:

2.7.1. Analogy: Arranging students with different heights: This is an analogy of quick sort as any one student here looking for its appropriate place can be a pivot. Student's smaller than him will stand ahead of him and taller behind him. In the same way when a single person has found it's exact place the other in the same manner can be the pivot and find it's place. Leading to a very quick way of line formation according to heights.

Quick Sort is another sorting algorithm following the approach of **Divide and Conquer.** Another name of quick sort is **Partition exchange sort** because of the reason, it selects a pivot element and does the array elements partitioning as per that pivot. Placing element smaller than pivot to the left and greater than pivot to the right.

Quick Sort is also known as **Selection exchange sort** because in selection sort we select the position and find an element for that position whereas in Quick Sort we select the element and finding the position. As compared to Merge Sort if the size of input is small, quick sort runs faster.

2.7.2. Applications of Quick Sort:

1. Commercial applications generally prefer quick sort, since it runs fast and no additional requirement of memory.
2. Medical monitoring.
3. Monitoring & control in industrial & Research plants handling dangerous material.
4. Search for information
5. Operations research
6. Event-driven simulation
7. Numerical computations
8. Combinatorial search

9. When there is a limitation on memory then randomised quicksort can be used. Merge sort is not an in-place sorting algorithm and requires some extra space.
10. Quicksort is part of the standard C library and is the most frequently used tool - especially for arrays. Example: Spread sheets and database program.

2.7.3. Partitioning of an array:

Generally, in merge sort we use to divide the array into two-halves but in quick sort division is done on the basis of pivot element which could be either first, last element or any random element.

Steps to divide an array into sub-arrays on the basis of the Pivot Element:

Divide: Initially, we divide Array A into two sub-arrays A[p……q-1] and A [q+1……..r]. A[q] returns the sorted array element. In line 2 we get q. Every element in A[p……q-1] are less than or equal to A[q]. Every element in A[q+1……r] is greater than A[q].

Conquer: Recursively call QuickSort.

Combine: Nothing done in combine.

Partitioning Algorithm:

ALGORITHM Partition(A[ ], p, r)

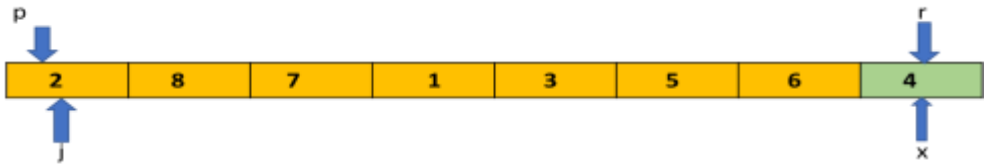
BEGIN:

```
x = A[r]
i = p-1
FOR j = p TO r-1 DO
    IF A[j] < = x THEN
        i = i+1
        ExchangeA[i] with A[j]
Exchange A[i+1] with A[r]
RETURN i+1
```

END;

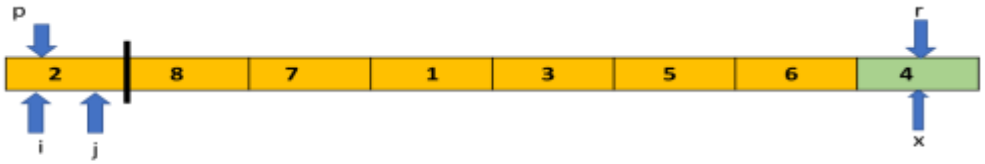
2.7.4.Example to demonstrate working of Quick Sort:

STEP 1



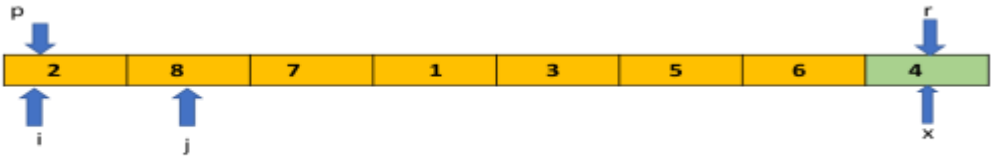
i is one step behind p, so i=0 here, assuming array indexing starting from 1. j will be equal to p. x=A[r]= 4, x is the pivot chosen.

STEP 2



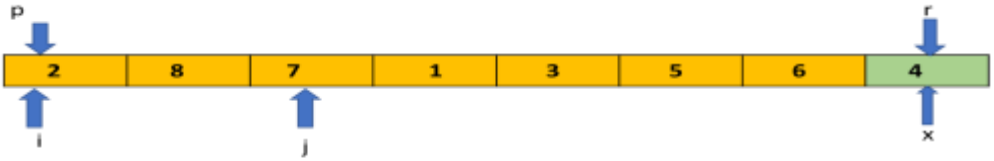
for j=1 2<=4 True so i will increment by i=1 exchange will happen which is same for this.

STEP 3



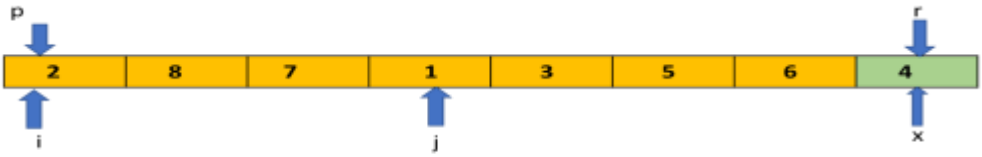
for j=2 8<=4 False so no change, j simply moves to next index

STEP 4



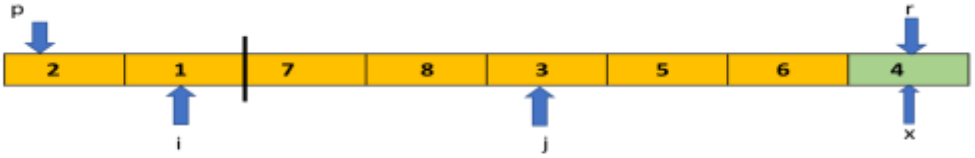
for j=3 7<=4 False so no change, j simply moves to next index

STEP 5



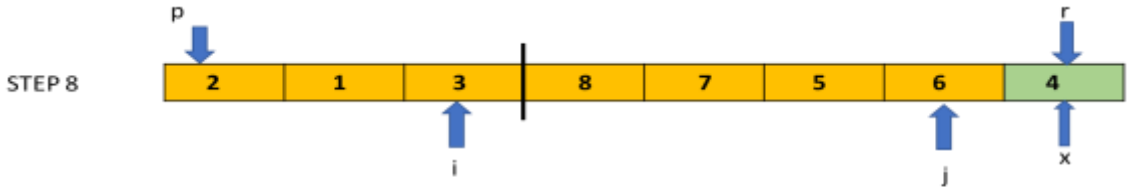
for j=4 1<=4 True so i increments by one, i=2, exchange A[2] and A[4] i.e., A[i] and A[j]

STEP 6

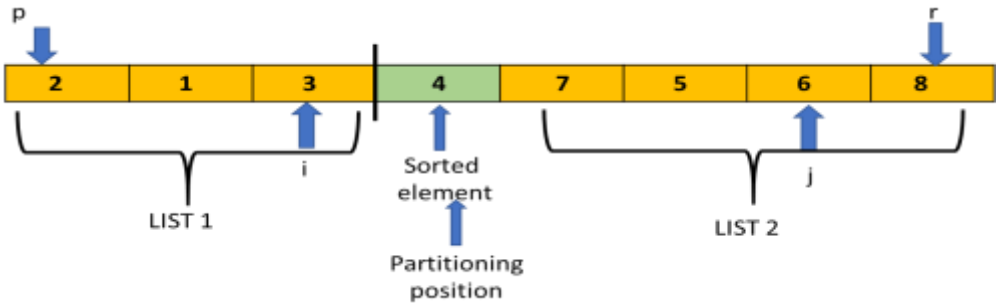


for j=5 3<=4 True so i increments by one, i=3, exchange A[3] and A[5] i.e., A[i] and A[j]

for j=6 5<=4 False so no change



for j=7 6<=4 False so no change
But since j has completed all iterations till r-1 i.e., 7 steps. Now , swap of A[4] and A[8] will be done i.e., A[i+1] and A[r]



After first partition call the index of i+1 will be returned which is 4 for the above example.

In same manner, QuickSort partition function will be called once for Array 1 and once for Array 2. For Array 1, QuickSort (A, 1, 3) and for Array 2, QuickSort(A, 5, 8) will be passed for partition function. The algorithm is: Considering p as the index of first element and r as the index of last element:

ALGORITHM QuickSort(A[ ], p, r)

BEGIN:

IF p<r

q = Partition (A, p, r)

QuickSort(A, p, q-1)

QuickSort(A, q+1, r)

END;

Step by step Complexity analysis of Partition Algorithm:

ALGORITHM Partition(A, p, r)

BEGIN:	**Cost**	**Time**
x = A[r].....	C1	1
i = p-1.....	C2	1
FOR j = p to r-1.....	C3	n+1
IF A[j] < = x.....	C4	n
i = i+1.....	C5	n
Exchange A[i] with A[j].....	C6	1
Exchange A[i+1] with A[r].....	C7	1
RETURN i+1.....	C8	1

END;

So, total running time calculation:

F (n)= C1.1 + C2.1 + C3.n+1 + C4.n + C5.n + C6.1 +C7.1+ C8.1

F (n)= n(C3+ C4 + C5) + 1(C1 +C2+ C6 +C7+C8)

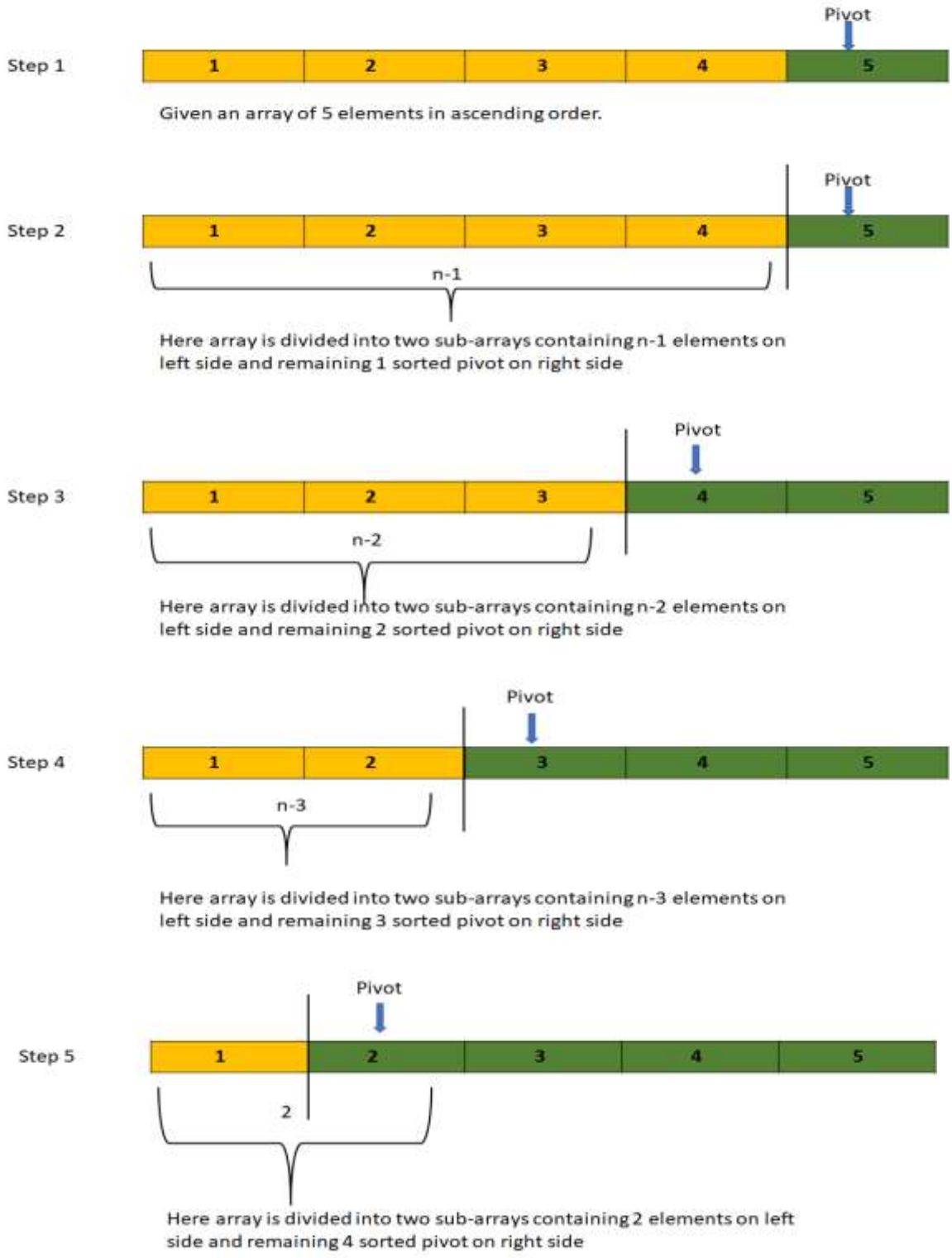
F (n) = a(n) +b

So, we can say that time complexity: θ(n) for partition algorithm. The space complexity of partition algorithm will be θ(1), as 5 extra variables are required.

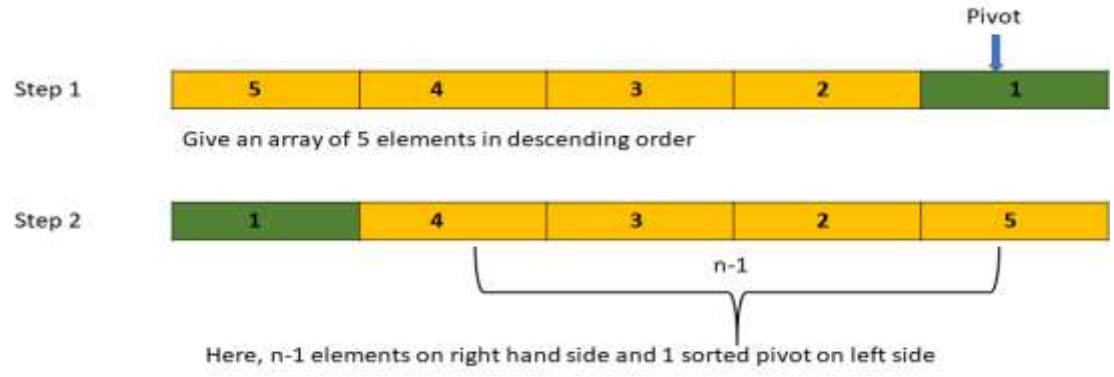
2.7.5.Detailed Complexity Analysis of Quick Sort with the example:

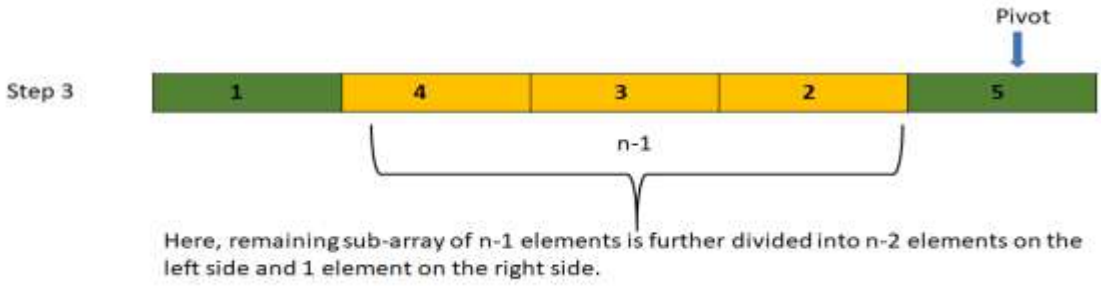
In the performance analysis of QuickSort selection of pivot element plays an important role. This can be classified as 3 cases in the form of input given.

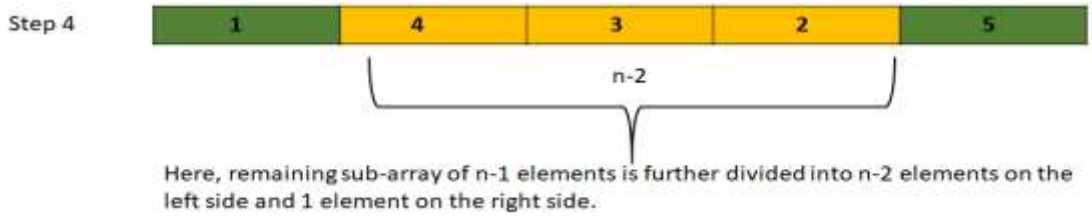
Case 1: a) When the input array is sorted in ascending order. Such a case experiences an unbalanced array split, with (n-1) elements on one side of an array and one as a sorted element (pivot element).

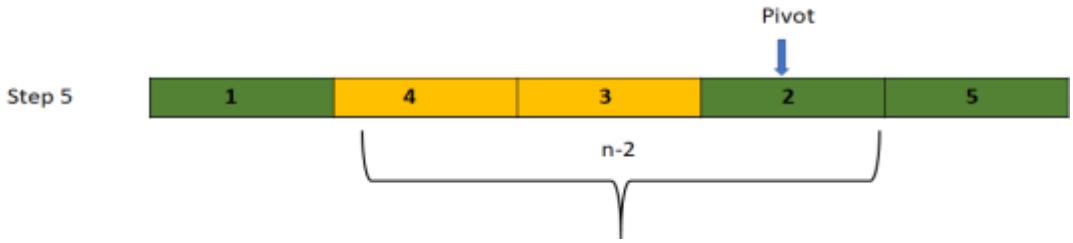


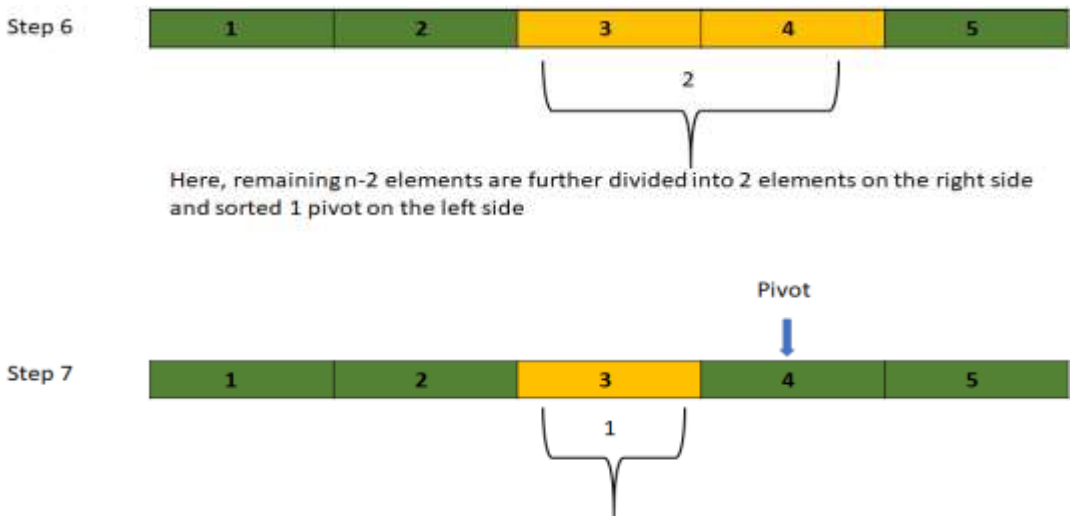
Case 1: b) When the input array is sorted in descending order. Given example below:









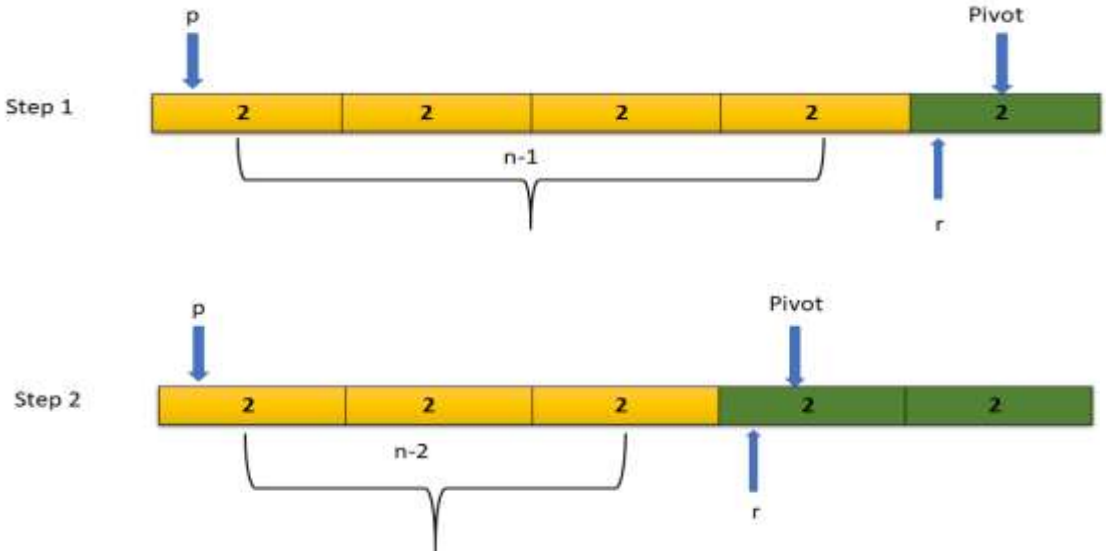


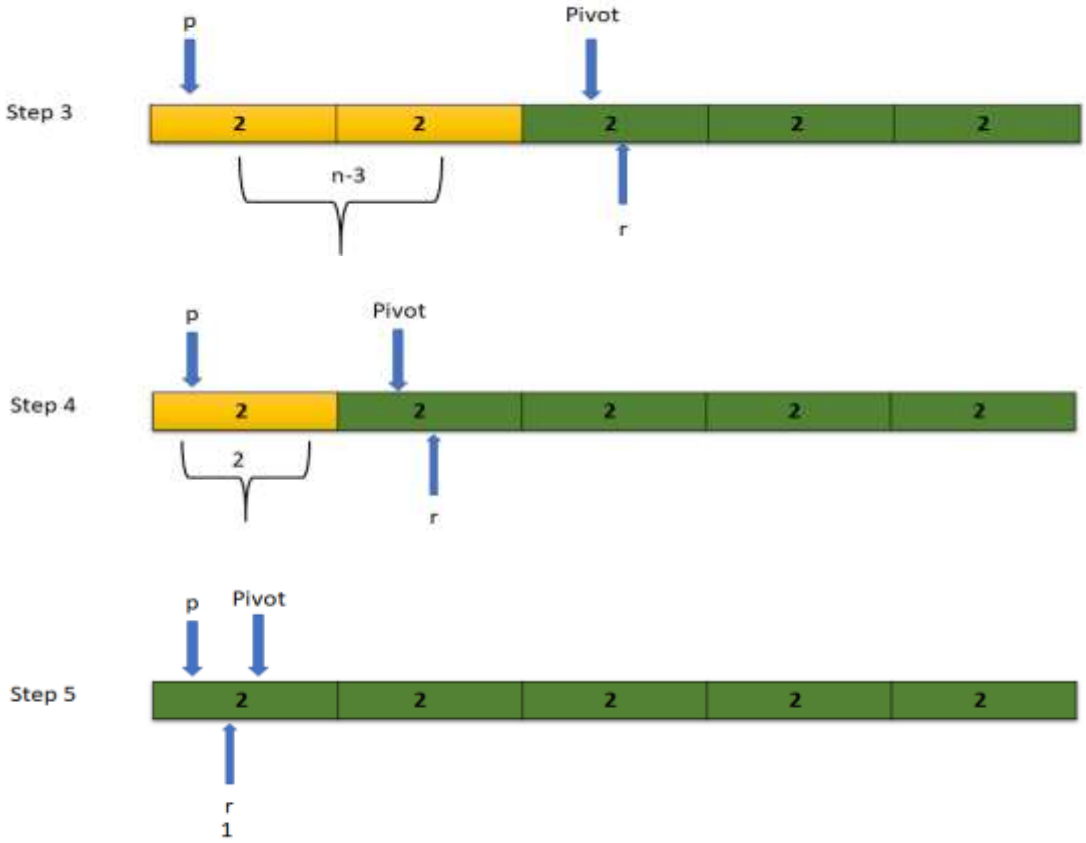
Here, worst case complexity of QuickSort can be explained either by the Recurrence relation or by iterative approach.

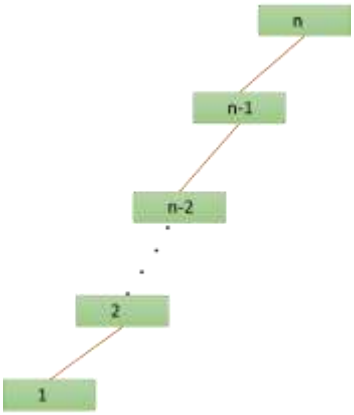
Step 1: Recurrence Method: Following Recurrence will be obtained for the unbalanced partitioning which occurs when the array is either in ascending order or descending order, that is, T(n) = T(n-1) + O(n). With the help of this recurrence we can simply say that running time complexity will be **$O(n^2)$.**

Step 2: Iteration Method: In this method, we will compute the cost at each level of the tree. For example, at 0^{th} level the cost is n, at 1^{st} level the cost is (n-1), and at 2^{nd} level the cost is (n-2) and so on. On the basis of this we derive the cost at all the levels of the tree, which is equal to **[n + (n-1) + (n-2) + (n-3) + …… +2] = (n(n+1)/2 -1] = $O(n^2)$ Worst case complexity.**

Case 1: c) When all elements in array are equal



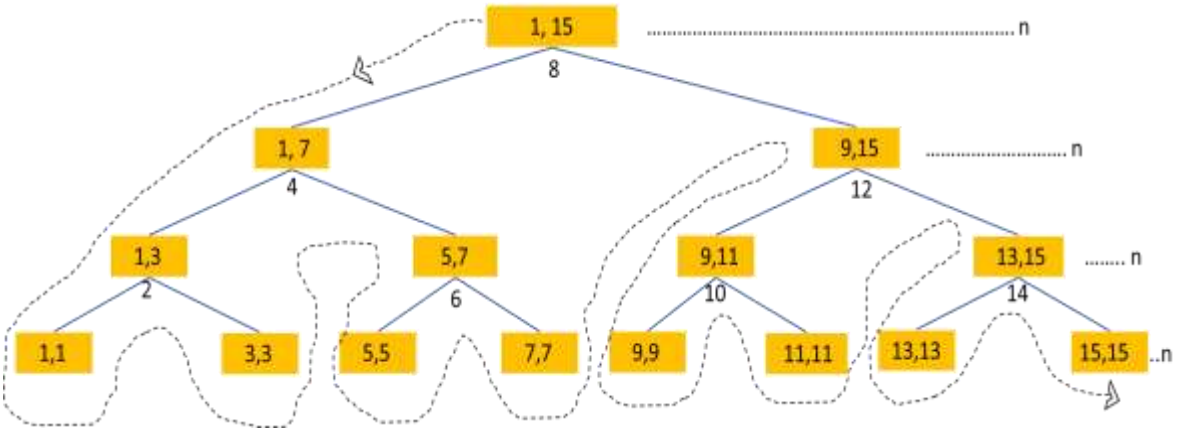




Thus, **[n + (n-1) + (n-2) + (n-3) + …… +1] = (n(n+1)/2 -1] = O(n^2) worst case complexity.**

Case 2: When the input array splits from the middle and partition occurs evenly from the middle at each level.

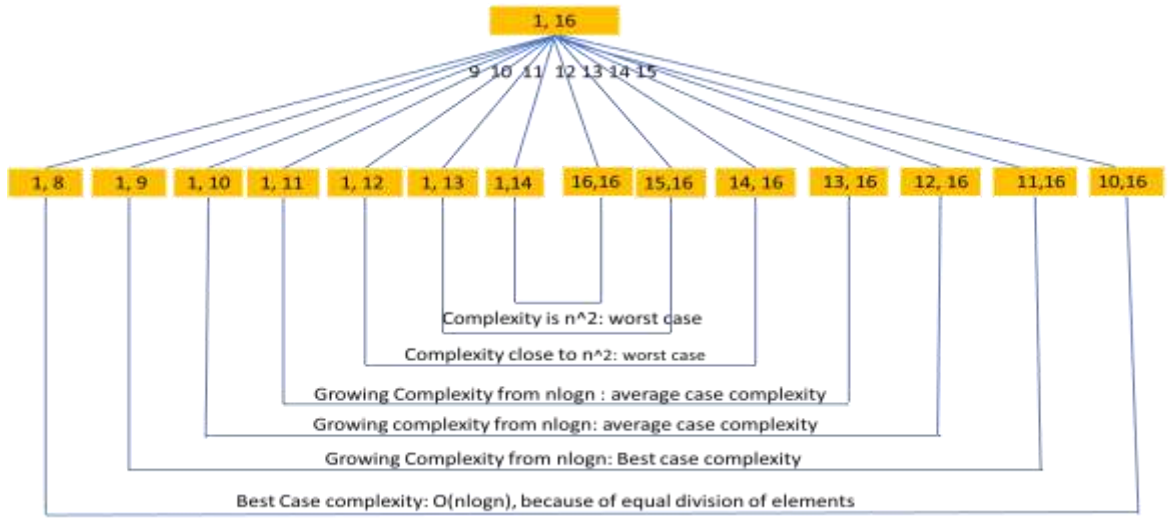
Suppose we have an array of 15 numbers A[1:15]. The best case is possible if:



At each and every level when input array is divided into two equal halves, so, we can simply say that cost at each level of the binary tree is n and the total number of levels is log n+1. Now, we can say that the running time complexity in the best case of quick sort will be equal to cost at each level multiplied by number of levels == n * (log n+1). Hence, we can conclude the complexity will be **Ω(nlog n). Recurrence derived from the above mentioned approach is as follows: T(n)= 2T(n/2) + Θ(n)** where 2T(n/2) is the cost for dividing the array into sub-arrays and n is the cost of merging the two sub-arrays. We can solve this Recurrence either by the Master Method or by Recursion Tree Method. After applying any of the method we can obtain best case complexity as O(nlog n).

Case 3: Average Case Complexity:

T(n)= 2T(n/2) + Θ(n)where 2T(n/2) is the cost for dividing the array into sub-arrays and n is the cost of merging the two sub-arrays. Thus, best case complexity is O(nlog n). In worst case pivot element divides the n element array in such a way that one element remains on the left and remaining n-1 elements on right side or vice-versa. On the basis of this we generate a recurrence as**T(n)= T(n-1) + Θ(n),** resulting in a time-complexity as $O(n^2)$**.** Now, for the average case, the blended mode of both best and worst must be met in array splitting. So, if the partitioning lies somewhere between best and worst case in all levels of n comparisons.



On the basis of this we generate the recurrence for the average case as T(n)= T(n/3) + T(2n/3) + θ(n) which results in the complexity as order of (nlog n). This is only one of the recurrences generated for computing the average case complexity. There may be various other recurrences which can symbolise the average case complexity of a quick sort, such as, T(n)= T(n/4) + T(3n/4) + θ(n), T(n)= T(n/5) + T(4n/5) + θ (n), T(n)= T(n/6) + T(5n/6) + θ (n) and so on.

Let us suppose the problem size n=36. By applying QuickSort algorithm we can partition these 36 elements into 3 ways on the basis of best case, worst case and average case. In best case, the equal elements are available on both the sides. In worst case, maximum elements appear from one side, whereas in average case, the number of elements that appear on the left side may range from 8-16 and number of element that appear on right may range from 20-28 and vice-versa. On the basis of this division of elements on left and right hand side we can derive various types of Recurrences which reflects the average case complexity of the Quick Sort.

The auxiliary space required by the Quick Sort Algorithm is O (n) for the call stack in the worst case.

2.7.6.Various Approaches to boost the performance of Quick Sort:

1. **Better Selection of the Pivot Element:**
   Selecting the pivot element is one of the most important operations of the quick sort algorithm. Generally, we select the first or last element as the pivot element in the quick sort which leads to the worst case behavior in a sorted array or closely sorted array. We can also solve the problem by selecting any random element as a pivot (Randomized Quick Sort) or we can also solve the problem by taking the median of the first, medium and last element for the pivot partitioning.
2. **Tail Recursion:**
   In order to make sure that O (n log n) space is used. Recur first array into the partition smaller side, and then use the tail recursion into the other. Here, we sort the array in such a way that we reach the minimum recursive depth.
3. **Hybrid with Insertion Sort:**
   In this, threshold value K is set on the basis of size of an array. This threshold value is set in order to define up to which value of k Insertion sort is used.
   Insertion sort is used when whole array is processed; each element is almost k positions away from the sorted array position. Now, if we perform the Insertion sort on it, it will take O(K.n) time to finish the sort which is linear.

2.7.7. Detailed Variants of Quick Sort:

1. Three way Quick Sort : In Three Way Quick Sort, Array A= [1….n] is divided in 3 parts: a) Sub-array [1…..i] elements less than pivot. b) Sub-array [i+1…..j] elements equal to pivot. c) Sub-array [j+1…..r] elements greater than pivot. Running Time Complexity is θ(n) and Space Complexity is θ(1).

Partitioning Algorithm of Three Way Quick Sort:

```
ALGORITHM Partition (arr[ ], left, right, i, j)
BEGIN:
        IF right – left <= 1THEN
          If arr[right] <arr[left] THEN
              Swap arr[right] and arr[left]
          i = left
          j = right
          RETURN

          Mid = left
          pivot = arr[right]
          WHILE mid <= right DO
                    IF arr[mid] < pivot THEN
                              Swap arr[left], arr[mid]
                              Left=left+1
                              Mid=Mid+1
                    ELSE
                              IF arr[mid] = pivot THEN
                                        mid=mid+1
                              ELSE
                                        Swap arr[mid] and arr[right]
                                        right =right - 1

         i = left – 1
          j = mid
END;
```

Three Way Quick Sort Algorithms:

```
ALGORITHM QuickSort (arr[ ], left, right)
BEGIN:
          IF left >= right THEN
                    RETURN
            Define i and j
```

```
        Partition(arr, left, right, i, j)
        Quicksort(arr, left, i)
        Quicksort(arr, j, right)
END;
```

2. Hybrid Quick Sort :

Hybrid Quick Sort is basically the combination of the Quick sort and Insertion sort Algorithm.

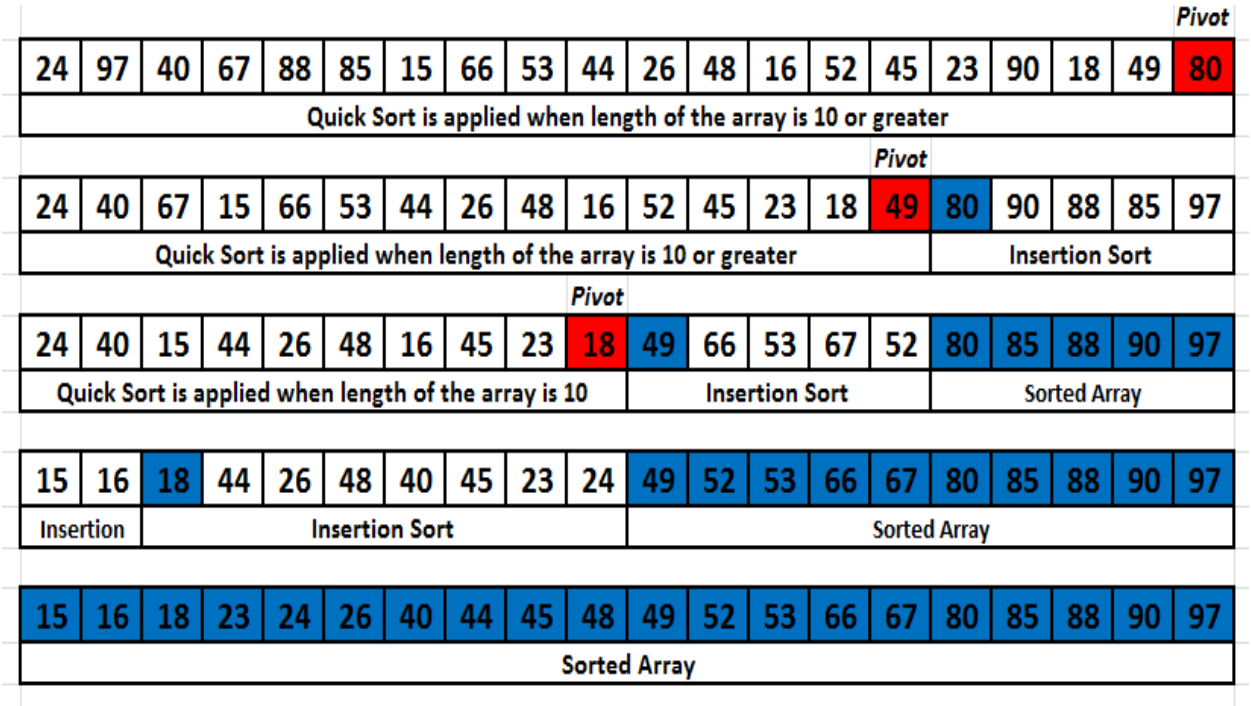
Why we use Hybrid algorithm:

Quick sort is one of the most efficient sorting algorithms when the size of the input array is large. Insertion sort is more efficient than quick sort when the size of array is small and number of comparisons and number of swaps is less in comparison to quick sort. Hence, we combine two approaches together in order to sort the array efficiently.

Note: Selection sort algorithm can also be used to combine with Quick Sort.

Example of Hybrid Quick Sort:

Array = {24, 97, 40, 67, 88, 85, 15, 66, 53, 44, 26, 48, 16, 52, 45, 23, 90, 18, 49, 80}



Approach of Hybrid Quick Sort:

1. The idea is to use recursion and continuously find the size of the array.

2. If the size is greater than the threshold value (10), then the quicksort function is called for that portion of the array.
3. If the size is less than the threshold value (10), then the Insertion Sort function is called for that portion of the array.

3. Median of an unsorted array using Quick Select Algorithm:

Problem Statement: We are given an unsorted array of length **N**; our objective is to find the median of this array.

Examples:
1.Input: Array [] = {12, 3, 5, 7, 4, 19, and 26}
Output: 7
The size of an array is 7. Hence the median element will be one that is available at 4^{th} Position i.e. 7.

2. Input: Array [] = {12, 3, 5, 7, 4, and 26}
Output: 6
The Size of an array is 6. Hence the median element will be the average of the elements available at 3^{rd} and 4^{th} Position i.e. 5+7/2= 12/2=6/

Naive Approach:
1. Sort the array in increasing order.

2. If number of elements in **array** is odd, then median is **array [n/2]**.

3.If the number of elements in **array** is even, median is **average of array [n/2] and array [n/2+1]**.

Randomized Quick Select:
1. Randomly pick pivot element from **array** and the using the **partition step** from the quick sort algorithm arrange all the elements smaller than the pivot on its left and the elements greater than it on its right.

2. If after the previous step, the position of the chosen pivot is the middle of the array then it is the required median of the given array.

3. If the position is before the middle element then repeat the step for the sub-array starting from previous starting index and the chosen pivot as the ending index.

4. If the position is after the middle element then repeat the step for the sub-array starting from the chosen pivot and ending at the previous ending index.

5. In case of even number of elements, the middle two elements have to be found and their average will be the median of the array.

Best case analysis: O(1)

Average case analysis: O(N)

Worst case analysis: $O(N^2)$

4. Randomized Quick Sort:

In this, Pivot Element can be chosen at random in randomized quick sort.

The new partitioning procedure simply implemented the swap before actually partitioning.

ALGORITHM RandomizedPartitioning (A[ ], p, r)

BEGIN:

```
i = Random(p, r)
Exchange A[p] with A[i]
RETURN PARTITION(A, p, r)
```

END;

Now Randomized Quick Sort will call the above procedure in place of PARTITION

Considering p as the First element and r as the last element:

ALGORITHM RandomizedQuickSort (A[ ], p, r)

BEGIN:

```
IF p < r THEN
        q = RandomizedPartition (A, p, r)
RandomizedQuick Sort (A, p, q-1)
RandomizedQuick Sort (A, q+1, r)
```

END;

This algorithm is used for selecting any random element as the pivot element.

Randomized Quick Sort has the expected running time complexity as θ (n log n) but in the worst case the time complexity will remain as same.

2.7.8. Quick Sort Animation Link:

1. Quick Sort | GeeksforGeeks - YouTube
2. https://www.youtube.com/watch?v=tIYMCYooo3c
3. https://www.youtube.com/watch?v=aXXWXz5rF64

2.7.9. Coding Problems related Quick Sort:

1. Eating apples:

Problem: You are staying at point (1, 1) in a matrix $10^9 \times 10^9$. You can travel by following these steps:

1. If the row where you are staying has 1 or more apples, then you can go to another end of the row and can go up.
2. Otherwise, you can go up.

The matrix contains N apples. The ith apple is at point (xi, yi). You can eat these apples while traveling. For each of the apples, your task is to determine the number of apples that have been eaten before.Practice Problem (hackerearth.com)

2. Specialty of a sequence:

Problem: You are given a sequence A of length n and a number k. A number A[l] is special if there exists a contiguous sub-array that contains exactly k numbers that are strictly greater than A[l]. The specialty of a sequence is the sum of special numbers that are available in the sequence. Your task is to determine the specialty of the provided sequence.
Practice Problem (hackerearth.com)

3. Noor and his pond:

Problem: Noor is going fish farming. There are N types of fish. Each type of fish has size(S) and eating Factor(E). A fish with eating factor of E, will eat all the fish of size ≤E.Help Noor to select a set of fish such that the size of the set is maximized as well as they do not eat each other.

Practice Problem (hackerearth.com)

4. Card game:

Problem: Two friends decided to play a very exciting online card game. At the beginning of this game, each player gets a deck of cards, in which each card has some strength assigned. After that, each player picks random card from his deck and them compare strengths of picked cards. The player who picked card with larger strength wins. There is no winner in case both players picked cards with equal strength.First friend got a deck with *n* cards. The *i*-th his card has strength ai. Second friend got a deck with *m* cards. The *i*-th his card has strength bi.

First friend wants to win very much. So he decided to improve his cards. He can increase by *1* the strength of any card for *1* dollar. Any card can be improved as many times as he wants. The second friend can't improve his cards because he doesn't know about this possibility.

What is the minimum amount of money which the first player needs to guarantee a victory for himself?

Practice Problem (hackerearth.com)

5. Missing Number Problem: You are given an array *A*. You can decrement any element of the array by *1*. This operation can be repeated any number of times. A number is said to be missing if it is the smallest positive number which is a multiple of *2* that is not present in the array *A*. You have to find the maximum missing number after all possible decrements of the elements.

Practice Problem (hackerearth.com)

2.7.10. GATE Objective questions on Quick Sort:

1.	In Quicksort while sorting integers in increasing order, if a1 and a2 are the comparison count for the provided inputs {6, 7, 8, 9, 10} and {9, 6, 10, 8, 7}. What situation will hold from the following, when pivot is the first element of the array.**GATE 2014**
A	a1 = 5
B	a1 < a2
C	a1 > a2
D	a1 = a2
ANS	**C**
2.	What is recurrence for worst case of Quick Sort and state it's time complexity in worst case.
A	Recurrence is T(n) = T(n-2) + O(n) and time complexity is $O(n^2)$
B	Recurrence is T(n) = T(n-1) + O(n) and time complexity is $O(n^2)$
C	Recurrence is T(n) = 2T(n/2) + O(n) and time complexity is O(nlog n)
D	Recurrence is T(n) = T(n/10) + T(9n/10) + O(n) and time complexity is O(nlog n)
ANS	**B**

3.	Suppose we have a O(n) time algorithm that finds median of an unsorted array. Now consider a QuickSort implementation where first median is found using above algorithm, then use median as pivot. What will be the worst case time complexity of this modified Quick Sort
A	$O(n^2 \log n)$
B	$O(n^2)$
C	O(n log nlog n)
D	O(nlog n)
ANS	**D**

4.	From the given sorting which is not a stable sort in its typical implementation?
A	Insertion Sort
B	Merge Sort
C	Quick Sort
D	Bubble Sort
ANS	**C**

5.	Suppose we are sorting an array of eight integers using QuickSort, after finishing just the first partitioning with array elements be like: { 3, 6, 2, 8, 10, 13, 12, 11}
A	The pivot could be either 8 or 10
B	The pivot could be 8, but not 10
C	The pivot is not 8, but could be 10
D	Neither 8 nor 10 is the pivot
ANS	**A**

6.	In QuickSort, for sorting n elements, the (n/4)th smallest element is selected as pivot using an O(n) time algorithm. What is the worst case time complexity of the QuickSort? (A)O(n) (B) O(nlog n) (C) $O(n^2)$ (D) $O(n^2 \log n)$ **GATE 2009**
A	A
B	B
C	C
D	D
ANS	**B**

7.	Consider QuickSort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let T(n) be number of comparisons required to sort n elements. **GATE 2008**
A	$T(n) <= 2T(n/5) + n$
B	$T(n) <= T(n/5) + T(4n/5) + n$
C	$T(n) <= 2T(4n/5) + n$
D	$T(n) <= 2T(n/2) + n$
ANS	**B**

8.	You have an array of n elements. Suppose you implement QuickSort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is: **GATE 2014**
A	$O(n^2)$
B	O(nlog n)
C	Θ(nlog n)
D	$O(n^3)$
ANS	**A**

9.	Randomized QuickSort is an extension of QuickSort where the pivot is chosen randomly. What is the worst case complexity of sorting n numbers using randomized QuickSort? **GATE 2001**
A	O(n)
B	O(nlog n)
C	$O(n^2)$
D	O(n!)
ANS	**C**

10.	Which of the following changes to typical QuickSort improves its performances on average and are generally done in practice? 1) Randomly picking up to make worst case less likely to occur. 2) Calling insertion sort for small sized arrays to reduce recursive calls. 3) QuickSort is tail recursive, so tail call optimizations can be done. 4) A linear time median searching algorithm is used to pick the median, so that the worst case time reduces to O(nlog n) **GATE 2015**
A	1 and 2
B	2, 3 and 4
C	1, 2 and 3
D	2, 3 and 4
ANS	**C**

11.	Which one of the following is the recurrence equation for the worst case time complexity of the QuickSort algorithm for sorting n(>=2) numbers? In the recurrence equations given in the options below, c is a constant **GATE 2015**
A	T(n)= 2T(n/2) +cn
B	T(n) = T(n-1) + T(0) + cn
C	T(n) = 2T(n-2) + cn
D	T(n) = T(n/2) + cn
ANS	**B**

12.	What is the best sorting algorithm to use for the elements in array are more than 1 million in general?
A	Merge Sort
B	Bubble Sort
C	Quick Sort
D	Insertion Sort
ANS	**C**

13.	QuickSort is run on 2 inputs shown below to sort in ascending order taking first element as pivot, 1) 1, 2, 3, ……., n 2) n, n-1, n-2, ……., 2, 1 Let C1 and C2 be the number of comparisons made for the inputs 1) and 2) respectively. Then, **GATE 1996**
A	C1<C2
B	C1>C2
C	C1=C2
D	We cannot say anything for arbitrary n
ANS	**C**

14.	Consider the following array 35 50 15 25 80 20 90 45.How the array looks like after the first pass if it applies a quick sort on the array.
A	25 20 15 35 80 50 90 45
B	25 15 20 35 45 50 80 90
C	15 20 45 35 25 50 80 90
D	25 20 15 35 80 50 90 45
ANS	**A**

15.	The recurrence relation to solve the Quick sort algorithm in the average case will be?
A	T(n) = 2T(n/2) + n
B	T(n) = T(n-1) + log n
C	T(n) = T(n/4) + n
D	T(n) = T(n-1) + 1
ANS	**A**

16.	The recurrence relation to solve the Quick sort algorithm in the worst case will be?
A	T(n) = T(n-1) + n
B	T(n) = T(n-1) + log n
C	T(n) = T(n/4) + n
D	T(n) = T(n-1) + 1
ANS	**A**

17.	How much time complexity will be required by the partition algorithm in one iteration to sort the element in the case of quick sort ?
A	O(n)
B	O(1)
C	O(log n)
D	O(n^2)
ANS	**A**

18.	When does the worst case occur in the quick sort algorithm?
A	Pivot element is randomly selected
B	Elements are randomly selected
C	All elements are the same
D	Pivot element is always the smallest or the largest element
ANS	**D**

19.	When does the best case occur in the quick sort algorithm?
A	Pivot is at ⅓ rd position
B	Pivot is randomly selected
C	Pivot is always the middle element
D	Pivot element is the smallest or the largest element
ANS	**C**

20.	When pivot is the middle element, we are able to reduce a problem of size N to two problems of size (N-1)/2 and (N-1)/2 results in ? (Considering quick sort)
A	Best case
B	Worst case
C	Average Case
D	None of the above
ANS	**A**
21.	When pivot is the smallest or largest element, we are able to reduce a problem of size N to two problems of size 1 and (N-1) results in? (Considering quick sort)
A	Best case
B	Worst case
C	Average Case
D	None of the above
ANS	**B**

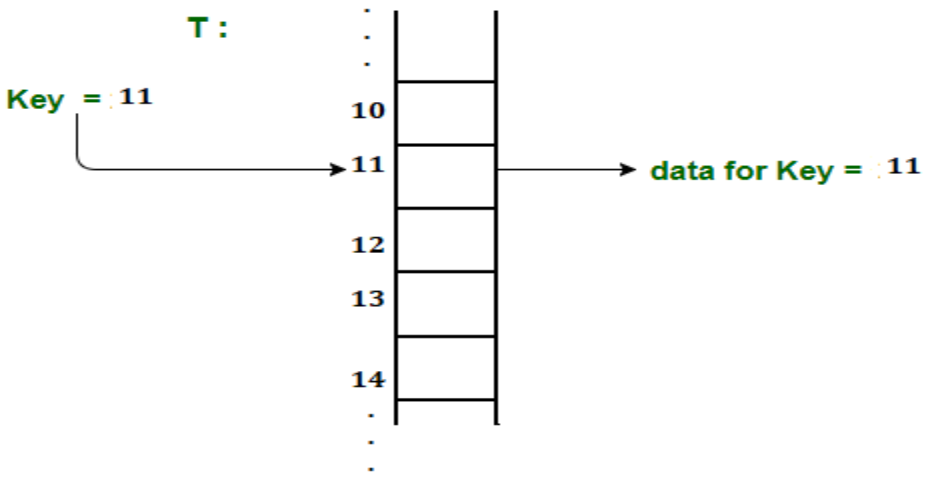
22.	To guarantee achieving the complexity of quick sort as O(n log n). Which method must be selected for picking the pivot element?
A	Choose the middle element
B	Choose the left most or the rightmost element
C	Median of three
D	Median of the medians
ANS	D

2.8 Counting Sort

Counting Sort, as the name suggests, must be counting something to sort the elements. The situation wherein we are working with the data set having a very short range, counting sort may be handy. Usually we deal with Direct address table while doing the sorting with the Counting Sort. Let us first see what the Direct Address Table is.

2.8.1.Direct Address Table

DAT is A data structure for mapping records to their corresponding keys using arrays. Records are placed using their key values directly as indexes.



Problem Statement: Given An array of integers (Range 1:10) , Find which of the elements are repeated and which are not.{4,3,1,2,5,7,1,6,3,2,4,1,8,10}

To solve this problem, let us take the Direct address table of size 10. Each of the elements in this table are initialized to zero.

1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0

For Input Key: 4 we will increase a count at that index.

1	2	3	4	5	6	7	8	9	10
			1						

For Input Key: 3 we will increase a count at that index.

1	2	3	4	5	6	7	8	9	10

		1	1						

For Input Key: 1 we will increase a count at that index.

1	2	3	4	5	6	7	8	9	10
1		1	1						

For Input Key: 2 we will increase a count at that index.

1	2	3	4	5	6	7	8	9	10
1	1	1	1						

For Input Key: Similarly we will get the Final Array as:

1	2	3	4	5	6	7	8	9	10
3	2	2	2	1	1	1	1	0	1

The values in the output array having values greater than 1 are the one having frequency more than one.

2.8.2.Explanation on Counting Sort:

So far all the sorting methods we have talked about works on the principle of comparing two numbers whether they are equal, smaller or larger. Counting sort algorithms rely solely on non-comparison approach. In counting sort basically works on the principle of counting the occurrence of the elements to be sorted. This sorting algorithm works faster having a complexity of linear time without making any comparison between two values. It is assumed the numbers we want to sort are in range from 1 to k where the value of k small. The main idea is to find the rank of each value in the final sorted array. Counting sort is not used frequently because there are some limitations that make this algorithm impractical in many applications. However if the input data is in small range then this algorithm has a distinct advantage. As it is the only algorithm that sorts the elements in order of (n) Complexity. This is also a stable sort algorithm. Many of the comparison sort algorithms sorts in quadratic time ($\theta(n^2)$), The exceptions – Merge, Heap and Quick Sort algorithms sorts elements in ($\theta(n \log n)$) time . We rarely use Counting Sort but if the requirements are fulfilled then it proves to be the best algorithm in choice.

2.8.3.Limitations:-

Positive Integers Only

Counting sort is an integer sort algorithm. For sorting we use the data values concurrently as indices and keys. There is a requirement that the objects or values or elements that we are sorting must be integers greater than 0 as they used to represent the index of an array.

Lesser Values

As the name suggests, counting sort utilizes a counting procedure. It is in the form of an auxiliary Array which stores the frequency count, i.e. number of occurrences, of each value. It is done by initializing an Array of 0s to accommodate the maximum input value. For example, if the input was the sequence 5 1 3, the count Array would need to accommodate an index 5, plus an additional element to represent the index 0.

Stable Sorting:- A sorting algorithm is stable if the relative order of elements with the same key value is preserved by the algorithm.

Example application of stable sort:- Assume that names have been sorted in alphabetical order. Now, if this list is sorted again by tutorial group number, a stable sort algorithm would ensure that all students in the same tutorial groups still appear in alphabetical order of their names.

Assumptions:

- Data size is n
- Each item contains keys and data
- The range of all keys is from 1 to k

Space

- The unsorted list is stored in A, the sorted list will be stored in an additional array B
- Uses an additional array C of size k

Input:

- A [1 .. n],

  $A[J] \in \{1,2, \ldots , k \}$

Output:

- B [1 .. n], sorted

- Uses C [1 .. k],
  auxiliary storage

ALGORITHM CountingSort(A[ ], B[ ], k)

BEGIN:

```
FOR  i = 0 TO k DO
    c[i] = 0
FOR j = 1 to length[A] DO
    C[A[j]] = C[A[j]] + 1
//C [i] now contains the number of elements equal to i
 FOR i = 1 TO k DO
   C[i] = C[i] + C[i-1]
//C [i] now contains the number of elements less than or equal to i
 FOR j = length[A]  to 1 STEP -1 DO
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[J]] – 1
```

END;

Running time of Counting Sort

The first for loop takes time Θ(k), the second for loop takes time Θ(n), third for loop takes time Θ(k), and the fourth for loop takes time Θ(n). Thus, the overall time is Θ(k+n). In practice, we usually use counting sort when we have k = O(n), in which case the running time is Θ(n).

Some of the Applications of counting sort algorithm are:

1. We can apply the counting sort to sort the data lexicographically whether the data may be in form of punch cards, words, integers, or mails.
2. We can apply counting sort in the area of parallel computing.
   a) We can also apply counting sort in the field of molecular biology, data compression and finding plagiarism it has the capabilities of finding redundant data
3. Radix sort is quite useful for very large in-memory sorts in a multiprocessor or cluster. Imagine that the entire dataset is in ram, distributed across a cluster. Each node sorts

locally, based on the first digit of the key, The nodes then exchange data (what in MPI is called an ALL TO ALL collective) so that the data is distributed across the machine but with the first digit in overall order. Then the nodes move to the next key digit. Repeat until done..

4. Suppose you want to implement medals tally counter country wise. Gold has more priority than silver and silver has more priority than bronze irrespective of the count of each of the individual type.

2.8.4.Example on Counting Sort:

Example: Illustration the operation of Counting Sort in the array.

A= (6,0,2,0,1,5,4,6,1,5,2)

Solution:

	1	2	3	4	5	6	7	8	9	10	11
A[1…n]	6	0	2	0	1	5	4	6	1	5	2

Here k=6 (largest number in A)

For i=0 TO 6 DO

C[i]=0 , i.e,

	0	1	2	3	4	5	6
C	0	0	0	0	0	0	0

FOR j=1 TO 11 DO

C[A[J]]] = C[A[J]]+1 // Array c contains the frequency of elements contained in array A. Now C[i] contains the number of elements equal to I //

	0	1	2	3	4	5	6
C	2	2	2	0	1	2	2

FOR i=1 TO 6 DO

C[i] = C[i]+C[i-1] // C [i] now contains the number of elements less than or equal to I //

	0	1	2	3	4	5	6
C	2	4	6	6	7	9	11

FOR j = 11 TO 1 STEP -1 DO

B[C[A[j]]] = A[j]

C[A[j]] = C[A[j]]-1

J	A[i]	C[A[j]]	B[C[A[j]]] = A[j]	C[A[j]] = C[A[j]]-1
11	2	6	B[6]=2	C[2]= 5
10	5	9	B[9]=5	C[5]=8
9	1	4	B[4]=1	C[1]=3
8	6	11	B[11]=6	C[6]=10
7	4	7	B[7]=4	C[4]=6
6	5	8	B[8]=5	C[5]=7
5	1	3	B[3]=1	C[1]=2
4	0	2	B[2]=0	C[0]=1
3	2	5	B[5]=2	C[2]=4
2	0	1	B[1]=0	C[0]=0
1	6	10	B[6]=6	C[6]=9

After execution of all iterations of the loop the Resultant

	1	2	3	4	5	6	7	8	9	10	11
Final Array B=	6	0	2	0	1	5	4	6	1	5	2

2.8.5..Objective Questions:

1	**Q1. How many comparisons will be made to sort the array arr={1, 5, 3, 8, 2} using counting sort?**
A	5
B	7
C	9
D	0
ANS	**D**

2	**Which of the following is not an example of non comparison sort?**
A	Bubble Sort
B	Counting Sort
C	Radix Sort
D	Bucket Sort
ANS	**A**

3	**Which of the following sorting techniques is most efficient if the range of input data is not significantly greater than a number of elements to be sorted?**
A	Bubble Sort
B	Counting Sort
C	Radix Sort
D	Bucket Sort
ANS	**B**

4	**What is the auxiliary space requirement of counting sort?**
A	O(1)
B	O(n)
C	O(log n)
D	O(n+k) where k is the range of input
ANS	**D**

5	**It is not possible to implement counting sort when any of the input element has negative value.**
A	TRUE
B	FALSE
ANS	**B**

6	If we use Radix sort to sort n integers in the range ($n^{k/2}$, n^k) for some value k>0 which is independent of n , the time taken would be :
A	O(n)
B	O(kn)
C	O(nlog n)
D	$O(n^2)$
ANS	**B**

7	Following algorithm can be used to sort n integers in the range [1…..n^3] in O(n) time :
A	Heap Sort
B	Quick Sort
C	Merge Sort
D	Radix Sort
ANS	**D**

2.8.6.Competitive Coding– Problem Statement-

Imagine a situation where heights of students are given. Your task is to find out any positive number n if possible, by using this number n all the given heights becomes equal by using any of these given operations

1) Adding number n in given heights, not necessary to add in all heights

2) Subtracting number n in given heights, not necessary to subtract in all heights

3) No operation perform on given heights

Example-

12 5 12 19 19 5 12

Let us suppose that value of n=7

If addition operation perform on 5 then = 5+7=12

If subtraction operation perform on 19 then = 19-7=12

Perform no operation on 12

Now heights becomes

12 12 12 12 12 12 12

Input format

First line contains heights of N students

Second line contains an integer n

Output format

Print single line height becomes equal or height not becomes equal

1-Identify problem statement

Read the story and try to convert it into technical form. For this problem reframes as-

Store the heights in an array and read a number n and try to perform given operations.

2-Identify Problem Constraints

1<N<100

Sample input sample output heights becomes equal

7

12 5 12 19 19 5 12

7

Design Logic

1 – Take an auxiliary array and store all the unique heights in this array.

2 – Count unique heights and store it in a variable c

```
IF c==1 THEN
        WRITE("Height becomes equal")
        RETURN 0
IF c==2 THEN
        WRITE("Height becomes equal")
        RETURN h1-h2
  //two unique heights let h1 and h2 and also h1>h2
IF c==3 THEN
        IF h3-h2==h2-h1 THEN          //h1<h2<h3
                WRITE("Height becomes equal")
                RETURN h3-h2
        ELSE
                WRITE("Height did not become equal")
                RETURN
```

Time Complexity-θ(n)

2.8.7.Un solved Coding problem:

1. https://www.hackerearth.com/practice/algorithms/sorting/counting-sort/practice-problems/algorithm/shil-and-birthday-present/

2. https://www.hackerearth.com/practice/algorithms/sorting/counting-sort/practice-problems/algorithm/finding-pairs-4/

2.9.RADIX SORT

The major problem with counting sort is that when the range of key elements is very high it does not work efficiently as we have to increase the size of auxiliary array and sorting time is high. In such input, Radix sort proves to be the better choice to sort elements in linear time. In Radix Sort we used to sort every digit hence the complexity is O(nd). This algorithm is fastest and most efficient when we talk about linear time sorting Algorithms. It was basically developed to sort large range integers.

2.9.1Analogy:- If you want to sort the 32 bit numbers, then the most efficient algorithm will be Radix Sort.

2.9.2.Radix Sort Algorithm

ALGORITHM Radix Sort (A[ ], N, d)

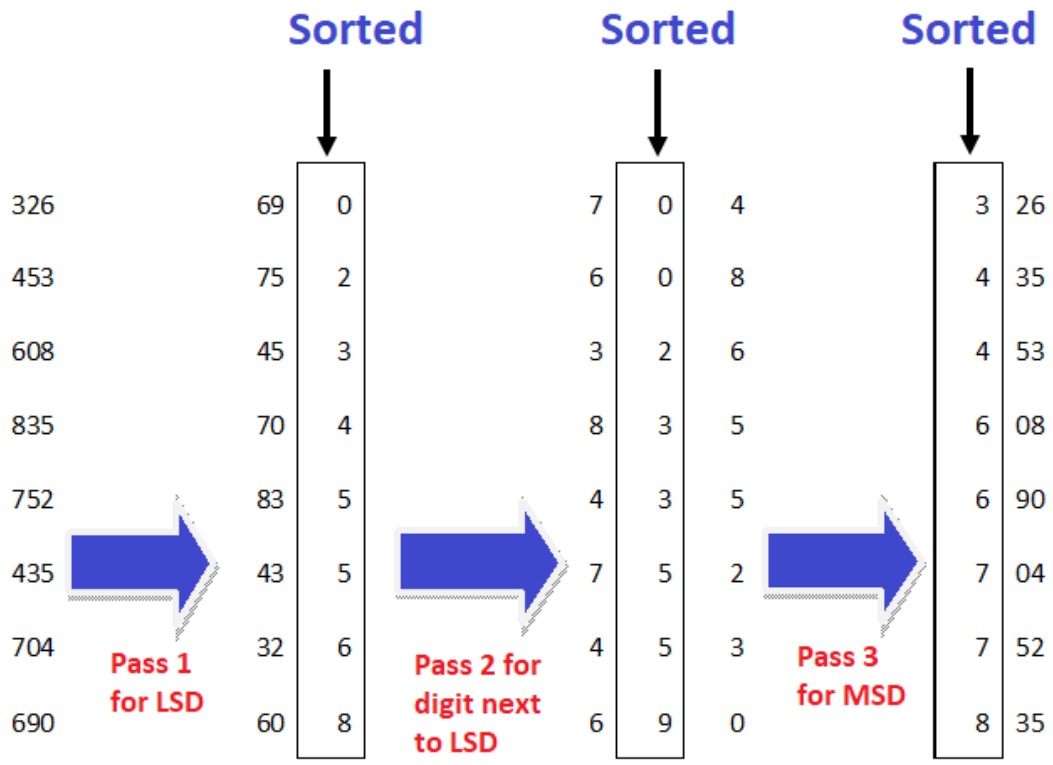
```
BEGIN:
    FOR i=1 TO d DO
        Perform Counting Sort on A at Radix i
END;
```

Running Time Complexity of Radix Sort:

Radix Sort is a linear sorting algorithm. The running time complexity of Radix Sort is O(nd), Here n represents the elements of input array and the number of digits is represented by d. This sorting algorithm uses an auxiliary array for the purpose of sorting that's why it's not a In place sorting algorithm. Radix Sort is a stable sort as the relative order of elements with equal values is maintained. When the applied operations are not efficient Radix sort works slower as compared to other sorting algorithms like quick sort and merge sort. These operations include insert and delete functions of the sub-list and the process of isolating the digits we want. One of the limitations with radix as compared to other sorting is its flexibility as it depends on the digits or letter. When the data is changed we have to rewrite the algorithm.

2.9.3.Explanation of Radix Sort with Example:

Observe the image given below carefully and try to visualize the concept of this algorithm.



Detailed Discussion on the above example:

1. **In First Iteration:** the least significant bit i. e the rightmost digit is sorted by applying counting sort. Notice that the value of 435 is below 835 and the least significant bit of both is equal so in the second list 435 will be below 835.
2. **In Second Iteration:** The sorting is done on the next digit (10s place) using counting sort. Here we can see that 608 is below 704, because the occurrence of 608 is below 704 in the previous list, and likely for (835, 435) and (752, 453).
3. **In third Iteration:** The sorting is done basis of the most significant digit (MSB) i.e 100s place using counting sort. Here we can see that here occurrence of 435 is below 453, because 435 occurred below 453 in the previous list, and similarly for (608, 690) and (704, 752).

2.9.4 Coding Problem on Radix Sort:

1. Problem Statement:

We have already studied about many sorting techniques such as Insertion sort, Bubble sort, Selection sort, Quick sort, Merge sort, Heap sort etc. Here I will talk about a different type of sorting technique which is called as "Radix Sort" and is probably the best sorting technique as far as time complexity is concerned.

Operations which are performed in Radix Sort is as follows:-

1) Do following for each digit **i** where **i** varies from least significant digit to the most significant digit.

a) Sort input array using counting sort (or any stable sort) according to the **ith** digit.

Example

Original Unsorted List 170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives:

[Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u>

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

8<u>0</u>2, 2, <u>2</u>4, <u>4</u>5, <u>6</u>6, 1<u>7</u>0, <u>7</u>5, <u>9</u>0

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, <u>1</u>70, <u>8</u>02

Hence we get a sorted sequence for the corresponding random sequence.

For a given set of N random numbers, generate a sorted (non-decreasing order) sequence using above discussed technique

2. **<u>Radix Sort | CodeChef:</u>** Given n strings, how to output their order after k phases of the radix sort

Problem

You are given n strings. Output their order after k phases of the radix sort.

Input

The first line of the input file contains three integer numbers: n, the number of strings, m, the length of each string, and k, the number of phases of the radix sort.

(1<=n<=106,1<=k<=m<=106,n∗m<=5∗107)

Then the description of the strings follows. The format is not trivial. The i-string (1<=i<=n) is written as the i^{th} symbols of the second, …, $(m+1)^{th}$ lines of the input file. In simple words, the strings are written vertically. **This is made intentionally to reduce the running time of your programs. If you construct the strings from the input lines in your program, you are doing the wrong thing.**

The strings consist of small Latin letters, from "a" to "z" inclusively. In the ASCII table, all these letters go in a row in the alphabetic order. The ASCII code of "a": 97, of "z" : 122.

Output

Print the indices of the strings in the order these strings appear after kk phases of the radix sort.

Example:

Input

3 3 1

bab

bba

baa

Output

2 3 1

In all examples the following strings are given:

- "bbb", with index 1;
- "aba", with index 2;
- "baa", with index 3.

Consider the first example. The first phase of the radix sort will sort the strings using their last symbol. As a result, the first string will be "aba" (index 2), then "baa" (index 3), then "bbb" (index 1). The answer is thus "2 3 1".

algorithms - Given n strings, how to output their order after k phases of the radix sort (huge constraints)? - Computer Science Stack Exchange

3. Descending Weights

Problem

You have been given an array *A* of size *N* and an integer *K*. This array consists of *N* integers ranging from *1* to 107. Each element in this array is said to have a **Special Weight**. The special weight of an element a[i] is a[i]%K.

You now need to sort this array in **Non-Increasing** order of the weight of each element, i.e the element with the highest weight should appear first, then the element with the second highest weight and so on. In case two elements have the same weight, the one with the lower value should appear in the output first.

Input Format:

The first line consists of two space separated integers *N* and *K*. The next line consists of *N* space separated integers denoting the elements of array *A*.

Output Format:

Print *N* space separated integers denoting the elements of the array in the order in which they are required.

Constraints:

$1 \leq N \leq 105$

$1 \leq A[i] \leq 107$

$1 \leq K \leq 107$

Note: You need to print the value of each element and not their weight.

Sample Input

5 2

1 2 3 4 5

Sample Output

1 3 5 2 4

2.10. Bucket Sort or Bin Sort

2.10.1 Analogy:

Bucket Sort can be best understood with the following Example-

1. **Super market product arrangement**- In this, we have to sort product according to product rack. All household products in same area, food items in same area, all cloth in same area. Then these areas are sorted in some order like if we take example of food area then all type of biscuits are sorted at same place, chocolates are at same place and so on. These areas are treated as bucket in bucket sort.
2. **Bar Chart- I**n bar chart, we set the range for the chart, like 0-10, 11-20, 21-30…… then we put the elements in these range bucket.

2.10.2.BUCKET SORT:

Bucket sort or Bin Sort is a sorting algorithm that works by distributing the elements of an array into different buckets uniformly. After distribution each bucket is sorted independently using any sorting algorithms or recursively or by recursively applying bucket sort on each bucket.

Why Bucket Sort?

1. After distribution of elements into bucket array size become smaller and can be solve each bucket independently.
2. Each bucket can be solved in parallel.
3. Bucket sort solve fractional numbers efficiently.
4. Bucket Sort is not in place sorting algorithm.

```
ALGORITHM Bucket Sort (arr [ ], n)          // n is length of array
BEGIN:
        Create n bucket with NULL value.    // initialize empty bucket
        FOR i=0 TO n-1 DO                   // start loop from first element to last element
            bucket[ n*array[i]] = arr[i]    // enter element into respective Bucket
        Sort each bucket using insertion sort or any other sort.
        Concatenate all sorted buckets.
END;
```

Concatenated Bucket is the sorted buckets

Complexity:

1. **Time Complexity:**
    a. Average Case complexity is θ(n)
    b. Best case complexity is Ω(n)
    c. Worst case time complexity is $O(n^2)$
2. **Space Complexity** of bucket sort is θ (n + k). n is number of elements and k is number of buckets.

Cases:

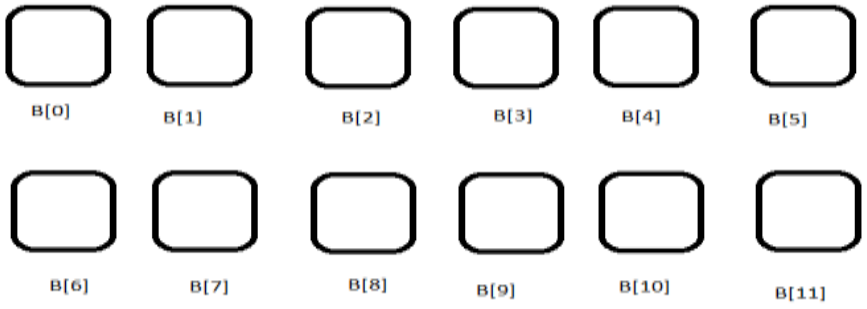
1. **Average case & Best Case:** when all element distribution is uniform in buckets**.**
2. **Worst Case:** when n or approximate n elements lie in one Bucket. Then it will work as insertion sort.

Example 1.

0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

n= length of array (12)

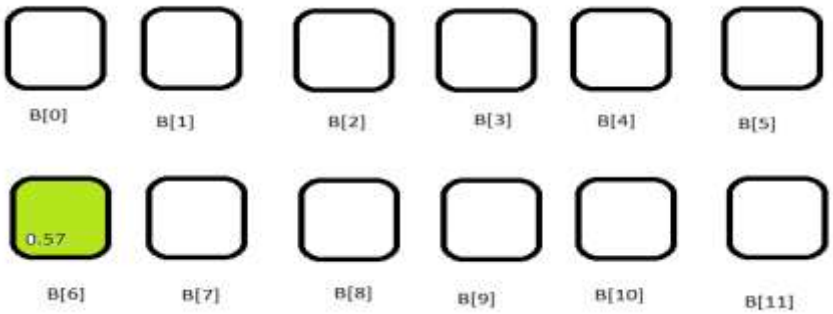
B[int n* a[i]]=a[i]



0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

1st Element B[int 12* 0.57] <- .57

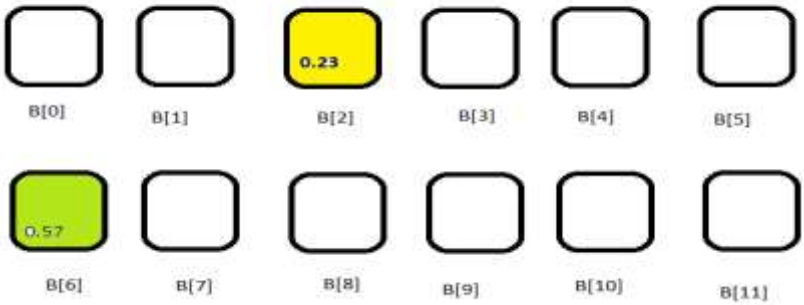
B[6] = 0.57



0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

2nd Element B[int 12*0.23] <- 0.23

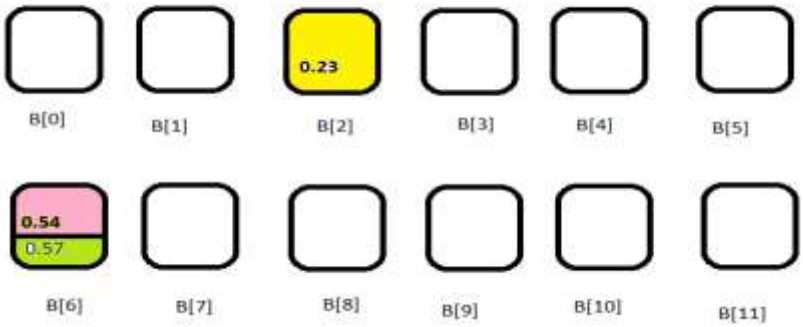
B[2] <- 0.23



0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

3rd Element B[int 12*0.54] <- 0.54

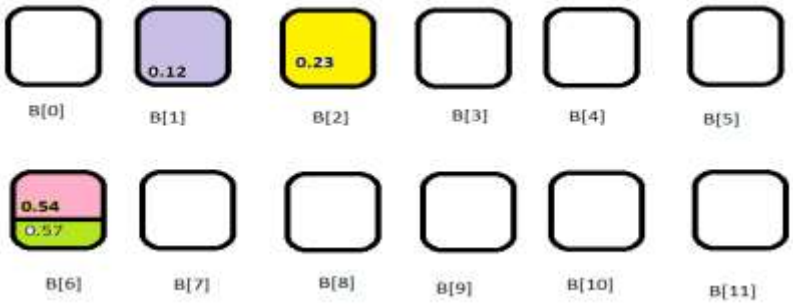
B[6] <- 0.54



0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

4th Element B[int 12*0.12] <- 0.12

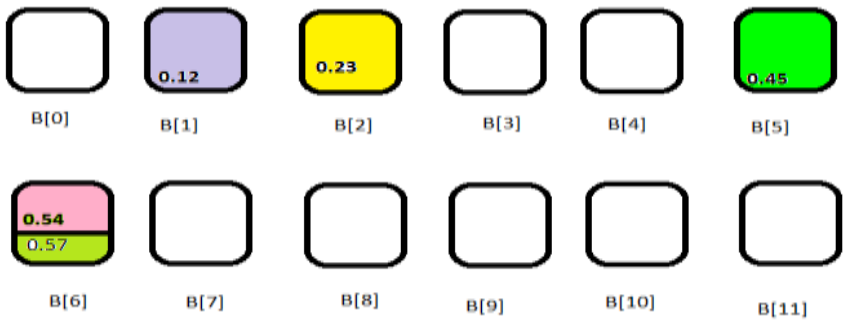
B[1]=0.12



0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

5th Element B[int 12*0.45] <- 0.45

B[5]=0.45



0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

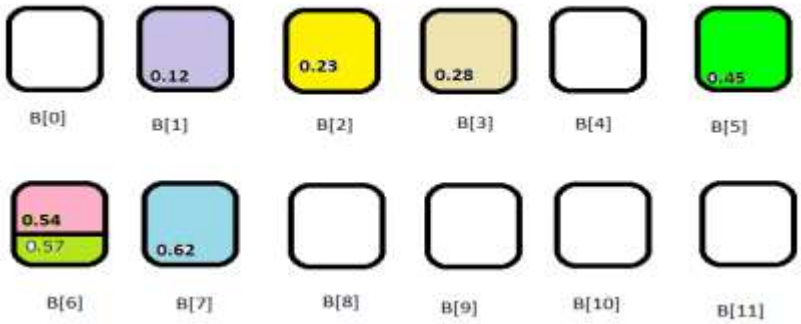
6th Element B[int 12*0.62] <- 0.62

B[7]=0.62

0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

7th Element B[int 12*0.28] <- 0.28

B[3]=0.28



0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

8th Element B[int 12*0.16] <- 0.16

B[1]=0.16

0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

9th Element B[int 12*0.49] <- 0.49

B[5]=0.49

0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

10th Element B[int 12*0.35] <- 0.35

B[4]=0.35

0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

11th Element B[int 12*0.01] <- 0.01

B[0]=0.01

0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9

12th Element B[int 12*0.9] <- 0.9

B[10]=0.9

Final sorted Array

0.01											

0.01	0.12										

0.01	0.12	0.16									

0.01	0.12	0.16	0.23								

0.01	0.12	0.16	0.23	0.28							

0.01	0.12	0.16	0.23	0.28	0.35						

0.01	0.12	0.16	0.23	0.28	0.35	0.45					

0.01	0.12	0.16	0.23	0.28	0.35	0.45	0.49				

0.01	0.12	0.16	0.23	0.28	0.35	0.45	0.49	0.54			

0.01	0.12	0.16	0.23	0.28	0.35	0.45	0.49	0.54	0.57		

0.01	0.12	0.16	0.23	0.28	0.35	0.45	0.49	0.54	0.57	0.62	

0.01	0.12	0.16	0.23	0.28	0.35	0.45	0.49	0.54	0.57	0.62	0.9

2.10.3.Cases of Bucket Sort

CASE1- NUMBER OF BUCKETS=2(Quick Sort Partition)

If number of buckets is two then we can use quick sort partition of elements.(best one) and after that apply insertion sort to sort the elements.

CASE2- NUMBER OF BUCKETS=n(Counting Sort)

If number of buckets is n then it becomes counting sort.

Algorithm 1–

1-Create array of pointers of size n

2-insert elements using link list but takes last pointer so that it takes constant time.

3-Sort individually each buckets using insertion sort.

4-finally concatenate in to original array.

Algorithm 2–

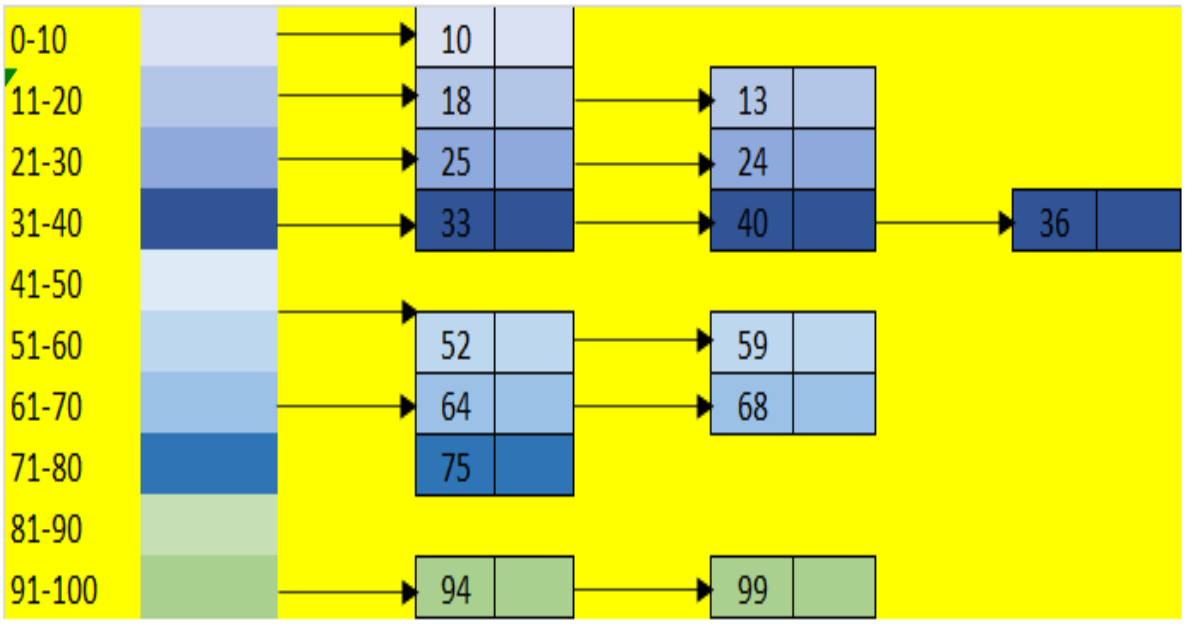
1-Create array of pointers of size n

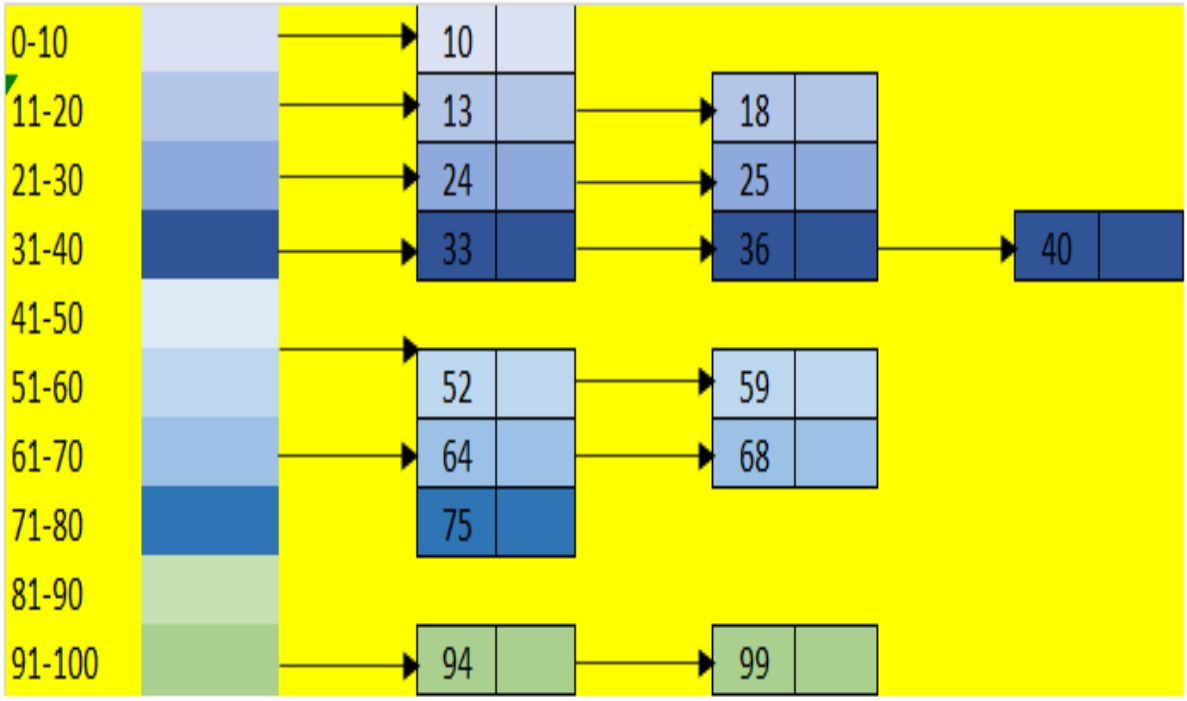
2-insert elements using link list but takes last pointer so that it takes constant time and keep in insertion happens in sorted order.

3-finally concatenate in to original array.

Example 2

Arr[]= 33,18,10,64,52,40,25,75,59,94,36,13,99,24,68 **number of element=10**





2.10.4.Bucket sort for integer numbers

Algorithm:

1. Find Max and min element of array.
2. Calculate the range of each bucket

   Range = (max+ min) / n // n is the number of bucket
3. Create n Buckets of Calculated Range
4. Distribute the elements in the buckets.
5. Bucket index=(arr[i] – min) / range
6. Now Sort each bucket individually.
7. Concatenate the sorted elements from buckets to original array.

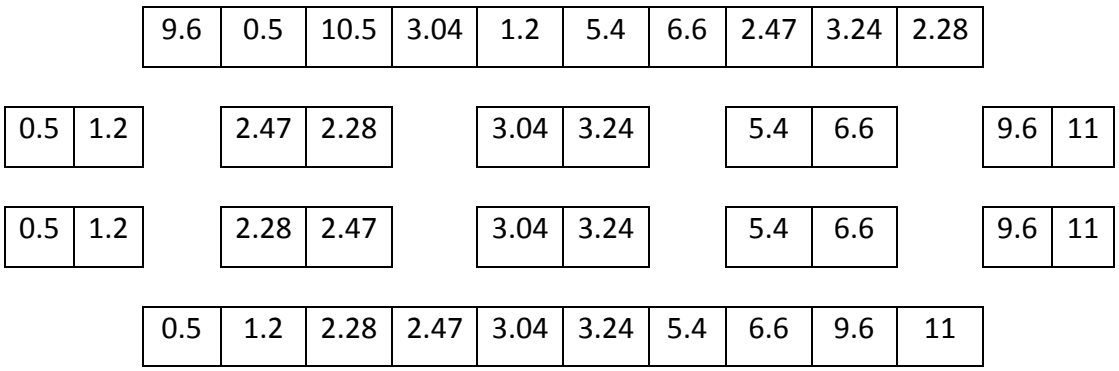
Example:

Input array 9.6, 0.5, 10.5, 3.04 , 1.2 , 5.4 , 8.6 , 2.47, 3.24 , 1.28 and number of bucket = 5

Max=10.5

Min=0.5

Range=(10.5-0.5)/5 = 2



2.10.5.Objective Type Questions:

1	If we use Radix Sort to sort n integers in the range ($n^{k/2}$, n^k), for some k>0 which is independent of n, the time taken would be. (GATE 2008)
A	O(n)
B	O(kn)
C	O(nlogn)
D	O(n^2)
ANS	**B**

2	How many comparisons will be made to sort the array arr={1, 5, 3, 8, 2} using bucket sort?
A	5
B	7
C	9
D	0
ANS	**D**

3	What is the alternate name of bucket sort?
A	Group sort
B	Radix Sort
C	Bin Sort
D	Uniform Sort
ANS	**C**

4	Which of the following non-comparison sort can also be considered as a comparison based sort?
A	Counting sort
B	MSD radix sot
C	bucket sort
D	pigeonhole sort
ANS	**C**

5	Which of the following is not true about bucket sort?
A	It is a non-comparison based integer sort
B	It is a distribution sort
C	can also be considered as comparison based sort
D	It is in place sorting algorithm
ANS	**D**

6	Which of the following don’t affect the time complexity of bucket sort?
A	Algorithm implemented for sorting individual buckets
B	number of buckets used
C	distribution of input
D	input values
ANS	**D**

7	Bucket sort is most efficient in the case when _____
A	the input is non-uniformly distributed
B	the input is uniformly distributed
C	the input is randomly distributed
D	the input range is large
ANS	**B**

8	Bucket sort is a generalization of which of the following sort?
A	LSD radix sort
B	Pigeonhole sort
C	Counting sort
D	MSD radix sort
ANS	**B**

9	What is the worst case time complexity of bucket sort (k = number of buckets)?
A	O(n + k)
B	O(n.k)
C	**$O(n^2)$**
D	O(n log n)
ANS	**C**

10	What is the best case time complexity of bucket sort (k = number of buckets)?
A	O(n + k)
B	O(n.k)
C	$O(n^2)$
D	O(n log n)
ANS	**A**

11	Which of the following is not necessarily a stable sorting algorithm?
A	bucket sort
B	counting sort
C	merge sort
D	pigeonhole sort
ANS	**A**

12	Bucket sort is an in place sorting algorithm.
A	True
B	False
ANS	**B**

13	What is the worst space complexity of bucket sort (k = number of buckets)?
A	O(n + k)
B	O(n.k)
C	$O(n^2)$
D	O(n log n)
ANS	**B**

2.10.6..Animated Link

:https://www.cs.usfca.edu/~galles/visualization/BucketSort.html