

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.

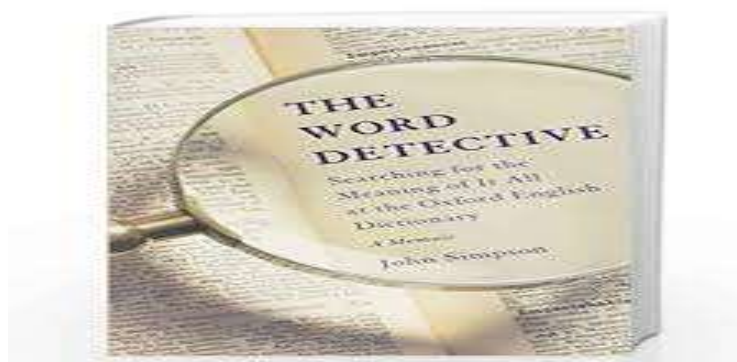


Figure 1.1: Representation of finding a word in a dictionary

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.

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.

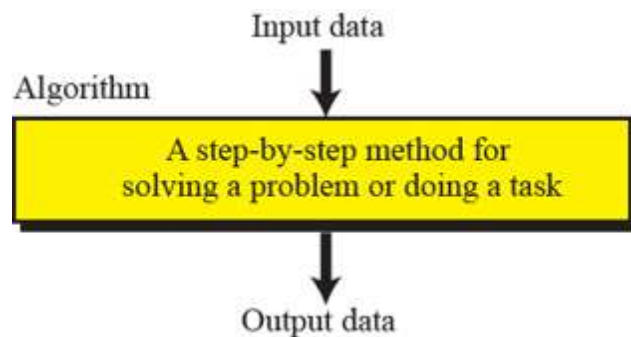


Figure 1.4 : Definition of Algorithm

Example 1: Let's understand an example of a person brushing his teeth in context of an algorithm. In this example, we have written an algorithm in natural language.

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 by considering an example for addition of two numbers.

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?

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.

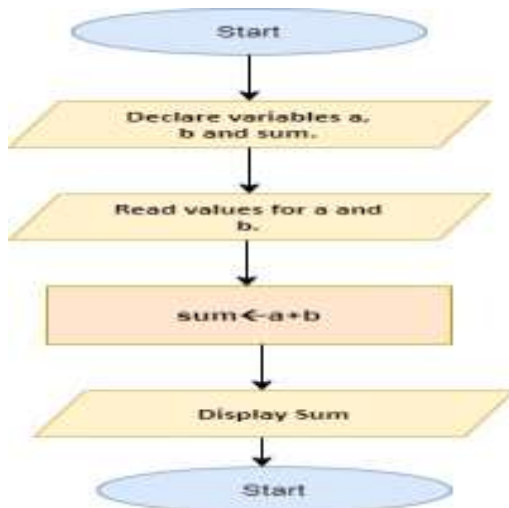
Step 4: Add a and b and assign the result to a variable sum.

Step 5: Display sum

Step 6: Stop

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.

Method 2. Algorithm in terms of 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.

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; }
```

4. **Pseudo code:** This method is the largest applicable one. It's very simple as its syntax less.

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.

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).

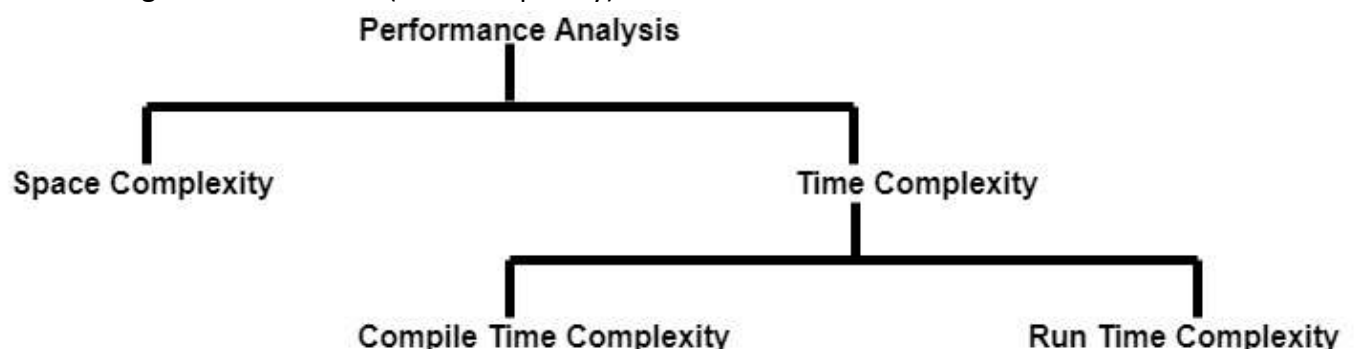


Figure 1.5: Analysis of Algorithm in terms of time and space complexity

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

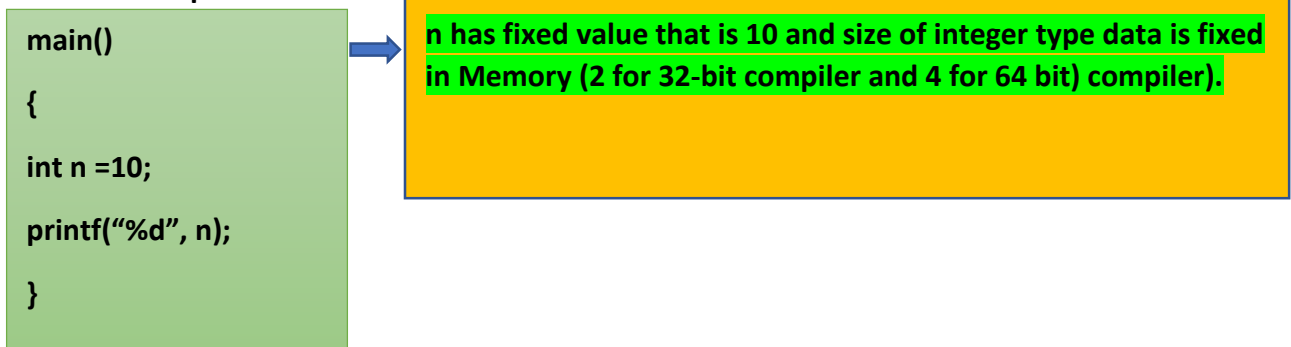
2.1.1 Space Complexity:

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-

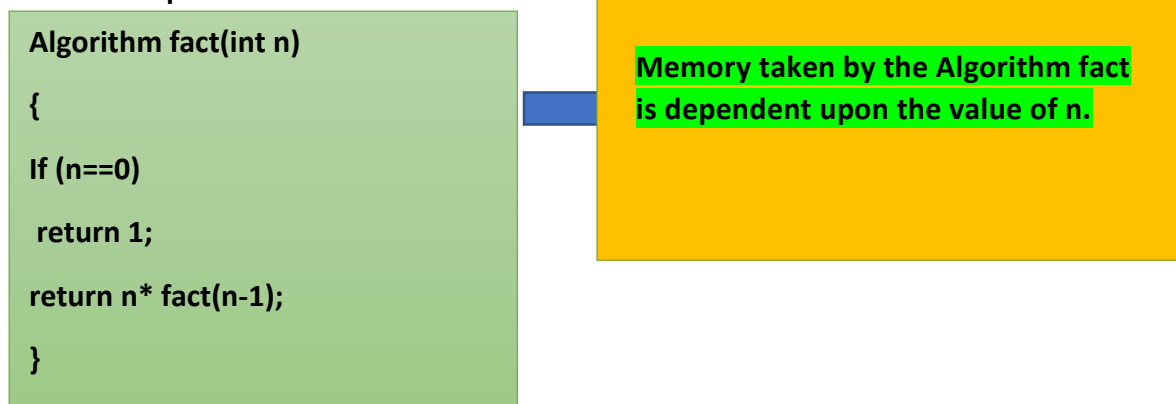
- i) **Fixed Part:** -fixed part is space for code this is independent of the input and output. This part includes the instruction space, space for simple variables, space for constant, so on.

Example:



- 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:



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

Statement	Step/execution (s/e)	Frequency	Total step
Algorithm sum(a,n)	0	1	0
{	0	1	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	1	0

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

Example 2: Write code for sum of n natural numbers.

Solution 2: For this problem we can write pseudo code in different ways:

Method 1:

```
int fun1(int n)
{
    int sum=0;
    for(int i=1; i<=n; i++)
        for(int j=1; j<=i; j++)
            sum++;
    return sum;
}
```

Time complexity =
order of n^2

Space complexity =
order of 1
(constant)

Time Complexity: outer loop(i loop) executes n times(1....n) and inner loop(j loop) executes n times. The complexity of code is $n(\text{outer loop}) * n(\text{inner loop}) = \text{order of } (n^2)$.

Space Complexity: it takes storage for n, sum, i and j. total 4 spaces in memory. No matter value of n is 1 or 100 or any value. Space is fixed for any value of n then the space complexity of above code is order of (1). Order of (1) is used for constant time complexity.

Method 2:

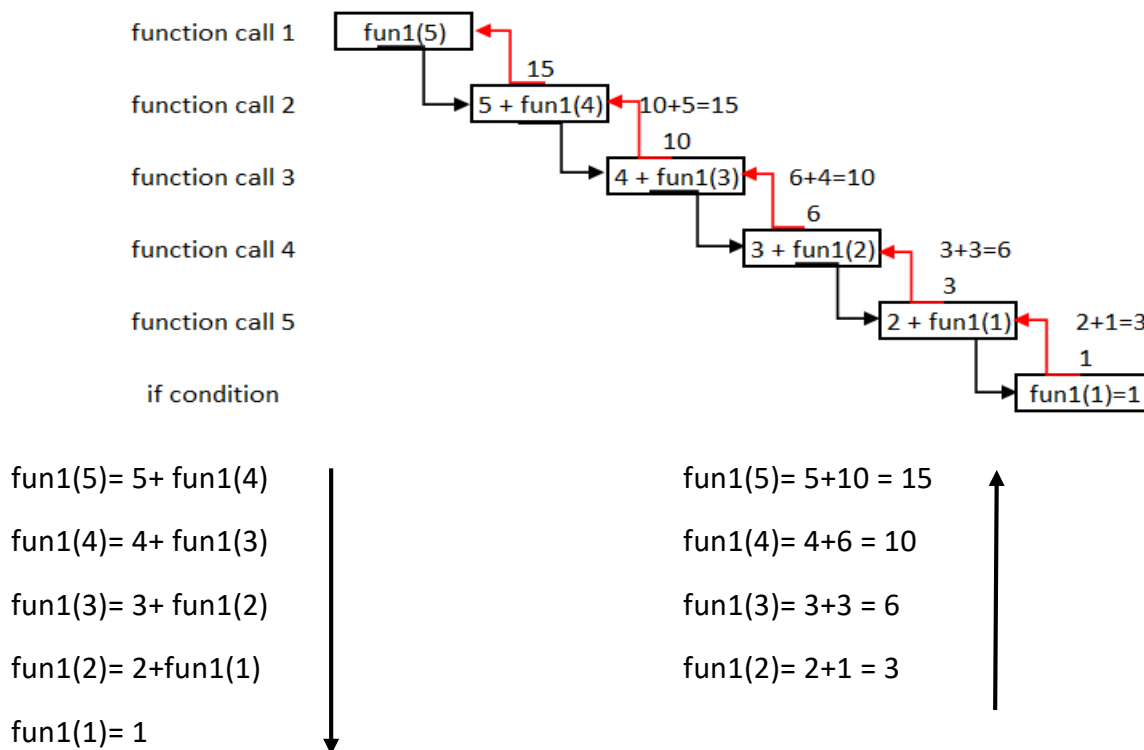
```
int sum = 0;
int fun1(n)
{
    if(n==1)
        return 1;
    sum = n+ fun1(n-1);
    return sum;
}
```

Time complexity =
order of n

Space complexity =
order of n

Let $n=5$ and answer should be $5 + 4 + 3 + 2 + 1 = 15$;

How function work

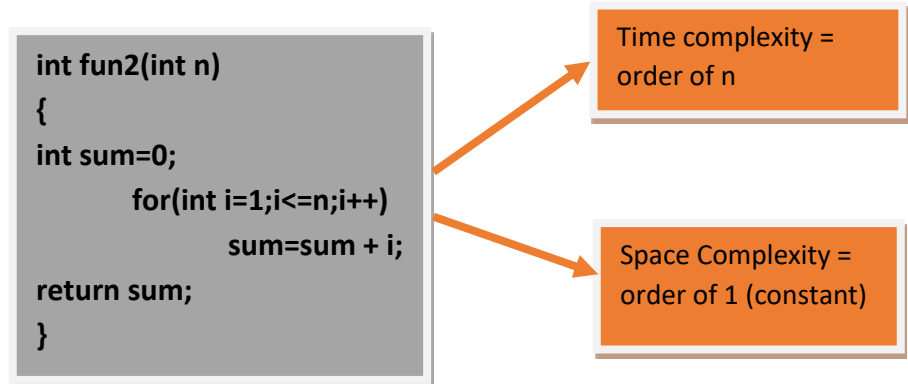


Time complexity: time complexity of above code is equal to number of functions calling
Time complexity= $5 (=n)$. order of n.

Space Complexity: For above function $(n+1)$ memory is required(n times for function calling and temporary storage of data generated by function call and 1 storage for value of n). So we can say that complexity of above code is order of $(n+1)$. In complexity we ignore constant terms or we take only highest power term.

Space complexity=order of (n).

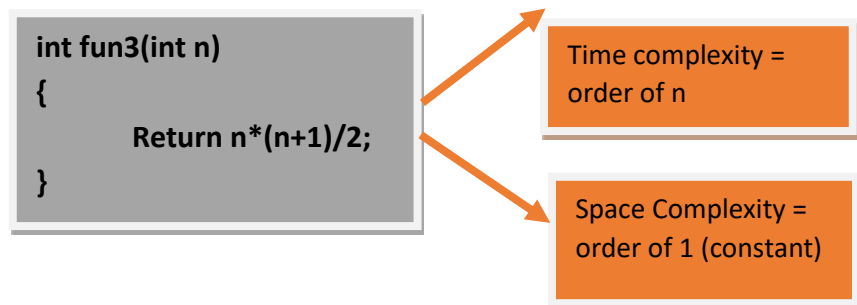
Method 3:



Time complexity: for loop executes n times (from 1 to n). time complexity is order of n.

Space complexity: Space required for value of n, sum and i (constant space). Space complexity is order of (1).

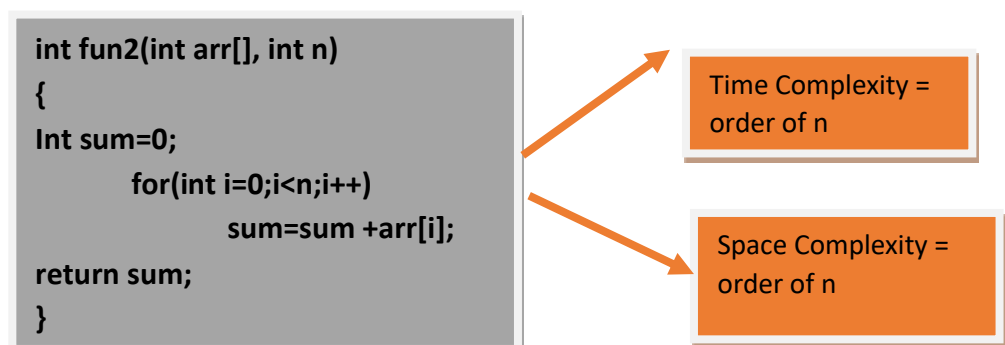
Method 4:



Time complexity: this code executes only 1 statement (**return n * (n+1)/2**). No loop no function calling. No matter what is the value of n every time only 1 statement will execute. The time complexity is order of (1). (For constant time execution)

Space complexity: it will take space only for value of n (constant). Space complexity of above code is order of (1).

Method 5:



Time complexity: for loop executes n times (from 1 to n). time complexity is order of n.

Space complexity: space required for value of n , integer type array of size n , sum and i (array size is variable so space required is also variable). Space complexity is order of (n) .

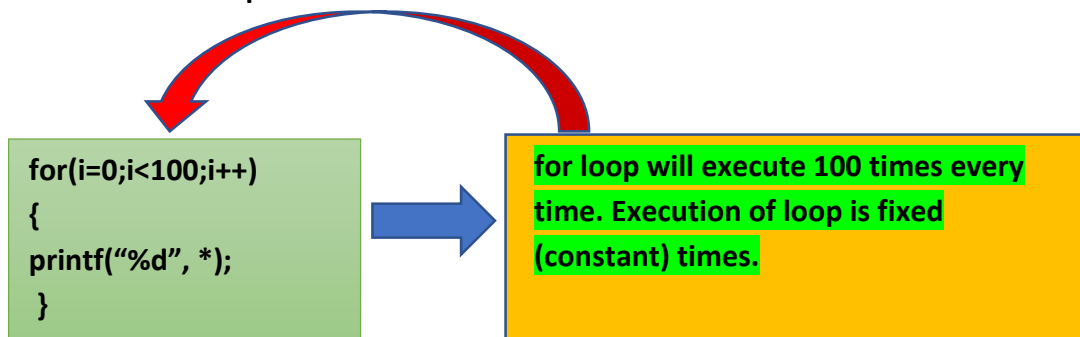
2.2. Common Mathematical Functions Used in Complexity Analysis:

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.

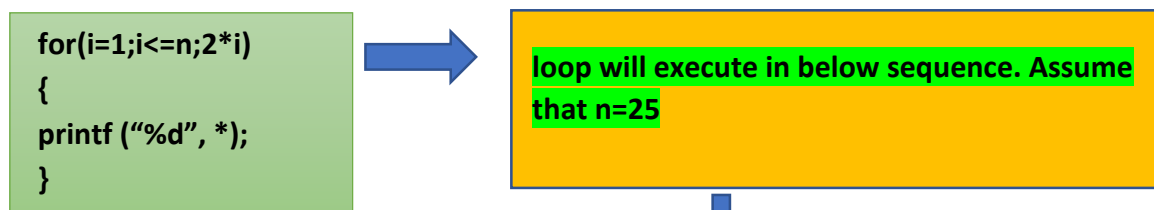
Let's take an example-



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:



1	2	4	8	16	32
True/ false	$2 < 25$ (true)	$4 < 25$ (true)	$8 < 25$ (true)	$16 < 25$ (true)	$32 > 25$ (false)

Now we can see that i increases in $2^1, 2^2, 2^3, \dots$ manner and the value of 2^i must be less or equal to n .

$$2^i \leq n$$

Take log both side

$i \log 2 \leq \log n$

$i \leq (\log n) / (\log 2)$

$i \leq \log_2 n$

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;2*j)  
    {  
        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 $j=0, 1, 2, 3, \dots, n-1$ will execute n times and $k=0, 1, 2, 3, \dots, n-1$.
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.

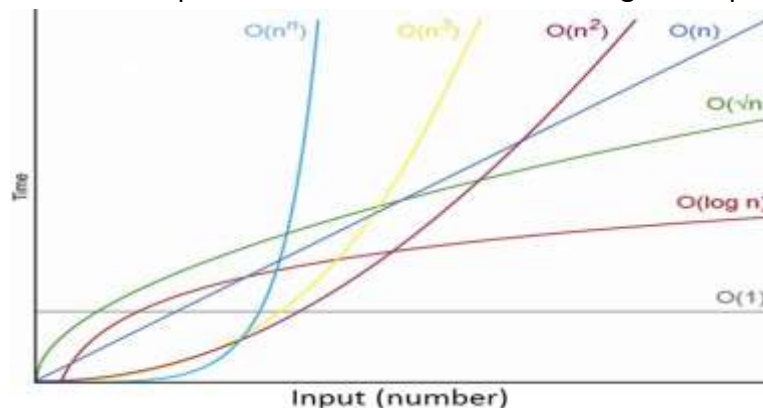
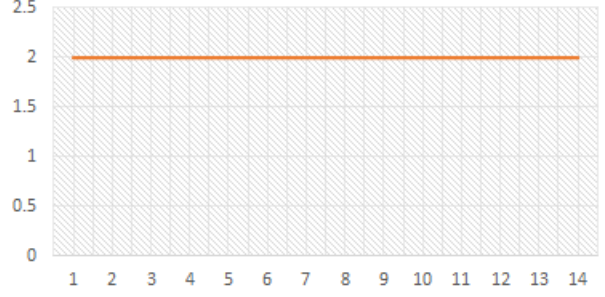
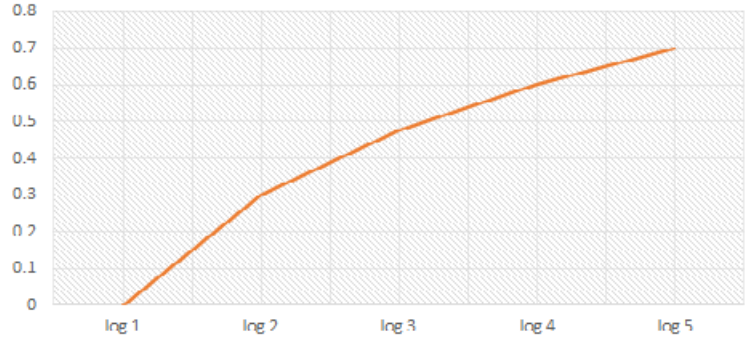
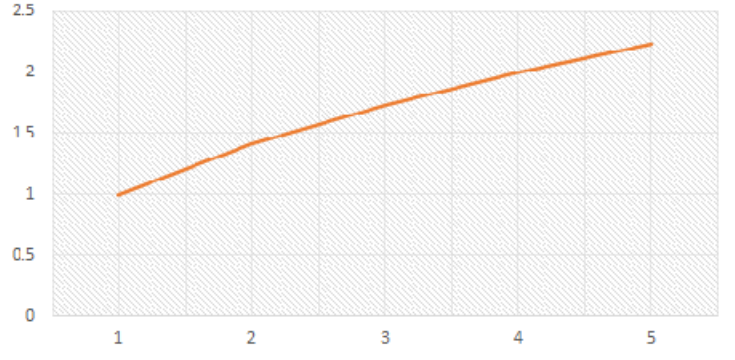
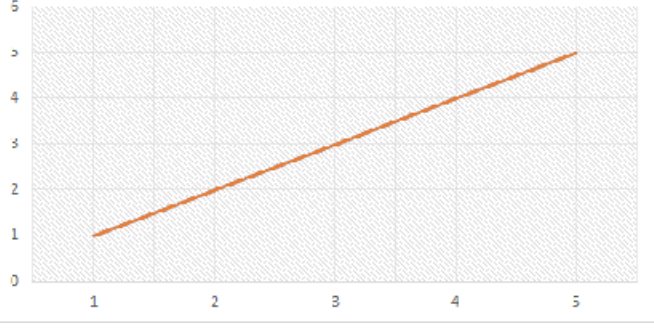
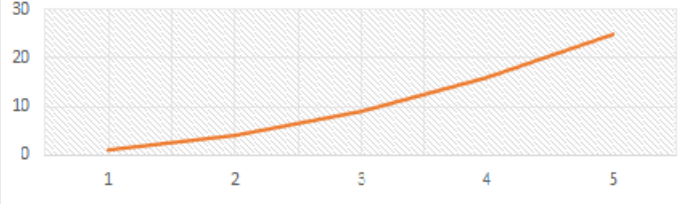
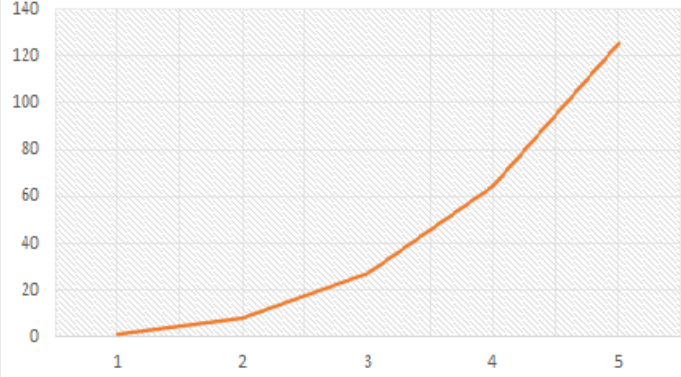
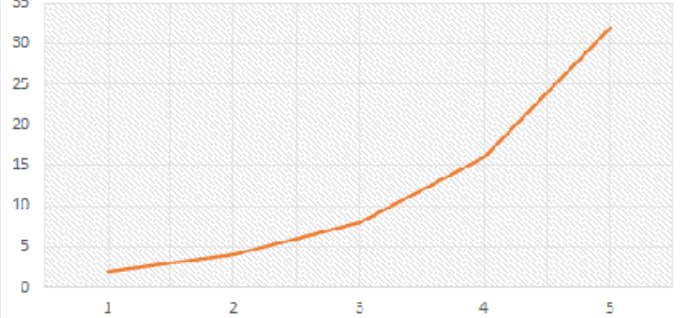
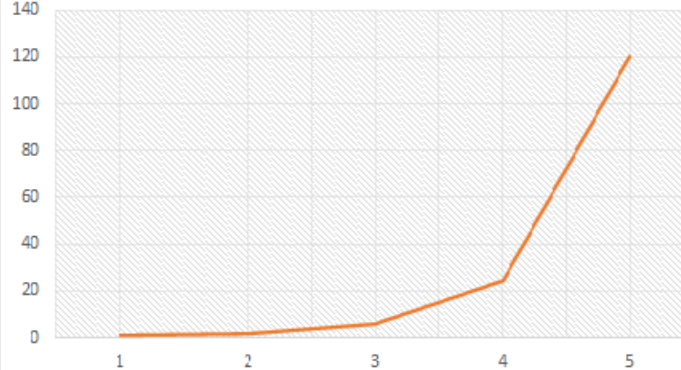


Figure 1.6: Representing the growth of various mathematical functions over input size n

Table 3: With the help of above diagram, we can see growth of functions at some value of n

Function	Remark	Graph
----------	--------	-------

Constant (C)	<p>Always order of 1. No matter value of C is 1, 10, 100, 1000 or any number</p>	<p>Constant (C) = 2</p> 
Logarithmic (log n)	<p>Complexity is order of log n.</p> <p>If n=1 then $\log(1)=0$ If n=2 then $\log(2)=.3010$ If n=3 then $\log(3)=.4771$ If n=4 then $\log(4)=.6020$ ----- If n=10 then $\log(10)=1$ Faster than constant(C)</p>	<p>log n</p> 
Square root (\sqrt{n})	<p>If n=1 then $\sqrt{1}=1$ If n=2 then $\sqrt{2}=1.414$ If n=3 then $\sqrt{3}=1.732$ If n=4 then $\sqrt{4}=2$ ----- If n=16 then $\sqrt{16}=4$ Faster than logarithmic (log n)</p>	<p>\sqrt{n}</p> 
Linear (n)	<p>Complexity is order of n. Always grow in linear manner.</p> <p>If n=1 then complexity is 1 If n=2 then complexity is 2 If n=3 then complexity is 3 If n=4 then complexity is 4 ----- Complexity is dependent on the value of n, and n is variable. Faster than square root(\sqrt{n})</p>	<p>n</p> 

Quadratic (n^2)	<p>Function growth is fast If $n=1$ then complexity is 1 If $n=2$ then complexity is 4 If $n=3$ then complexity is 9</p> <hr/> <p>If $n=6$ then complexity is 36. Hence, (n^2) Faster than Linear (n)</p>	<p>n^2</p> 
Cubic (n^3)	<p>If $n=1$ then complexity is 1 If $n=2$ then complexity is 8 If $n=3$ then complexity is 27 ---- If $n=5$ then complexity is 125</p> <p>Faster than Quadratic (n^2)</p>	<p>n^3</p> 
Exponential (2^n)	<p>If $n=1$ then complexity is 2 If $n=2$ then complexity is 4 If $n=3$ then complexity is 8 If $n=4$ then complexity is 16 -----</p> <p>It is slow in starting but for larger value it is Faster than Cubic (n^3)</p>	<p>2^n</p> 
Factorial ($n!$)	<p>If $n=1$ then complexity is $1=1$ If $n=2$ then complexity is $2*1=2$ If $n=5$ then complexity is $5*4*3*2*1 = 120$</p> <p>Faster than Exponential (2^n)</p>	<p>$n!$</p> 

Comparison of various mathematical functions will result as follows:

Constant < log n < n < n^2 < n^3 < 2^n < $n!$

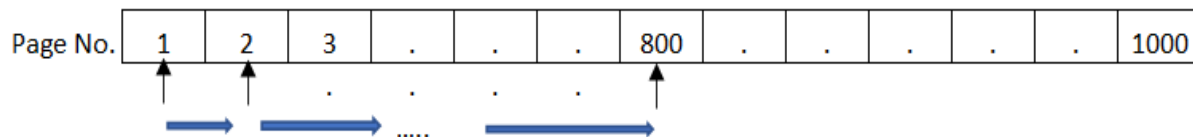
2.4. Why Analysis of Algorithms is important:

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

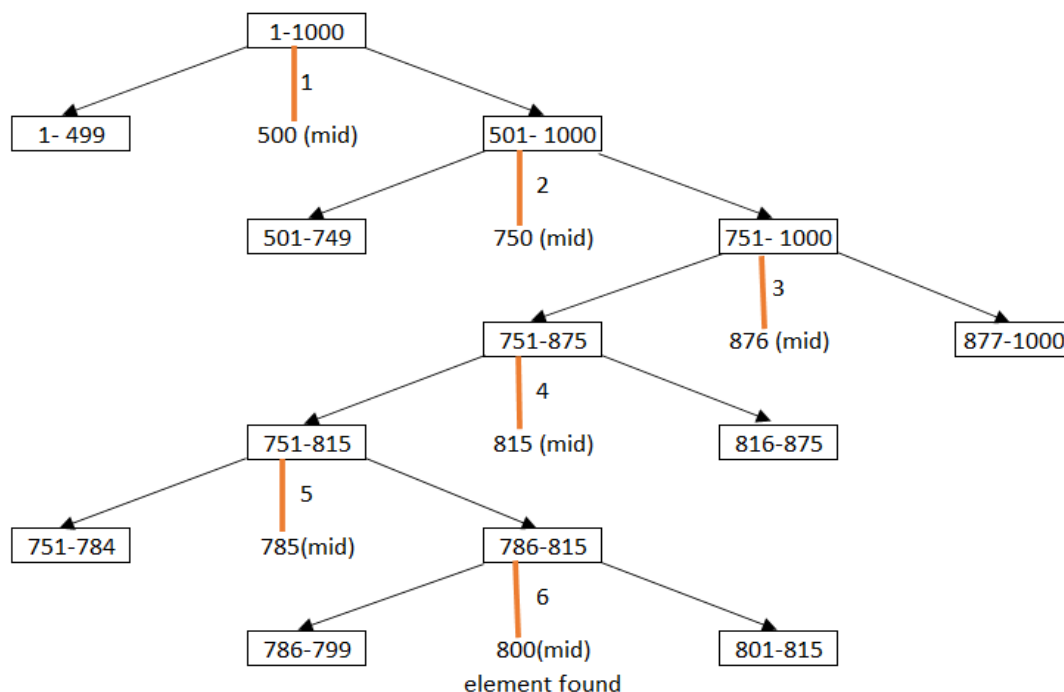
Example1: (Dictionary)

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: 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: 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 pages 750 to 1000. Repeat this step till reach page 800.



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

Three cases to analyse any problem:

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

To understand these cases,

Problem 1: Consider given array of random elements, we have to categorize best, worst and average case. In this example, searching of an element is done.

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

1. 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.

2. 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.

3. 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.

Another Example: 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.

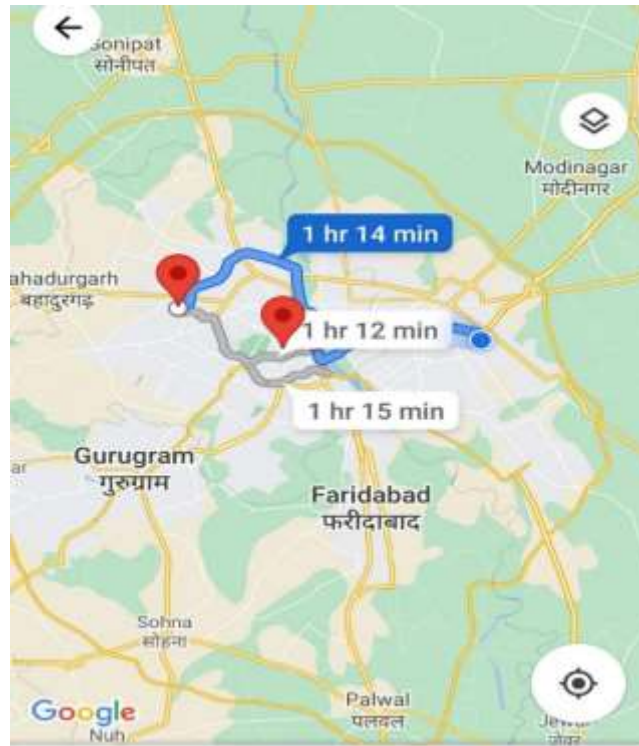


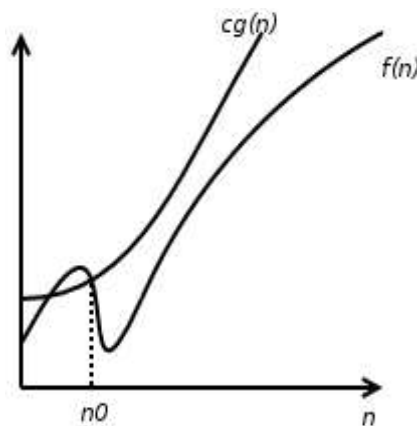
Figure: 1.7 - Google Map

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.

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 $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. $f(n)$ is thus $O(g(n))$.

$O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$

$0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$

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 definition: $f(n) \leq C \cdot g(n), \forall n \geq n_0$

From given problem statement,

$$f(n)=3n+5$$

$$g(n)=n$$

Now there exists a constant such that value of $c>0$ and $n \geq 1$

So, let's write the equation as $3n+5 \leq C \cdot n$

On solving the in-equality we get $C=4$

Then checking for different values of n

$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
$8 \leq 4$	$11 \leq 8$	$14 \leq 12$	$17 \leq 16$	$20 \leq 20$
False	False	False	False	True

So from case 5 $f(n)=O(n) \forall n \geq 5$

2. Given $f(n)=n^3$, then $f(n) \neq O(n^2)$

Solution 2: As per definition: $f(n) \leq C \cdot g(n), \forall n \geq n_0$

From given problem statement

$$f(n)=n^3$$

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

Now there exists a constant such that value of $c>0$ and $n \geq 1$

So, let's write the equation $n^3 \leq C \cdot n^2$

If we assume $C=1$ then $n=1$ (condition is true) but for any other value of n condition is false. $f(n)=O(n^2)$ condition is false.

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

Solution 3: As per definition: $f(n) \leq C \cdot g(n), \forall n \geq n_0$

From given problem statement $f(n)=2n^2+5n+1$

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

Now there exists a constant such that value of $c > 0$ and $n \geq 1$

So let's write the equation as $C(n^2) \geq 2n^2 + 5n + 1$

We have to find the value of C

$$C \geq 2n^2/n^2 + 5n/n^2 + 1/n^2$$

$$\text{We get } C \geq 1 + 5/n + 1/n^2$$

On putting different values of n we get $C=4$

Now check the value of n for different cases:

n=1	n=2	n=3
$8 \leq 4$	$19 \leq 16$	$34 \leq 36$
False	False	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 definition: $f(n) \leq C \cdot g(n), \forall n \geq n_0$

From given problem statement

$$f(n) = 4^n$$

$$g(n) = 8^n$$

Now there exists a constant such that value of $c > 0$ and $n \geq 1$

So let's write the equation as $4^n \leq C \cdot 8^n$

$$= 4^n / 8^n \leq C$$

$$= 1/2^n \leq C$$

So, try for different values of n i.e. 1, 2, ..., n , we get 1, 1/2, ...

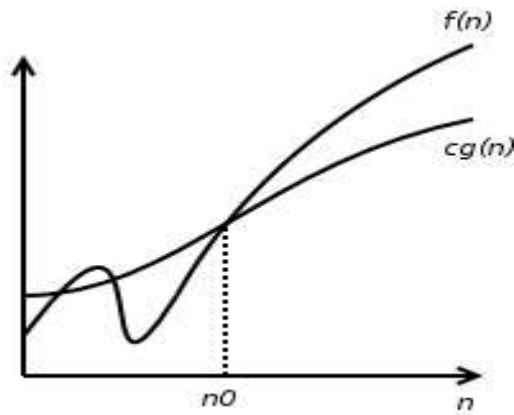
So, let's choose $C=1$

n=1
$4 \leq 8$
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 $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$. $f(n)$ is thus $\Omega(g(n))$.
 $\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$
 The above expression can be described as a function $f(n)$ belongs to the set $\Omega(g(n))$ 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 $\Omega(g(n))$

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 $\Omega(1)$ or $\Omega(\log(n))$, depending on the value of $foo(i)$.

Examples on Big Omega Notation:

Example 1: Given $f(n)=3n+5$, then prove that $f(n)=\Omega(n)$

Solution 1: As per def.: $f(n) \geq C \cdot g(n), \forall n \geq n_0$

For the given problem statement, $f(n)=3n+5$, $g(n)=n$

Now there exists a constants such that value of $c>0$ and $n \geq 1$

As per Big-Omega theory, $C \cdot g(n) \leq f(n)$

So let's write the equation as $C \cdot n \leq 3n+5$

On solving the inequality we find that when

Case 1: $C=1$ and $n=1$ i.e.

$$1(1) \leq 3(1) + 5$$

$$1 \leq 8 \text{ (True)}$$

So from case 1 $f(n)=\Omega(n) \forall n \geq 1$

Example 2: Given $f(n)=n^2$, then prove that $f(n) \neq \Omega(n^3)$.

Solution 2: As per def.: $f(n) \geq C \cdot g(n), \forall n \geq n_0$

For the given problem statement, $f(n)=n^2$, $g(n)=n^3$

Now there exists a constants such that value of $c>0$ and $n \geq 1$

As per Big-Omega theory, $C \cdot g(n) \leq f(n)$

Equation can be written as

$$n^2 \geq C n^3$$

If we take $C=1$ and $n=1$ i.e. $1^2 \geq 1 * 1^3$

then only condition is true otherwise for other values of C and n condition is false.

Example 3: Prove that $10n^2 + 3n + 3 = \Omega(n^2)$

Solution3: As per def.: $f(n) \geq C \cdot g(n), \forall n \geq n_0$

For the given problem statement, $f(n)=10n^2 + 3n + 3$ and $g(n)=n^2$

Now there exists a constants such that value of $c>0$ and $n \geq 1$

As per Big-Omega theory, $C \cdot g(n) \leq f(n)$

Equation can be written as

$$10n^2 + 3n + 3 \geq C(n^2)$$

For $C=1$

Let's test it for different values of n

$n=1$	$n=2$	$n=3$
$16 \geq 1$	$49 \geq 4$	$102 \geq 9$
True	True	True

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

$$10n^2 + 3n + 3 = \Omega(n^2) \quad \forall n \geq 1$$

Example 4: Prove that $100n + 5 = \Omega(n^2)$

Solution 4: As per def.: $f(n) \geq C \cdot g(n), \forall n \geq n_0$

For the given problem statement, $g(n) = n^2, f(n) = 100n + 5$

Now there exists a constants such that value of $c>0$ and $n \geq 1$

As per Big-Omega theory, $C \cdot g(n) \leq f(n)$

By inequality,

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

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

for $n = 2, C \leq 50$

$$\text{LHS} = 100 \cdot 2 + 5 = 205$$

$$\text{RHS} = 50 \cdot 4 = 200 < 205$$

Hence, $100n + 5 = \Omega(n^2), \forall n \geq 2$

Example 5: Prove that $n^3 = \Omega(n^2)$

Solution 5.: As per def.: $f(n) \geq C \cdot g(n), \forall n \geq n_0$

For the given problem statement, $g(n) = n^2, f(n) = n^3$

Now there exists a constants such that value of $c>0$ and $n \geq 1$

As per Big-Omega theory,

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

Equation can be written as

$$n^3 \geq C(n^2)$$

For $C=1$

Let's test it for different values of n

n=1	n=2	n=3
$1 \geq 1$	$8 \geq 4$	$27 \geq 9$
True	True	True

$$0 \leq C \cdot n^2 \leq n^3$$

For $C=1$, and $n \geq 1$, $n^2 \leq n^3$

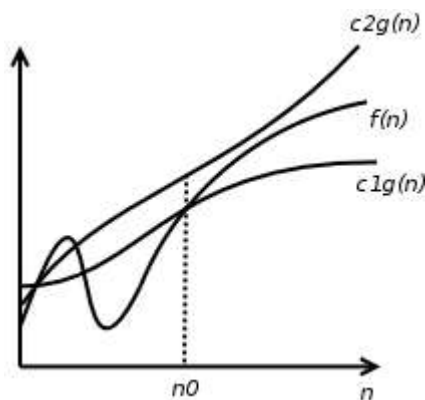
For $C=1$, and $n=2$ $2^2 \leq 2^3$

$4 < 8$ (condition is true)

Hence, $n^3 = \Omega(n^2)$, $\forall n \geq 1$

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 $\Theta(g(n))$.

For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}$

time is at least $c_1(n)$ and maximum $c_2(n)$.

Examples on Theta Notation:

Example 1: $f(n) = 12n^2 + 3n + 7$, then prove that $g(n) = \Theta(n^2)$

Solution 1. As per def.: $C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$

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 $n \geq 1$

As per Big-Theta theory $C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$

$$C1. n^2 \leq n^2 + 3n + 7 \leq C2. n^2$$

If $C1=1$ & $C2=2$ then $n \geq 5$

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

Example 2: Prove that $\frac{n^2}{2} - 2n = \Theta(n^2)$

Solution 2: As per def.: $C1.g(n) \leq f(n) \leq C2.g(n)$

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 $n \geq 1$

As per Big-Theta theory $C1.g(n) \leq f(n) \leq C2.g(n)$

Here $C1$ and $C2$ are constant.

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

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

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

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

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

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

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

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

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

$$\text{or, } 16 \leq 32 - 16 \leq 32$$

$$\text{or, } 16 \leq 16 \leq 32 \text{ (True condition)}$$

Example 3: Show that $(n + a)^b = \Theta(n^b)$ where $b > 0$

Solution 3: As per def.: $C1.g(n) \leq f(n) \leq C2.g(n)$

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 $n \geq 1$

As per Big-Theta theory $C1.g(n) \leq f(n) \leq C2.g(n)$

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,

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

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

Thus value of $c_1 = \left(\frac{1}{2}\right)^b$, $c_2 = 2^b$ and $n = 2a$,

Hence, proved.

Example 4: Prove that $2n - 2\sqrt{n} = \theta(n)$

Solution 4: As per def.: $C_1.g(n) \leq f(n) \leq C_2.g(n)$

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 $n \geq 1$

As per Big-Theta theory $C_1.g(n) \leq f(n) \leq C_2.g(n)$

Case 1: Solving inequality, $2n - 2\sqrt{n} \leq C_2.n$

$$C_2 \geq 2 - 2\sqrt{n}/n$$

$$C_2 = 2$$

Case 2: Solving inequality, $C_1.n \leq 2n - 2\sqrt{n}$

$$C_1 \leq 2 - (2\sqrt{n})/n$$

$$C_1 = 1$$

for $C_1 = 1$ and $C_2 = 2$ and $n = 4$

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

$$4 \leq 8 - 4 \leq 8$$

$$4 \leq 4 \leq 8 \text{ (True)}$$

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	1	0
{	0	1	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	1	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	1	0
{	0	1	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	1	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(int 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 $O(n)$

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:

Iteration Method

- Recursion Tree Method
- Master Method

2.5 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)

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

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

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

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

$$T(n - 2) = 1 + T(n - 3) \text{----- (iii)}$$

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

$$\begin{aligned}T(n) &= 1 + 1 + T(n - 2) \\&= 2 + T(n - 2) \text{----- (iv)}\end{aligned}$$

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

$$\begin{aligned}T(n) &= 2 + 1 + T(n - 3) \\&= 3 + T(n - 3) \\&\dots\dots \\&\dots\dots \\&\dots\dots \\&\dots\dots\end{aligned}$$

Now, we got a pattern that

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

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:

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

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

$$\begin{aligned}T(n) &= (n - 1) + T(n - (n - 1)) \\&= (n - 1) + T(1) \\&= (n - 1) + 1 \\&= n\end{aligned}$$

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

Example 2:

$$T(n) = n + T(n - 1); \text{ for } n > 1$$

$$= 1 ; \text{ for } n = 1$$

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

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

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

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

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

$$= n + (n - 1) + (n - 2) + \dots\dots\dots 2 + 1$$

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:

$$T(n) = T(n/2) + c; \text{ for } n > 1$$

$$= 1; \text{ for } n = 1$$

Solution:

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

$$T(n) = T(n/2) + c \text{ ----- (i)}$$

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

$$T(n/2) = T(n/4) + c \text{ ----- (ii)}$$

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

$$T(n/4) = T(n/8) + c \text{ ----- (iii)}$$

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

$$T(n) = T(n/4) + c + c \Rightarrow T(n/2^2) + 2c \text{ ----- (iv)}$$

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

$$T(n) = T(n/8) + c + 2c \Rightarrow T(n/2^3) + 3c$$

.....

.....

.....

.....

.....

Now, we got a pattern that

$$T(n) = T(n/2^k) + kc \text{ ----- (v)}$$

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:

$$2^k = n \text{-----(vi)}$$

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

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

$$= T(1) + kc$$

$$= 1 + kc \text{-----(vii)}$$

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

$$\log n = \log 2^k$$

$$\log n = k \log 2$$

$$\log n = k \text{ (since } \log 2 = 1 \text{)}$$

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

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

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

Therefore, the solution for the given recursive problem is $O(\log n \text{ base } 2)$.

Example 4:

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

$$= 1, \text{ if } n = 1$$

Solution:

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

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

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

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

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

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

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

$$\begin{aligned} T(n) &= 2 [2 T(n - 2)] \\ &= 2^2 T(n - 2) \text{ ----- (iv)} \end{aligned}$$

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

$$\begin{aligned} T(n) &= 4 [2 T(n - 3)] \\ &= 2^3 T(n - 3) \text{ ----- (v)} \end{aligned}$$

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

$$T(n) = 2^k T(n - k) \text{ ----- (vi)}$$

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

$$n - k = 1 \Rightarrow k = n - 1 \text{ ----- (vii)}$$

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

$$\begin{aligned} T(n) &= 2^{(n - 1)} T(n - (n - 1)) \\ &= 2^{(n - 1)} T(1) \\ &= 2^{(n - 1)} \end{aligned}$$

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

2.6 Drawbacks of 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:

$T(n) = 2T(\sqrt{n}) + \log n$. Solve the recurrence by changing variable.

Solution:

We have, $T(n) = 2T(\sqrt{n}) + \log n$ -----(i)

Suppose $m = \log n \Rightarrow n = 2^m$ ---- (ii)

Therefore, $n^{1/2} = 2^{(m/2)} \Rightarrow \sqrt{n} = 2^{(m/2)}$ ----- (iii)

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

$$T(2^m) = 2T(2^{(m/2)}) + m$$

Again, let's consider $S(m) = T(2^m)$

$$\text{Therefore, } S(m) = 2 S(m/2) + m \text{ ----- (iv)}$$

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

$$S(m) = O(m \log m)$$

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

$$\begin{aligned} T(n) = S(m) &= O(m \log m) \\ &= O(\log n \log \log n) \end{aligned}$$

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

Example 2: Consider the recurrence:

$T(n) = 2T(\sqrt{n}) + 1$. Solve the recurrence by changing variable.

Solution:

We have, $T(n) = 2T(\sqrt{n}) + 1$ -----(i)

Suppose $m = \log n \Rightarrow n = 2^m$ ---- (ii)

Therefore, $n^{1/2} = 2^{(m/2)} \Rightarrow \sqrt{n} = 2^{(m/2)}$ ----- (iii)

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

$$T(2^m) = 2T(2^{(m/2)}) + 1$$

Again, let's consider $S(m) = T(2^m)$

$$\text{Therefore, } S(m) = 2 S(m/2) + 1 \text{ ----- (iv)}$$

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

$$S(m) = O(\log m)$$

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

$$\begin{aligned} T(n) = S(m) &= O(\log m) \\ &= O(\log \log n) \end{aligned}$$

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 contain the terms in root.

Master Method:

2.7 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 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 $T(n) = a T(n/b) + f(n)$.

Definition of Master Method:

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

$$T(n) = a T(n/b) + f(n) \quad \text{where } a \geq 1 \text{ and } b > 1$$

- 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 work done in recursive calling, which includes the cost of dividing the problem and the cost of merging the solutions to the sub-problems

Three Cases of Master Method:

Case 1: If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

Case 2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3: If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n .

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.

- 2.7.1 In case 1, the function $n^{\log_b a}$ is larger, and the solution is $T(n) = \Theta(n^{\log_b a})$. In case 3 function $f(n)$ is larger, solution is $T(n) = \Theta(f(n))$. In case 2, both functions have the same value. The solution is $T(n) = \Theta(n^{\log_b a} \log n)$.
- 2.7.2 In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be polynomially smaller.
- 2.7.3 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 $aT(n/b) \leq cf(n)$.
- 2.7.4 In addition to that, all three cases do not cover all possibilities. Some function might lie 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.

Examples on Master Method:

Example 1: $T(n) = 3T(n/2) + n^2$

$$T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2) \quad (\text{Case 3})$$

Example 2: $T(n) = 4T(n/2) + n^2$

$$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n) \quad (\text{Case 2})$$

Example 3: $T(n) = T(n/2) + 2^n$

$$T(n) = T(n/2) + 2^n \Rightarrow T(n) = \Theta(2^n) \quad (\text{Case 3})$$

Example 4: $T(n) = 2^n T(n/2) + n^n$

$$T(n) = 2^n T(n/2) + n^n \Rightarrow \text{Does not apply (a is not constant)}$$

Example 5: $T(n) = 16T(n/4) + n$

$$T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2) \quad (\text{Case 1})$$

Example 6: $T(n) = 2T(n/2) + n \log n$

$$T(n) = 2T(n/2) + n \log n \Rightarrow T(n) = n \log_2 n \quad (\text{Case 2})$$

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

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

2.8 Recursion Tree Method

2.8.1 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.

2.8.2 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.

2.8.3 Sample Recurrences using Recursion Tree Method:

Problem-01:

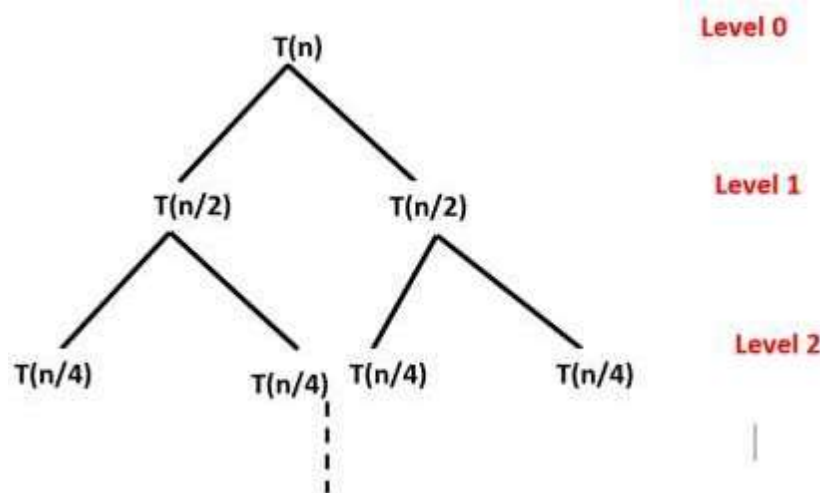
Solve the given recurrence using recursion tree method-

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

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.

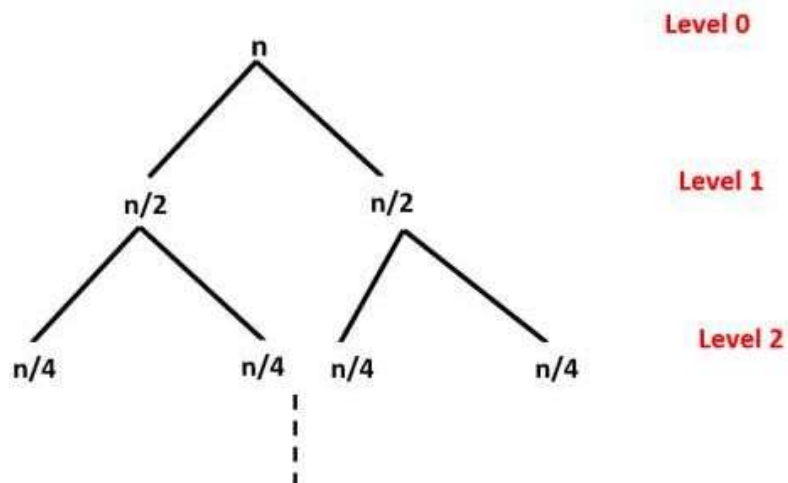


After the working of the recurrence function $T(n)$, we will further see the cost function through which we can determine the cost at each level of the recursion tree: The given cost recurrence shows:

- 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 1 is $n/2^1$

- The size of the sub-problem at level 0 is $n/2^0$ 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: $x \log_2 = \log_2 n$, $x = \log_2 n$

\therefore 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 = $n \times T(1) = \theta(n)$

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.

$$\begin{aligned}
 T(n) &= n + n + n + n + \dots + \theta(n) \quad (\text{Total Levels}) \\
 &\quad \log_2 n + 1 \quad \text{Last level} \\
 &= n \times \log_2 n + \theta(n) \\
 &= n \log_2 n + \theta(n) \\
 &= \theta(n \log_2 n)
 \end{aligned}$$

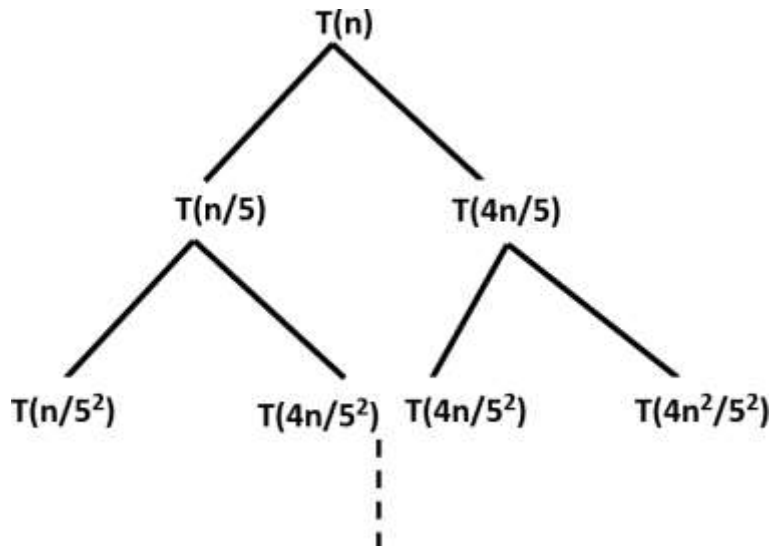
Problem-02: Solve the given recurrence using recursion tree method: $T(n) = T(n/5) + T(4n/5) + n$

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. $T(n)$

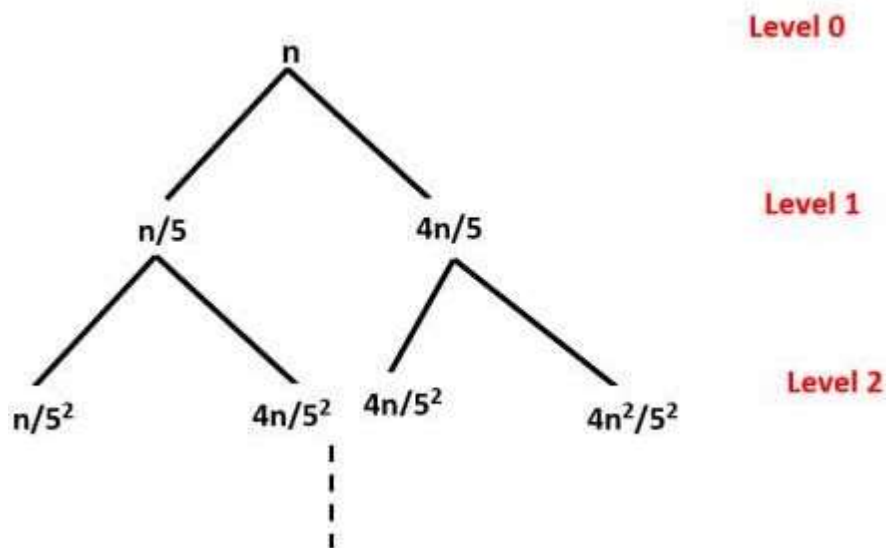
function will be elaborated as follows:



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 subtree 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- $X \log (4/5) =$

$$\log (1/n)$$

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

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

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-

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

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

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

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

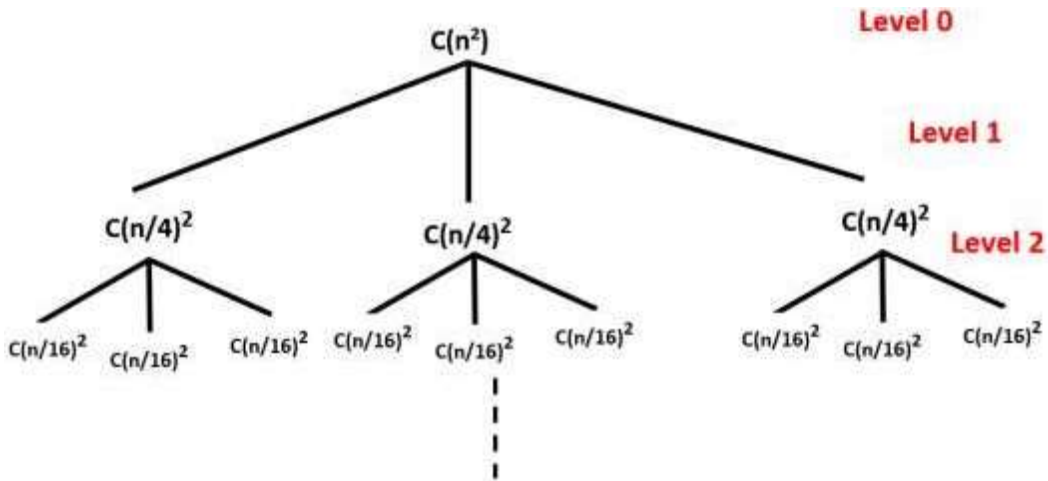
$$= \theta(n \log_{5/4} n)$$

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

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

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- $x \log 4 = \log n$

$$x = \log_4 n$$

$$\therefore \text{Total number of levels in the recursion tree} = \log_4 n + 1$$

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-

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

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

$$T(n) = \{ \underline{cn^2} + (3/16)cn^2 + (9/16^2)cn^2 + \dots \} + \Theta(n^{\log_4 3})$$

For $\log_4 n$ levels

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

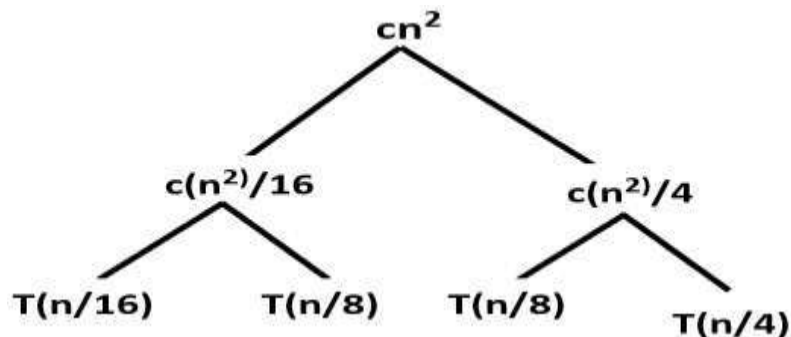
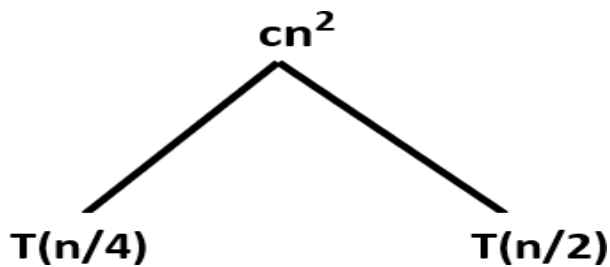
Now, $\{ 1 + (3/16) + (3/16)^2 + \dots \}$ forms an infinite Geometric progression. On solving, we get-

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

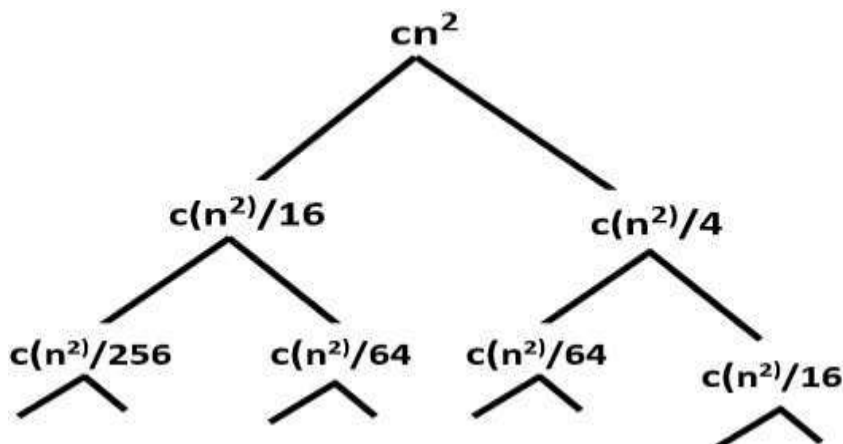
$$= (16/13) cn^2 - (16/13) cn^2 (3/16)^{\log_4 n} + \Theta(n^{\log_4 3}) = O(n^2)$$

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 $T(n) = c(n^2 + 5(n^2)/16 + 25(n^2)/256) + \dots$

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

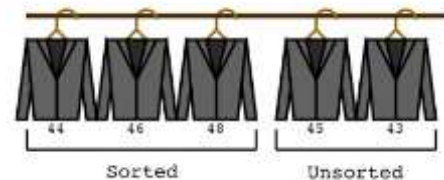
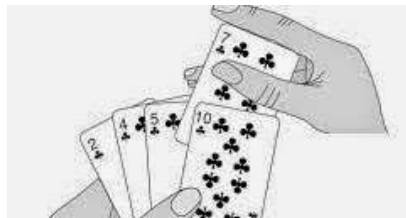
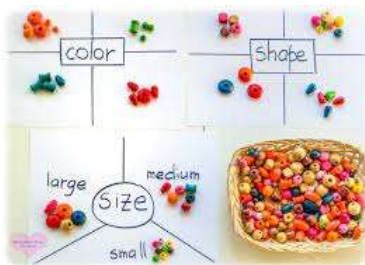
To get an upper bound, we can apply the formula of the sum to the infinite series. We get the sum as $(n^2)/(1 - 5/16)$, which is $O(n^2)$.

2.1. Introduction to Sorting

2.1.1. Case Studies to understand the requirement of Sorting:

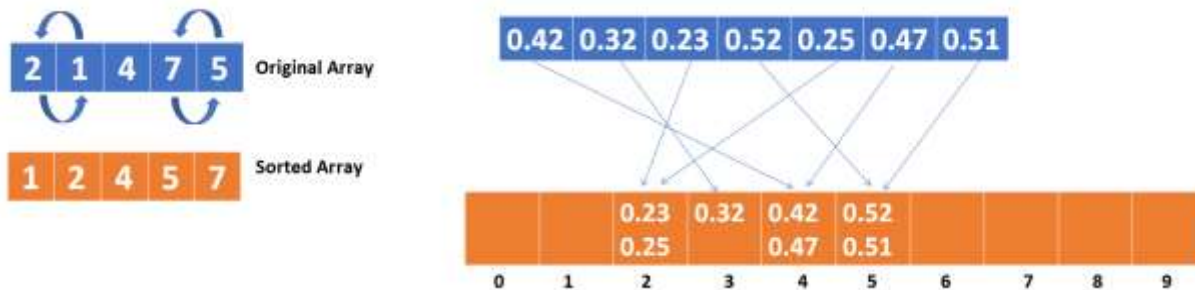
In real life scenario we knowingly and unknowingly use sorting extensively.

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.

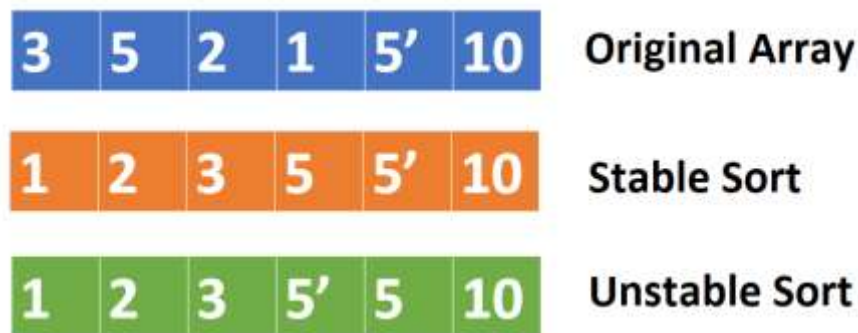


2.1.2 Types of Sorting Algorithm:

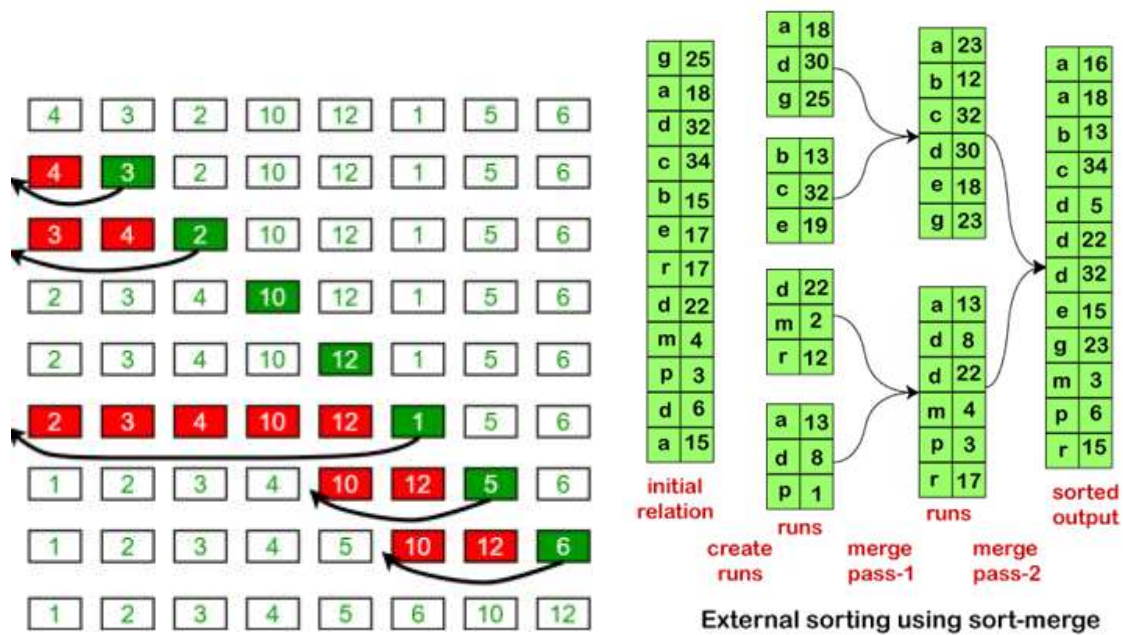
(a) In-place sorting and not-in-place sorting - In in-place sorting algorithm we use fixed additional space for producing the output (modifies only the given list under consideration). It sorts the list only by modifying the order of the elements within the list. e.g. Bubble sort, Comb sort, Selection 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.



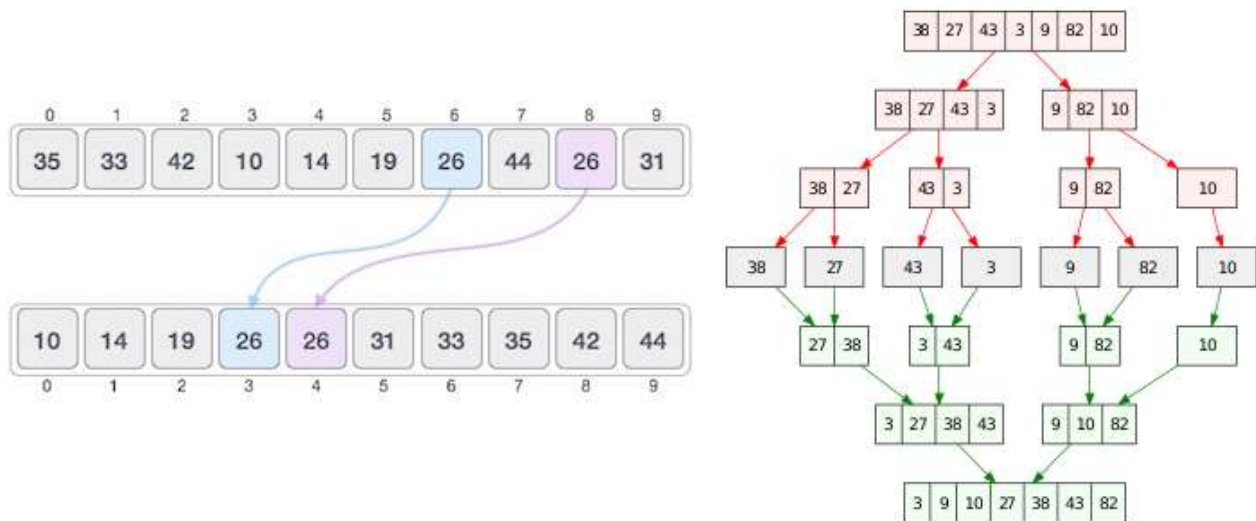
(b) Stable sorting and Unstable sorting— In stable sorting algorithm the order of two objects with equal keys in output remains same after sorting as they appear in the input array to be sorted. e.g. Merge Sort, Insertion Sort, Bubble Sort, and Binary Tree Sort. If a sorting algorithm, after sorting 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 $n \log n$ (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

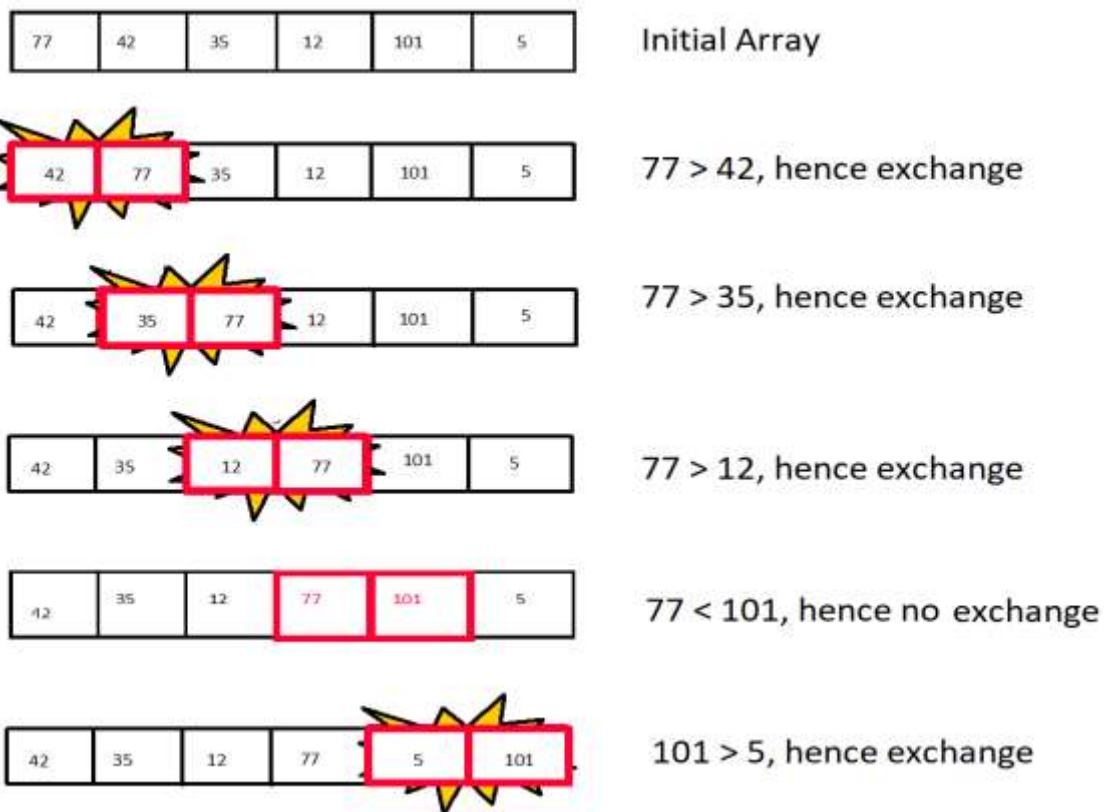
one less comparison than the last pass. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a concern).

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 1st element is compared with 2nd and if previous element is larger than the next one, elements are exchanged with each other. Then 2nd and 3rd elements are compared and if second is larger than the 3rd, they are exchanged. The process repeats and finally (n-1)st element gets compared with nth 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-1 comparisons in first iteration

N-2 comparison in second iteration

N-3 comparison in third iteration

...

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

$$\begin{aligned}\text{Total comparisons} &= (N-1) + (N-2) + (N-3) + \dots + 1 \\ &= N(N-1)/2 \\ &= N^2/2 - N/2\end{aligned}$$

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)

5	2	3	4	8	6
---	---	---	---	---	---

Initial Array

2	5	3	4	8	6
---	---	---	---	---	---

$5 > 2$, hence exchange

2	3	5	4	8	6
---	---	---	---	---	---

$5 > 3$, hence exchange

2	3	4	5	8	6
---	---	---	---	---	---

$5 > 4$, hence exchange

2	3	4	5	8	6
---	---	---	---	---	---

$5 < 8$, hence no exchange

2	3	4	5	6	8
---	---	---	---	---	---

$8 > 6$, hence exchange

Optimized Bubble Sort (Second Pass)

2	3	4	5	6	8
---	---	---	---	---	---

$2 < 3$, Hence no Exchange

2	3	4	5	6	8
---	---	---	---	---	---

$3 < 4$, Hence no Exchange

2	3	4	5	6	8
---	---	---	---	---	---

$4 < 5$, Hence no Exchange

2	3	4	5	6	8
---	---	---	---	---	---

$5 < 6$, Hence no Exchange

2	3	4	5	6	8
---	---	---	---	---	---

$6 < 8$, Hence no Exchange

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 $\Omega(N)$ 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.

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 (3rd to Nth 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

77	42	35	12	101	5
Min					
77	42	35	12	101	5
Min					
77	42	35	12	101	5
Min					
77	42	35	12	101	5
Min					
77	42	35	12	101	5
Min					
77	42	35	12	101	5
Min					
5	42	35	12	101	77
Min					

Initial Array
min = 77

42 < Min Hence Min = 42

35 < Min Hence Min = 35

12 < Min Hence Min = 12

101 > Min Hence Min does not change

5 < Min Hence Min = 5

Min = 5, Exchange it with first element

The Second Pass

5	42	35	12	101	77
Min					
5	42	35	12	101	77
Min					
5	42	35	12	101	77
Min					
5	42	35	12	101	77
Min					
5	42	35	12	101	77
Min					
5	42	35	12	101	77
Min					
5	12	35	42	101	77
Min					

Initial Array
Min = 42

35 < Min Hence Min = 35

12 < Min Hence Min = 12

101 > Min Hence Min does not change

77 > Min Hence Min does not change

Min = 12, Exchange it with second element

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

$$\begin{aligned}
 \text{Total} &= (N-1) + (N-2) + (N-3) + \dots + 1 \\
 &= N(N-1)/2 = N^2/2 - N/2
 \end{aligned}$$

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

$$\text{Total Effort} = 3 * (N-1) + N^2/2 - N/2 = N^2/2 + 5/2 * N - 3$$

As there is no best case possible as all iterations are compulsory, Complexity should be written in

Theta notation i.e. $\Theta(N^2)$

Space Complexity remains $\Theta(1)$ 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.

2.4. 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

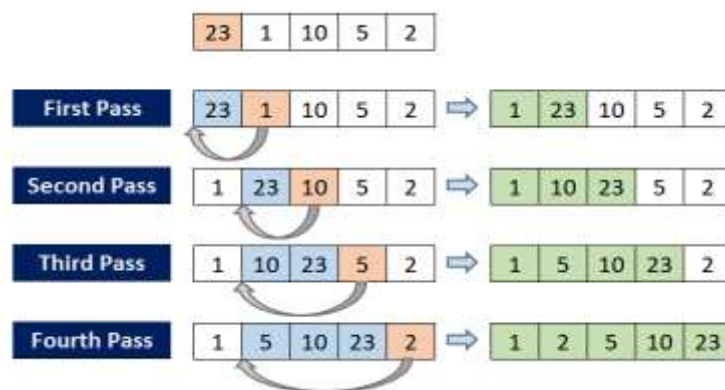
Red colored card set is sorted, Black Colored card set unsorted. Try inserting a card from unsorted set in the sorted set.

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 $(N-1)^{\text{st}}$ 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 comparisons in each iteration)

1 comparison in first iteration

2 comparisons in second iteration

3 comparisons in third iteration

...

N-1 comparisons 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) = $\Omega(N)$

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;

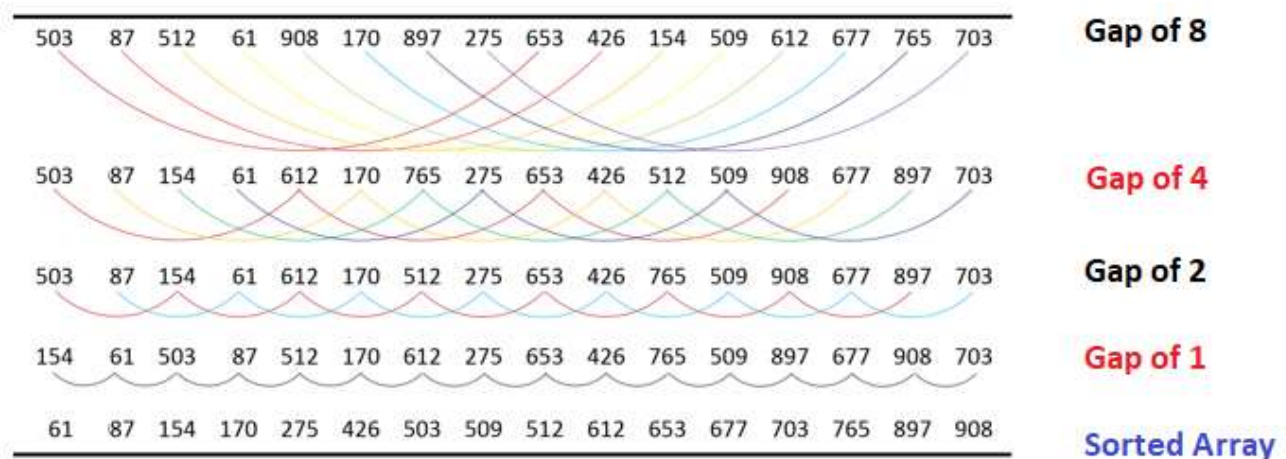
```

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 >= 1 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 Sort

1-Hibbard Gap Sequence

There is a comparison between Hibbard Shell Sort($n^{1.5}$) and Donald Shell Sort takes $O(n^2)$. It is most widely gap sequence suggested by Thomas Hibbard which is $(2^{k-1} \dots 31 \dots 15 \dots 7 \dots 3 \dots 1)$ which gives time complexity $O(n \sqrt{n})$

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
 / \

```

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-traverse 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: $O(N \log n)$.

Expected Auxiliary Space: $O(1)$.

Constraints:

$1 \leq N \leq 1000$

$1 \leq \text{arr}[i] \leq 1000$

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 $a[i]$.

Output:

For each test case, output sorted array and binary search calls counter modulo 1000000007.

Constraints

- $1 \leq N \leq 100000$
- $0 \leq M \leq 1090$

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 $\text{rev}(s)$ reverses the string ss . For example, $\text{rev}(010111)=111010$, and $\text{rev}(01)=10$.
- The string $\text{flip}(s)$ changes each character in ss from 00 to 11 or from 11 to 00. For example, $\text{flip}(010111)=101000$ and $\text{flip}(11)=00$.

If $s=\text{rev}(s)$ then we say that ss is a **palindrome**. If $s=\text{rev}(\text{flip}(s))$ then we say that ss is an **antipalindrome**.

Given a binary string $s = s_1s_2\dots s_{|s|}$, divide it into a palindrome and an antipalindrome. Formally, you should find two sequences i_1, i_2, \dots, i_k , and j_1, j_2, \dots, j_m , such that:

- $k, m \geq 0$
- $|s| = k + m$
- All indices $i_1, i_2, \dots, i_k, j_1, j_2, \dots, j_m$ are distinct integers satisfying $1 \leq i_x, j_x \leq |s|$.
- $i_1 < i_2 < \dots < i_k$ and $j_1 < j_2 < \dots < j_m$
- The string $s_{i_1}s_{i_2}\dots s_{i_k}$ is a palindrome.
- The string $s_{j_1}s_{j_2}\dots s_{j_m}$ 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, \dots, i_k .

In the third line for each test case, print m integers j_1, j_2, \dots, 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

- $1 \leq t \leq 105$
- $1 \leq |s| \leq 100000$
- the sum of lengths of all strings does not exceed 300000

Subtasks

Subtask #1:

- $t \leq 1000$
- $|s| \leq 10$

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 $s_2s_3s_5s_7s_9s_{10}=011001$.

2.5. 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.

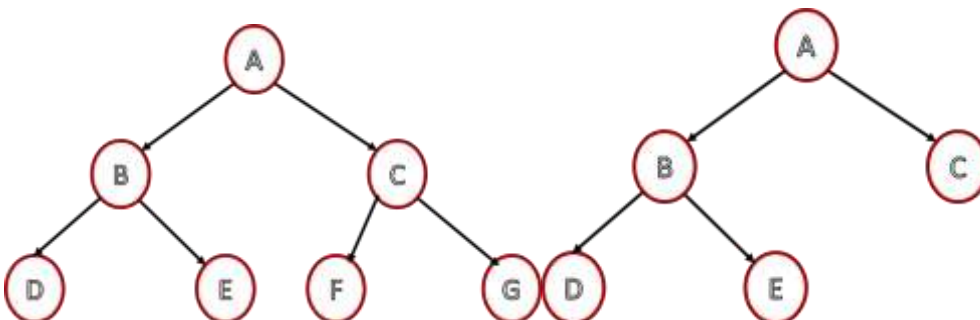
After the first round four teams will reach in semi final round where two semi finals will be played. From the semi Final round, two team will reach in final. Now the winner of the final will take This analogy resemble the concept of Heap(Max-Heap).



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

Application: To implement Heap Data structure.

2.5.3.Heap:

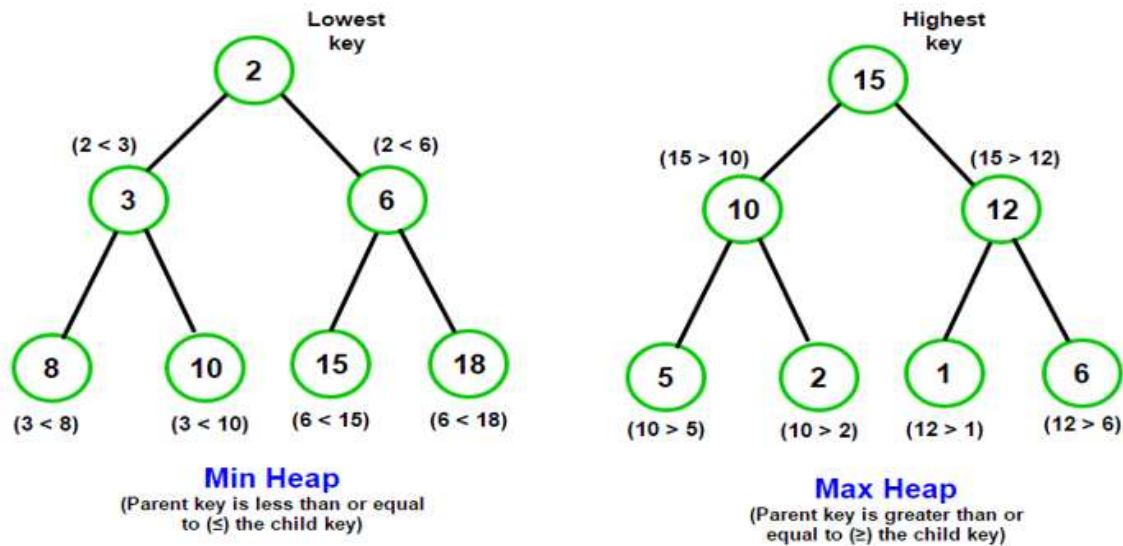
A Binary heap is a complete Binary Tree which makes it suitable to be implemented using array.

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.

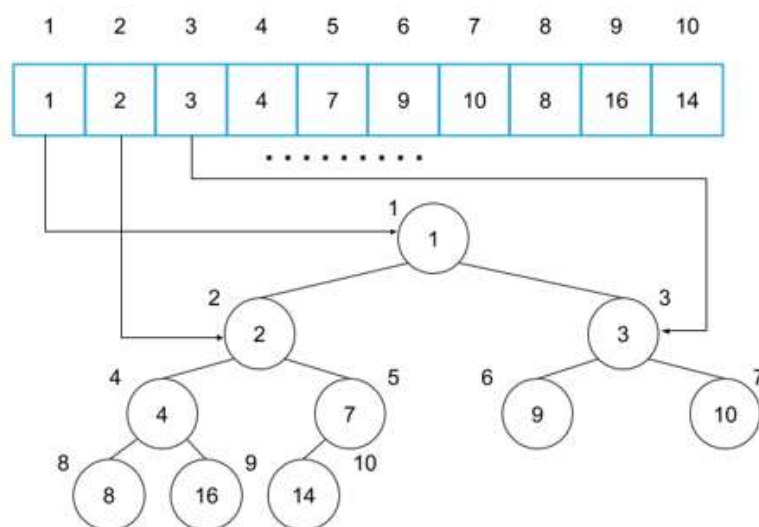
In a Max Binary Heap the value at the root node must be largest among all the values present in Binary Heap. This property of Max-Heap must be true repeatedly for all nodes in Binary Tree.

As heap is a complete binary tree therefore height of tree having N nodes will always $O(\log n)$.



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

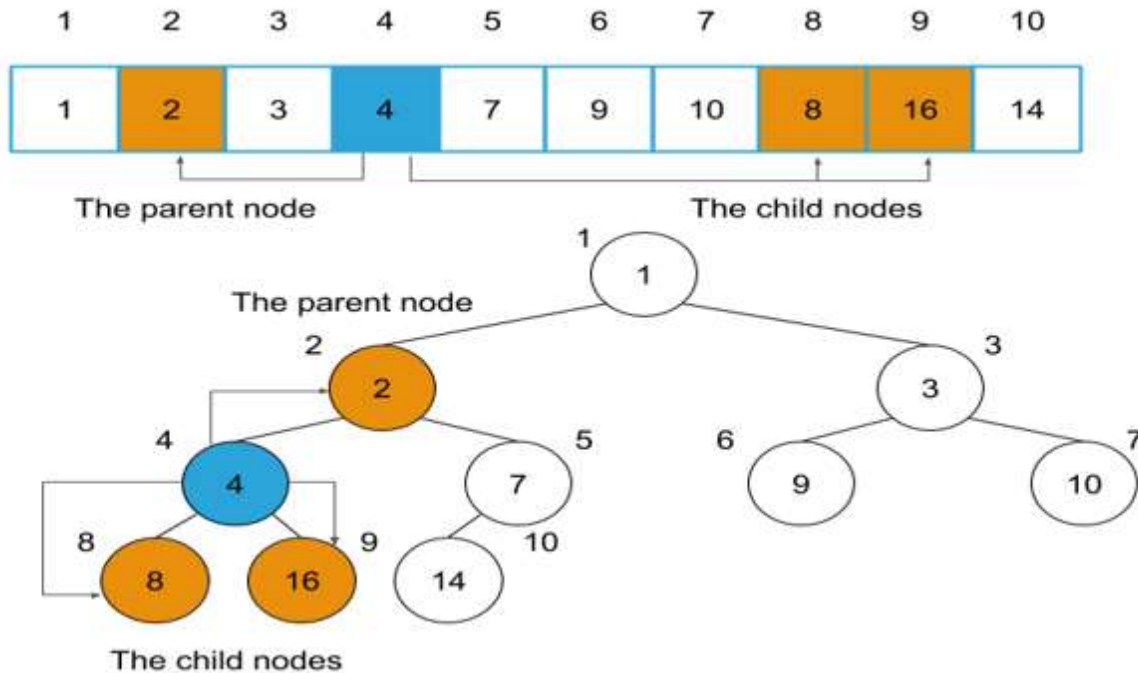
A left child node | $\text{left}(i) = 2*i$

A right child node | $\text{right}(i) = 2*i+1$

A parent node | $\text{parent}(i) = i / 2$

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 8th position and Right Child will be at $(2*4 + 1)$ 9th 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**:

- 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.
- 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 $\text{Left}(i)$ and $\text{Right}(i)$ 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:

- Given an arrays as below
- First construct a complete binary tree from the array
- 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).

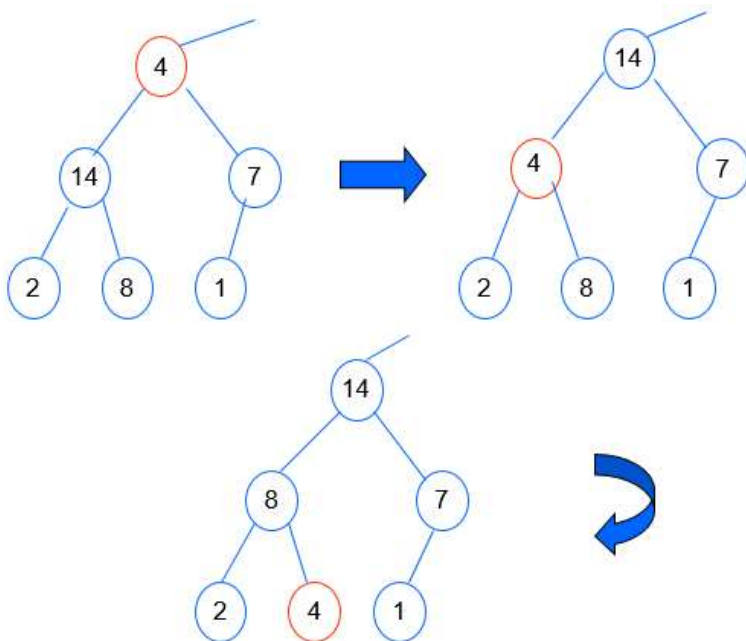
g) Repeat the steps from (c) to (f) until the subtrees are also get heapify.

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 4.
- 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)

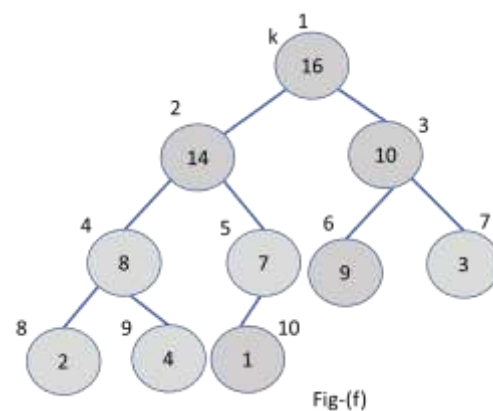
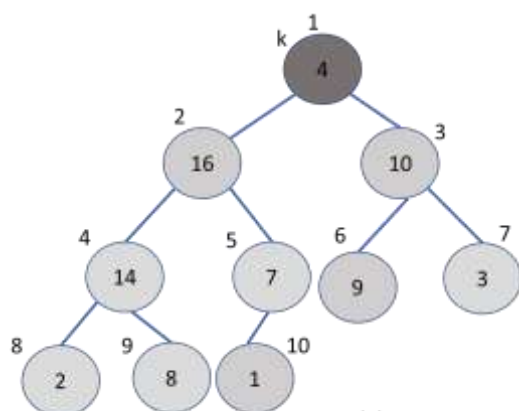
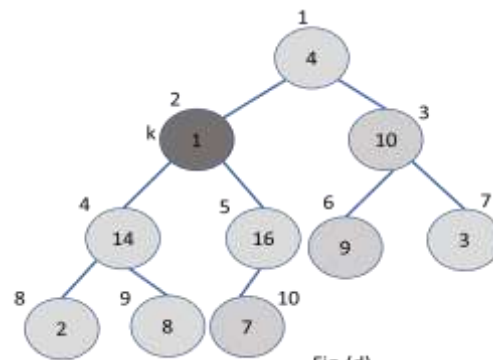
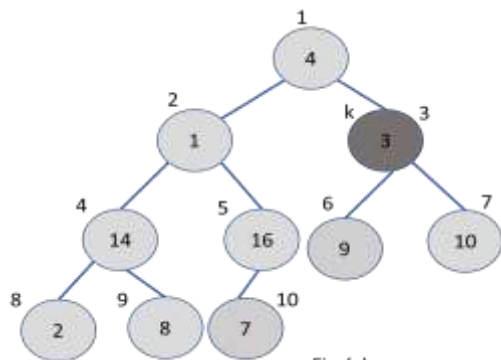
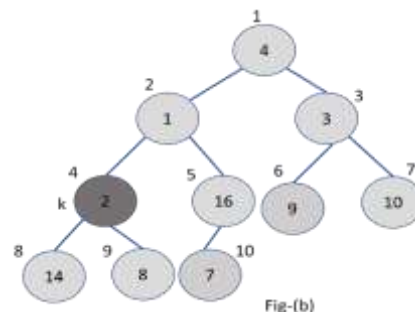
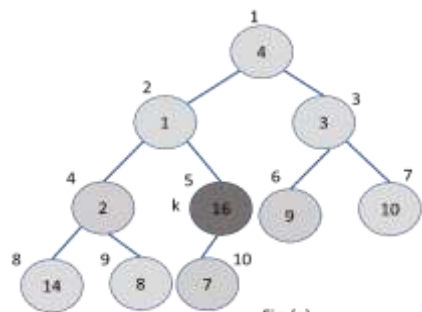
END;

Analysis

As we know that time complexity of Heapify operation depends on the height of the tree i.e. H and H should be $\lg N$ 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 $\theta(h)$.

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



	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:

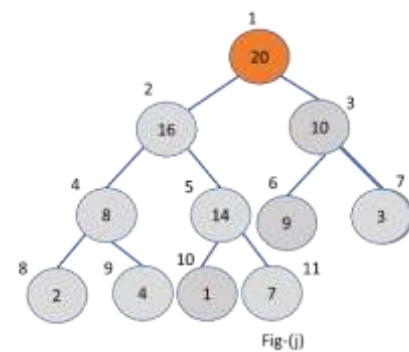
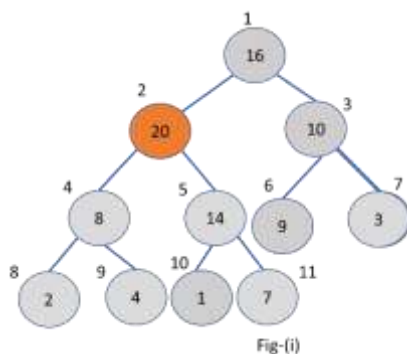
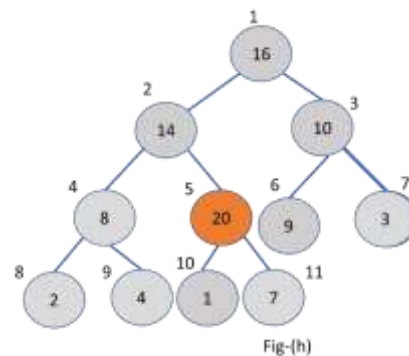
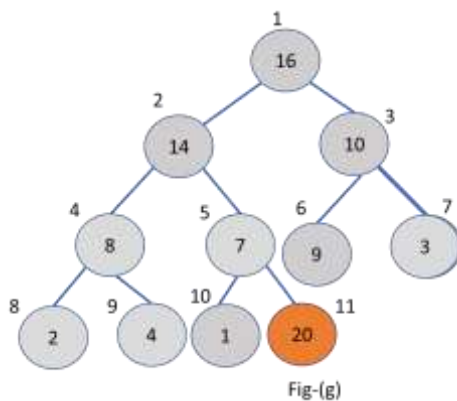
To insert an element in Max Heap.

Following are the steps to to insert an element in Max Heap.

- First update the size of the tree by adding 1 in the given size.
- Then insert the new element at the end of the tree.
- 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



	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:

$N = N + 1;$

$A[N] = \text{item};$

 ReHeapifyUp(A[], N, k)

END;

Algorithm: ReHeapifyUp(A[], N, k)

BEGIN:

$\text{parentindex} = k / 2;$

 IF $\text{parentindex} > 0$ THEN

 IF $A[k] > A[\text{parentindex}]$ THEN

 Exchnage($A[k], A[\text{parentindex}]$)

 ReHeapifyUp(A, N, parentindex)

END;

Analysis:

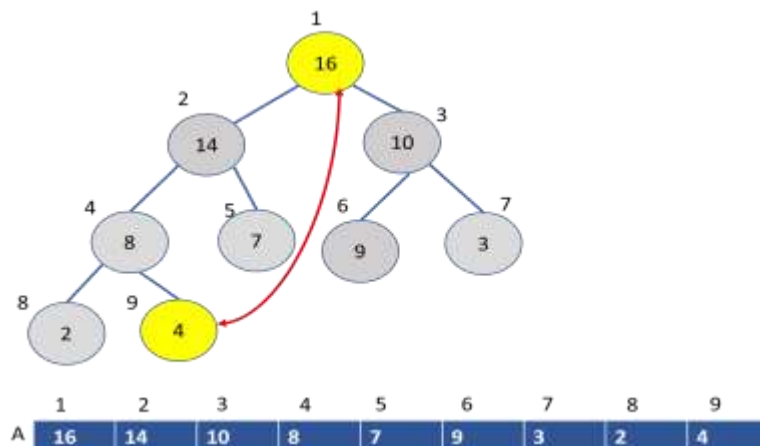
 Time Complexity: $\theta(\log n)$

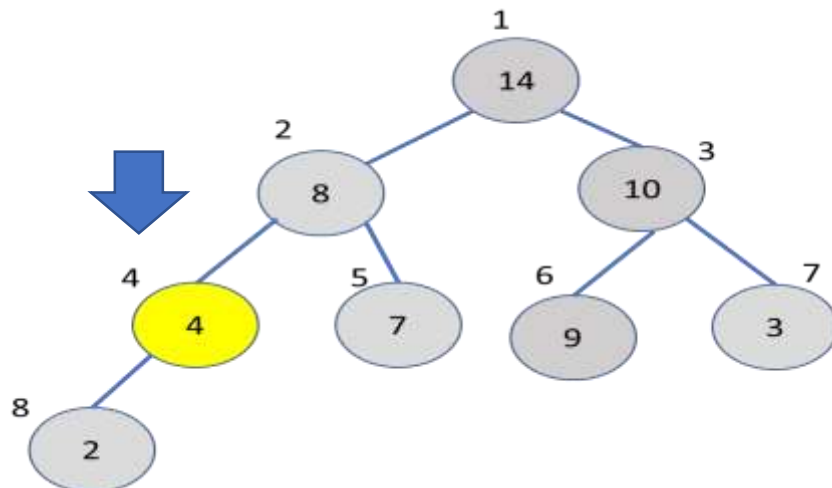
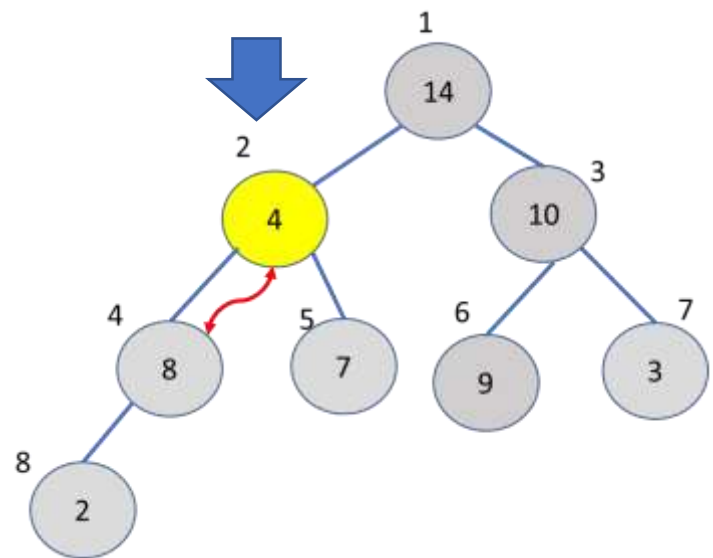
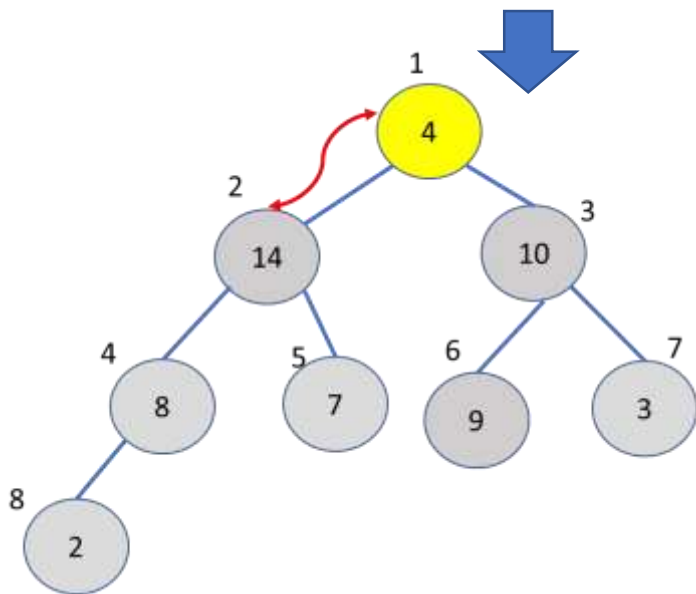
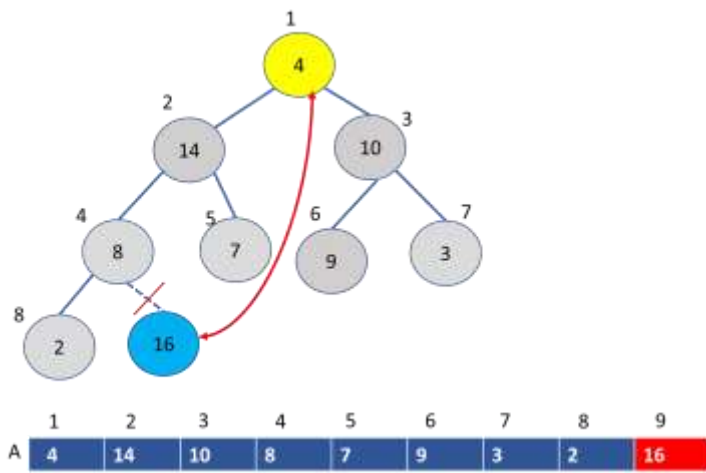
3 Delete Operation:

Method-1 To delete root element from Max Heap.

Following are the steps to delete an element from Max Heap.

- First exchange the root element with the last element in the heap..
- Then remove the last element by decreasing the size of the tree by 1.
- Now perform Heapify operation to make the tree either as max heap or min heap.





Method-2 To delete an element at any position from Max Heap.

Following are the steps to delete an element from Max Heap.

- First pick the element to be deleted.
- Now exchange it with the last element.
- Then remove this element by decreasing the size of the tree by 1.
- 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: $\theta(\log n)$

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.

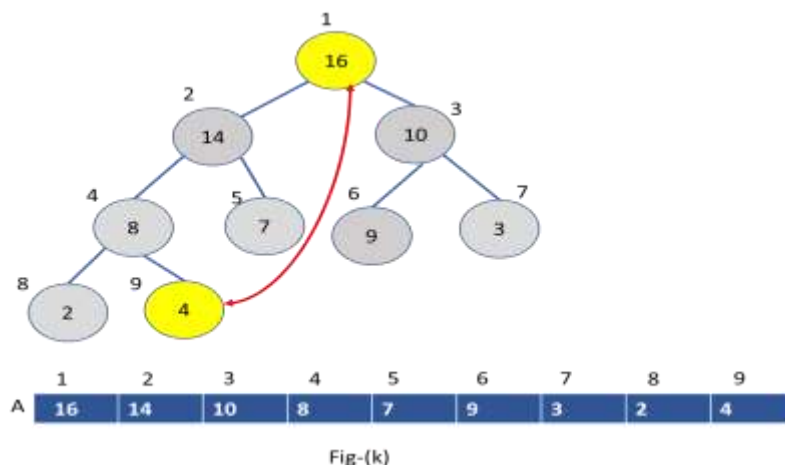
Step 2: Now exchange the root element with the last element in the heap.

Step 3: Decrease 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.

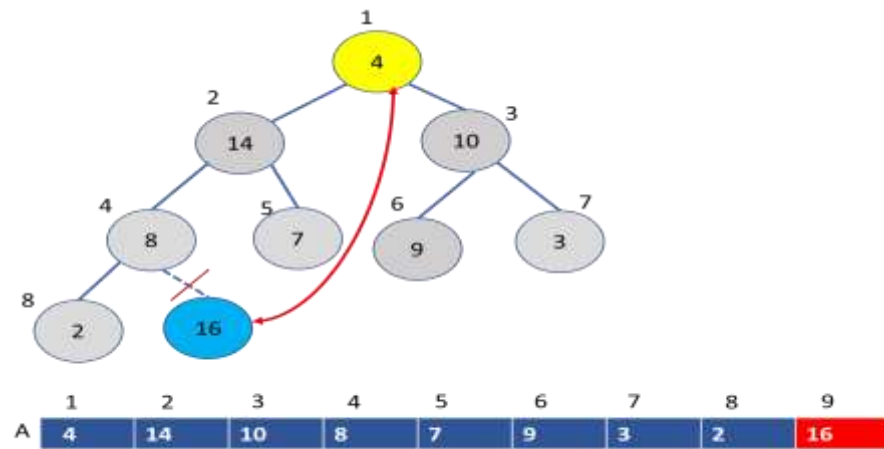
Example:

Step1: In the Max-Heap largest item is always stored at the root node.



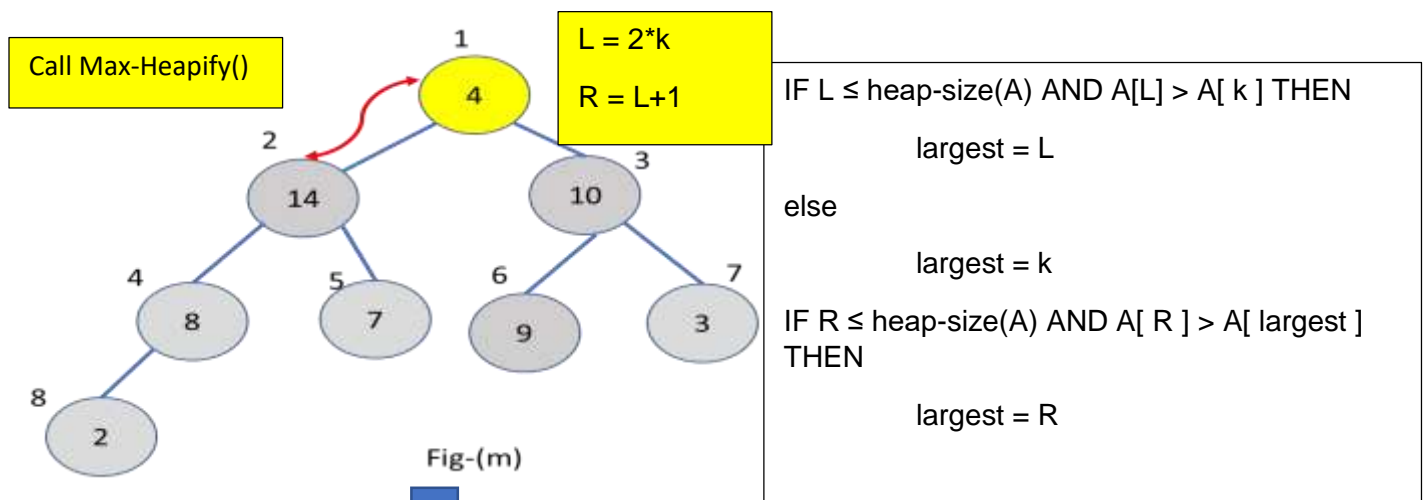
Step1: Replace
A[1] with A[last].

. Step 2 Now exchange the root element with the last element

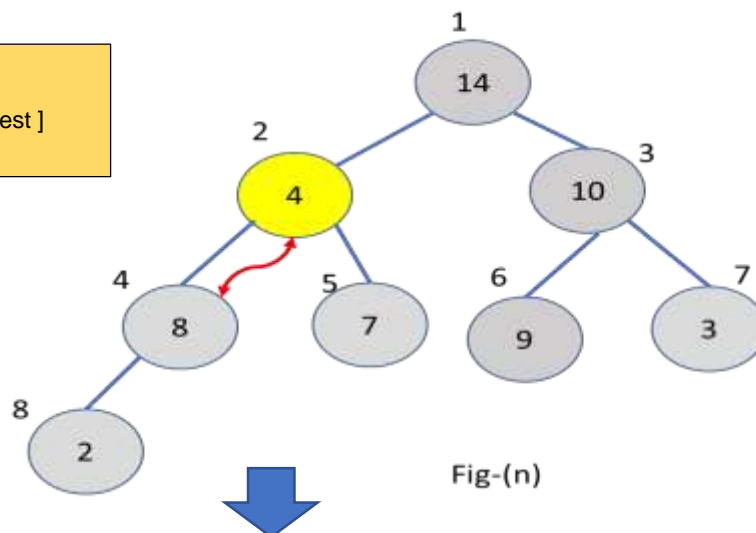


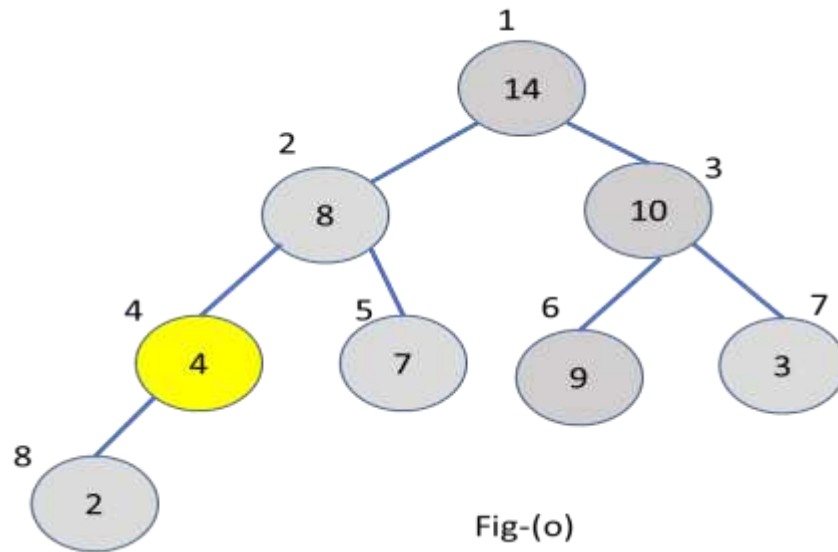
Step3 Decrease the size of the heap by 1 .

Step4 Now perform Max-Heapify operation so that highest element should arrive at root.



IF largest != k THEN
Exchange A[k] with A[largest]



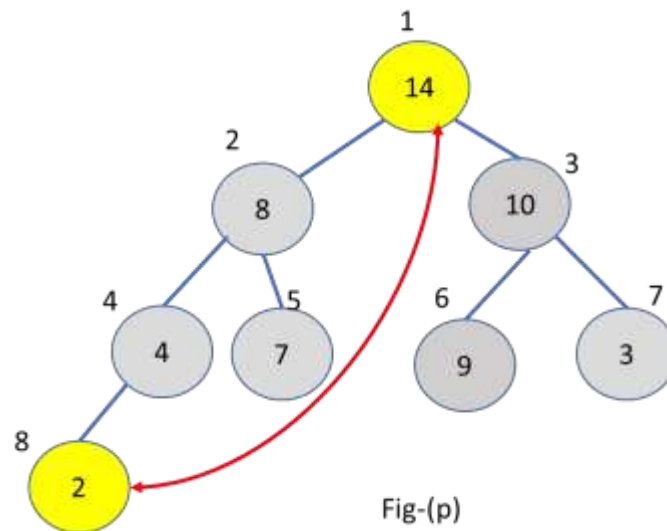


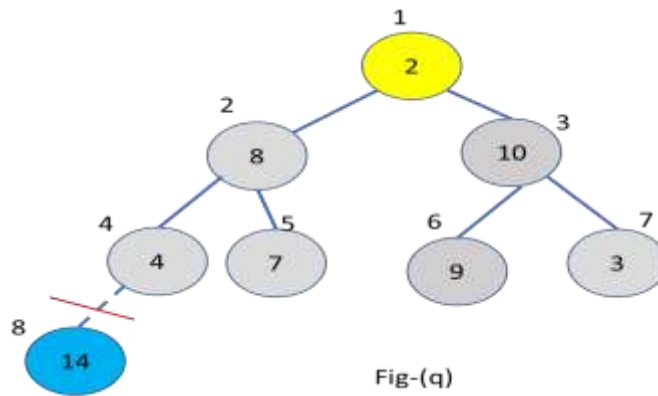
A

1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	16

Now this is the Max-heap after performing first deletion of root element.

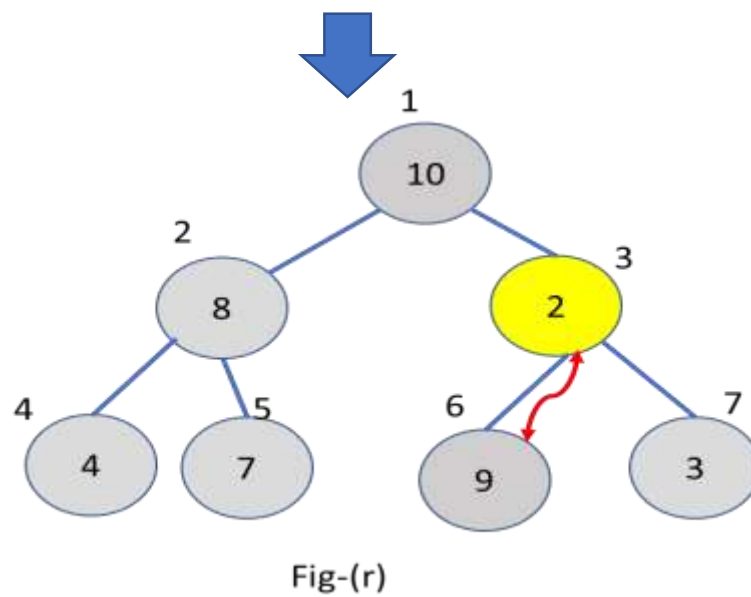
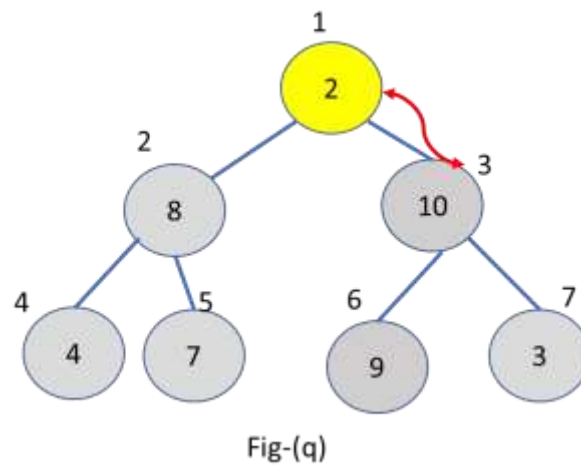
Repeat these steps until all the items are sorted.





	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.



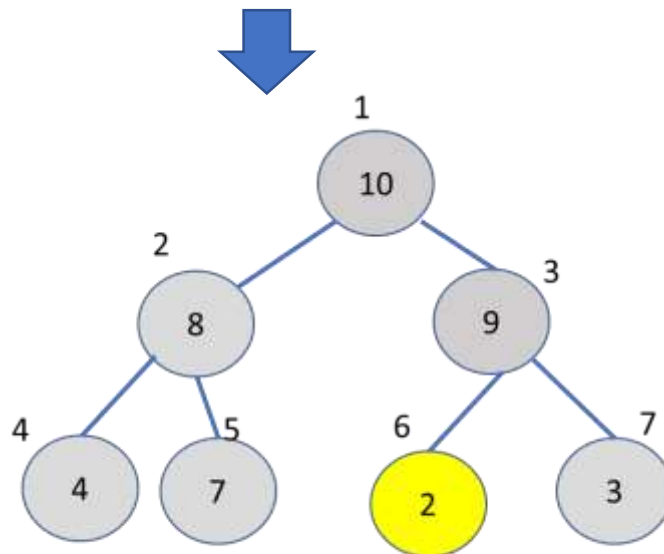


Fig-(s)

	1	2	3	4	5	6	7	8	9
A	10	8	9	4	7	2	3	14	16

We repeat the process until there is only one element left in the tree:

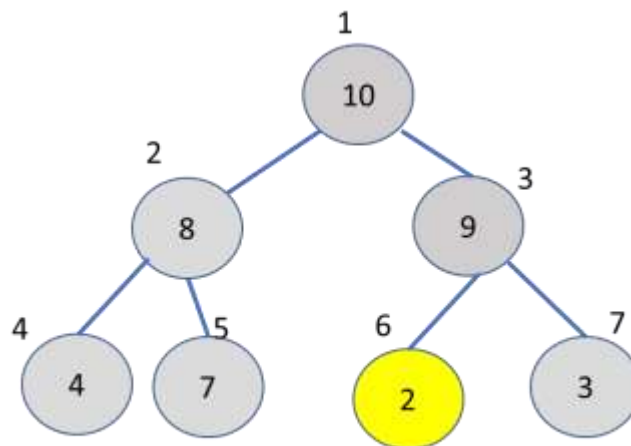


Fig-(t)

	1	2	3	4	5	6	7	8	9
A	2	3	4	7	8	9	10	14	16

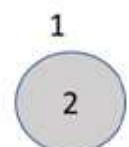


Fig-(u)

This last element is the smallest element and remains at the beginning of the array. The algorithm is finished, as the array is sorted:

	1	2	3	4	5	6	7	8	9
A	2	3	4	7	8	9	10	14	16

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 $\theta(n \log n)$ 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 $n \log n$.

It performs sorting in $\theta(1)$ 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.

. Coding Problems on Heap Sort

1) Cube Change Problem

Chandan gave his son a cube with side N. The $N \times N \times N$ cube is made up of small $1 \times 1 \times 1$ 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 \(hackerearth.com\)](https://www.hackerearth.com/practice-problems/algorithm/cube-scrap-off/)

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 \(hackerearth.com\)](https://www.hackerearth.com/practice-problems/algorithm/raghu-vs-sayan/)

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 \(hackerearth.com\)](https://www.hackerearth.com/practice-problems/algorithm/divide-apples/)

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.

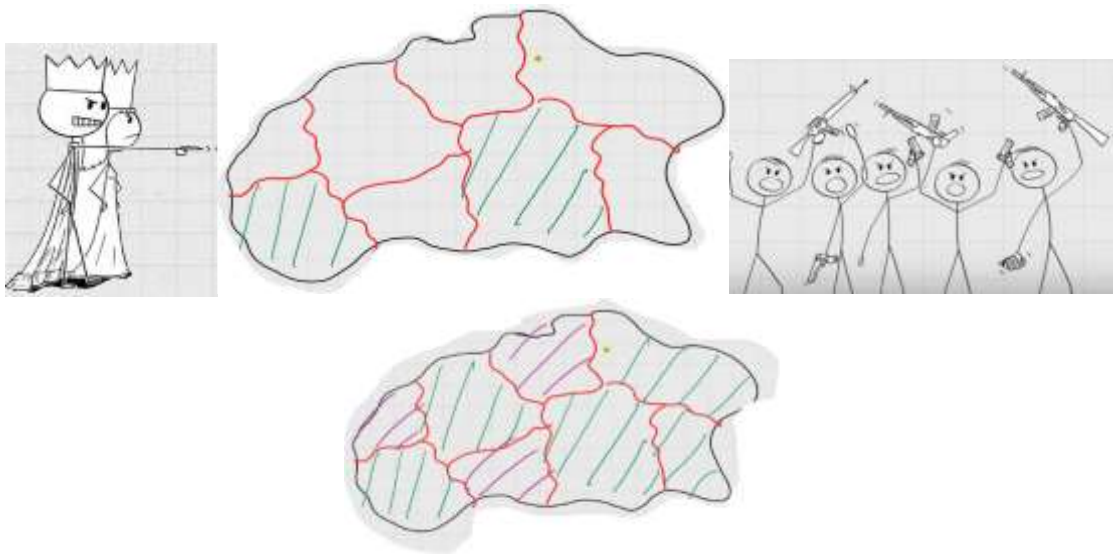


Figure 1: Elaboration about the King Conquering the Territory.

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 $\frac{1}{8}$ th $\frac{1}{8}$ th $\frac{1}{8}$ th 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 $\frac{1}{2}$ into two equal halves that is into two $\frac{1}{4}$ equal halves, further dividing these $\frac{1}{4}$ pieces will result in getting four equal $\frac{1}{8}$ pieces. Similarly, the second $\frac{1}{2}$ bread loaf can be further divided into two $\frac{1}{4}$ pieces and then further it can be divided into four $\frac{1}{8}$ equal pieces. Hence, total eight equal size pieces is divided among 8 beggars :

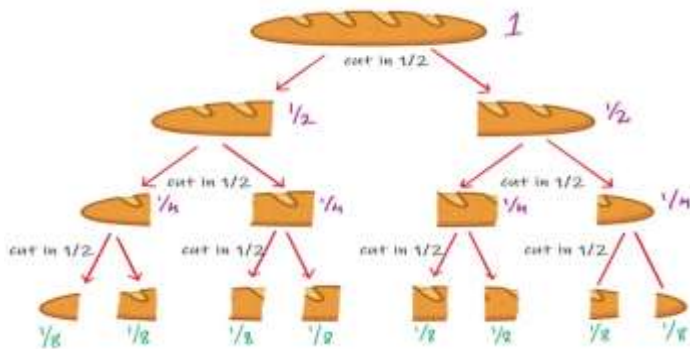


Figure 4: Divide the array into sub-arrays

Conclusion:-The manner in which the bread loaf was broken down into subparts is termed as a Divide procedure. The moment when it cannot be further subdivided 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 works 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 $A[p \dots q]$ and then from $A[q+1 \dots r]$. 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 $2+3+4+\dots+n$ times in each iteration. Hence, we can say that the running time complexity of Insertion Sort in worst case is $O(n^2)$.

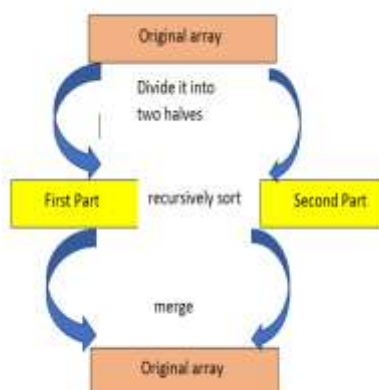


Figure 6: Divide and Conquer

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:

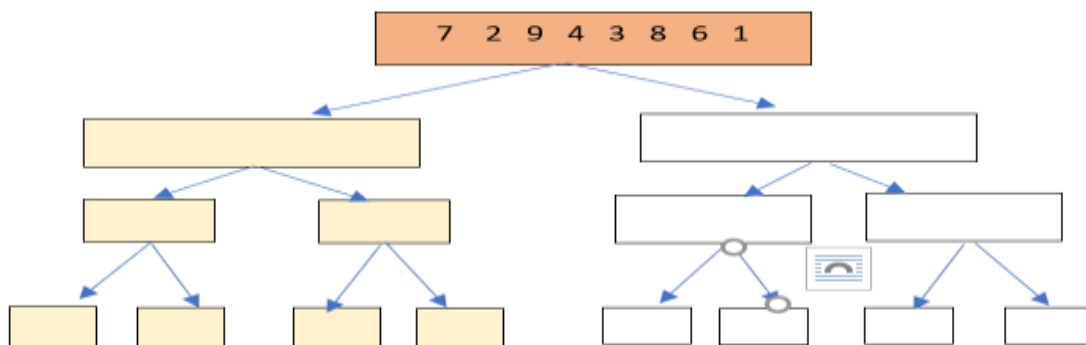
Theoretical View along with the Diagrammatic representation:

Merge sort is a sorting technique based on divide and conquer technique.

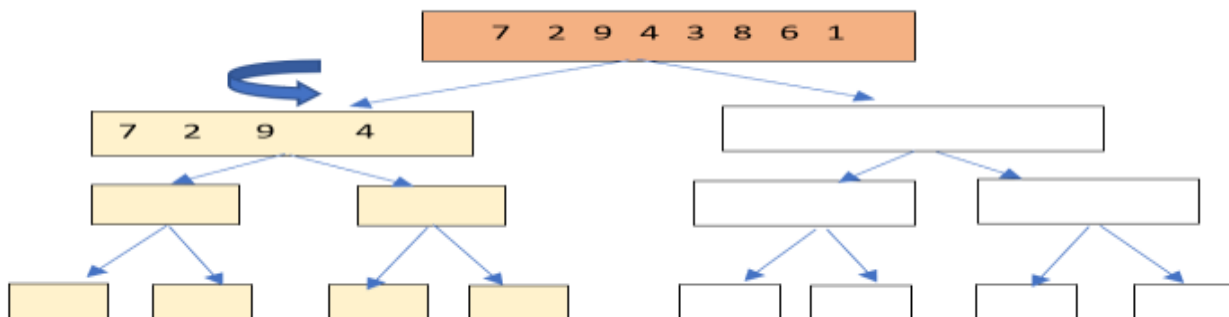
How Merge Sort Works: The Merge Sort algorithm has two procedures. These are calling of Merge function and merging procedure. Let's us understand the merging procedure before going towards calling procedure of Merge Sort.

Flow of Merge Sort:

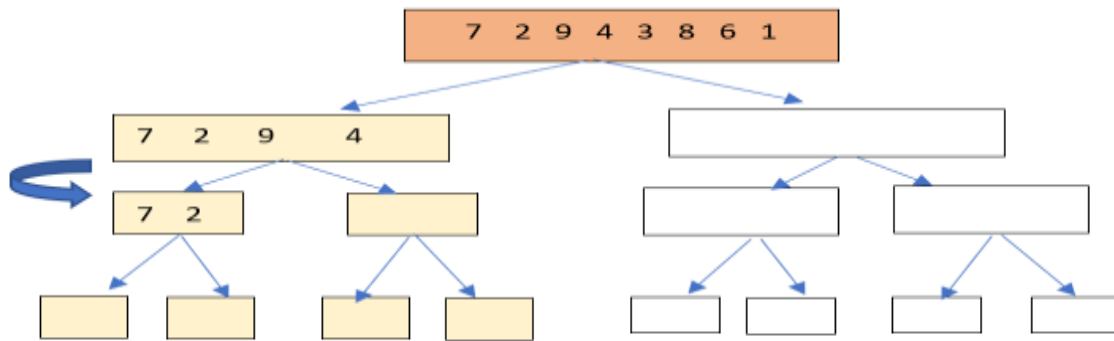
Step 1: Divide the Array of 8 Elements into two equal halves.



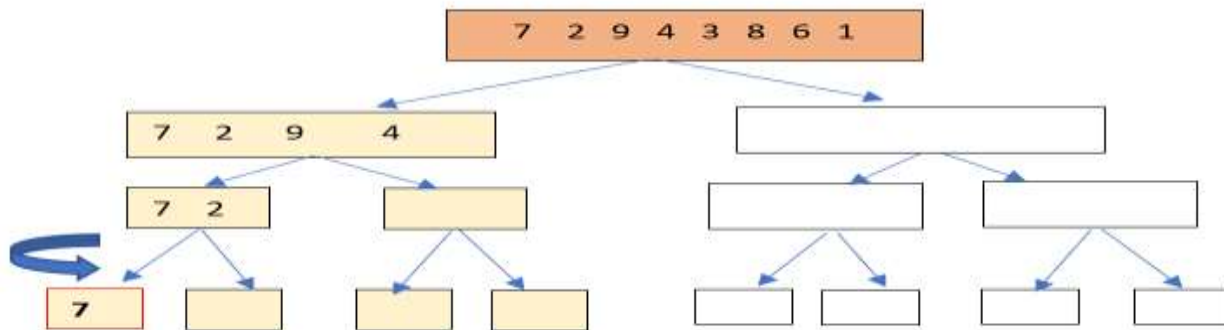
Step 2: Recursive Call and Partition:



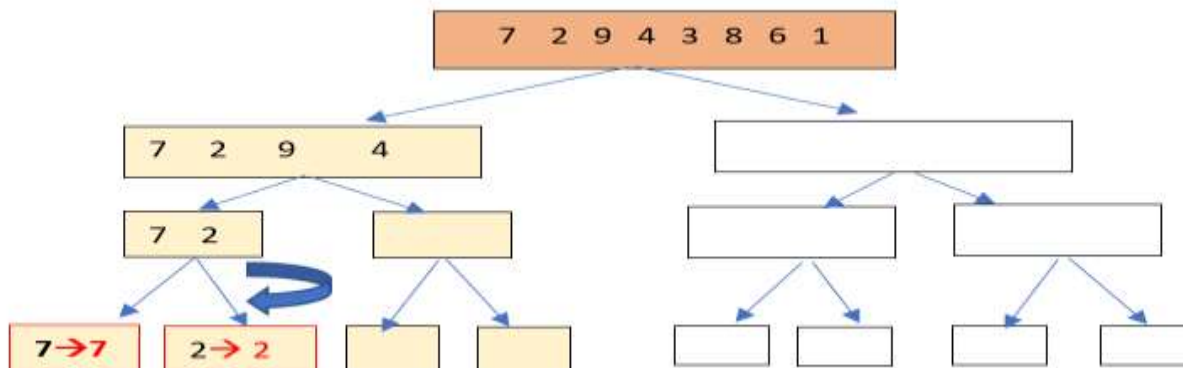
Step 3: Recursive Call and Partition:



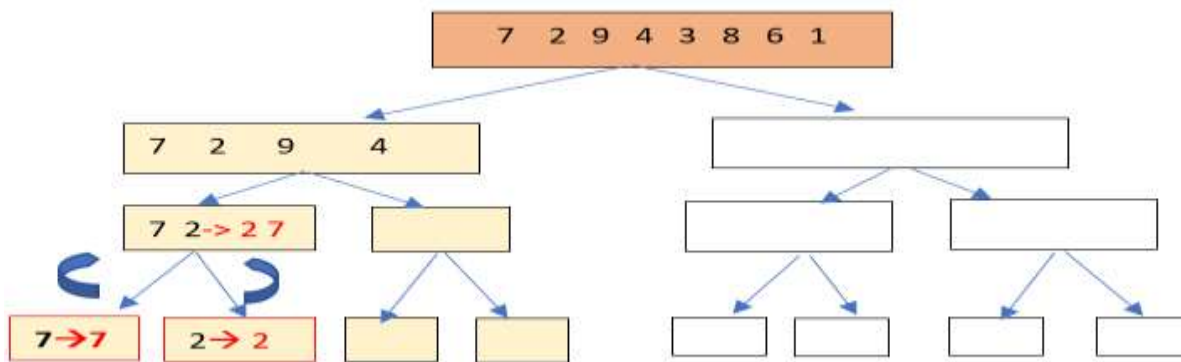
Step 4: Recursive Call and Base Case:



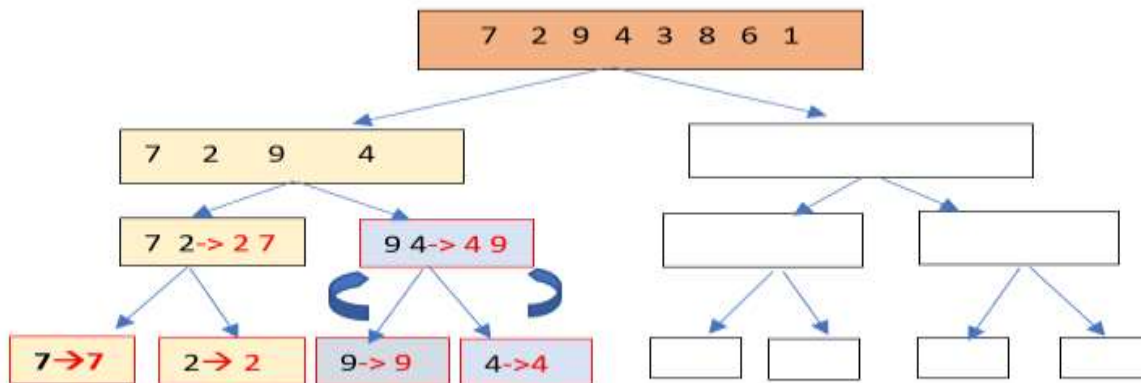
Step 5: Recursive Call and Base Case:



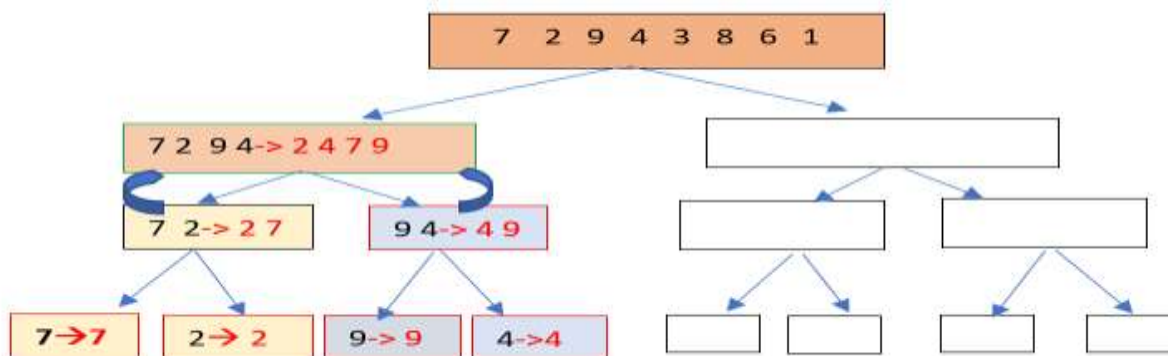
Step 6: Merge:



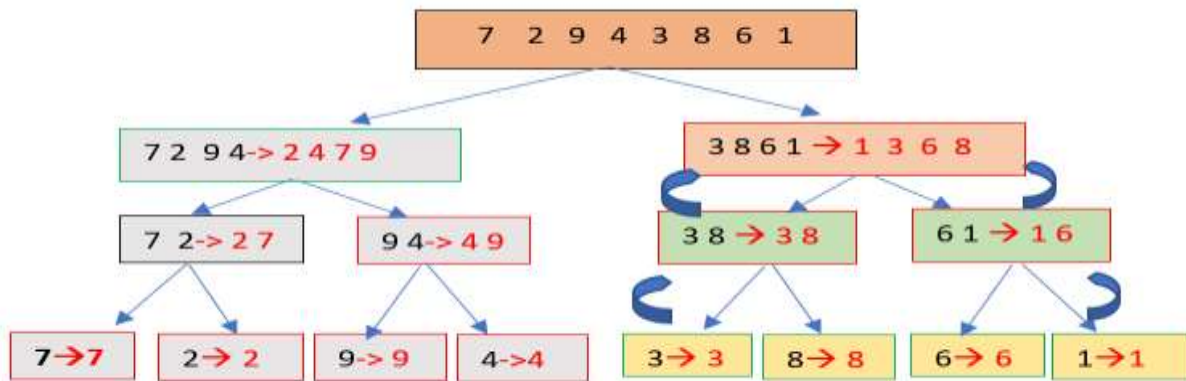
Step 7: Recursive Call, Base Case, Merge:



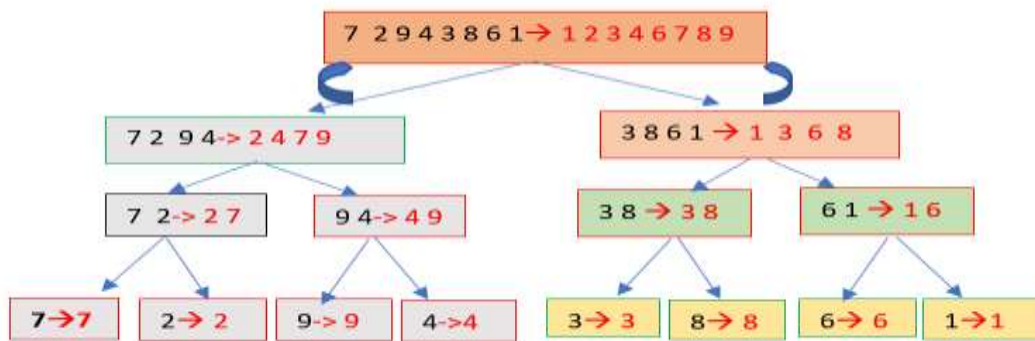
Step 8: Merge:



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. $L[n1+1] = \infty$
9. $R[n2+1] = \infty$
10. $i = 1$
11. $j = 1$
12. FOR $k=p$ TO r DO
13. IF $L[i] \leq R[j]$ THEN
14. $A[k]=L[i]$
15. $i = i + 1$
16. ELSE $A[k] = R[j]$
17. $j = j + 1$

END;

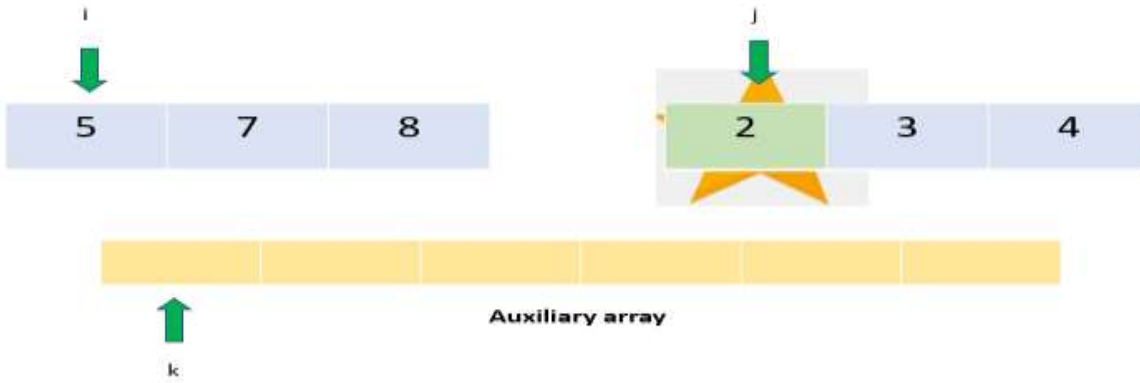
Step 1 and Step 2 is used to divide the array into sub-arrays. In Step 3, we will create left and right sub-arrays along with the memory allocation. From First three steps, running time complexity is constant $\theta(1)$. In Step 4 and 5, Independent for loop is used for assigning Array elements to the 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 $\theta(n_1 + n_2)$. 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 $\theta(n)$.

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:

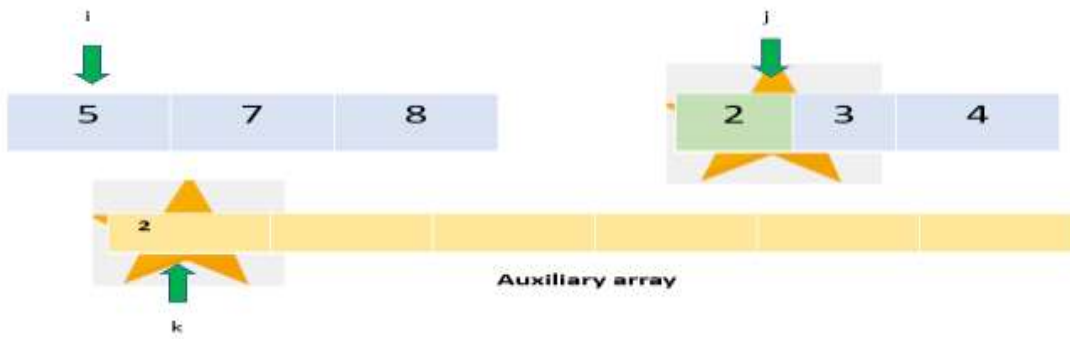


Now let's go through step by step:-

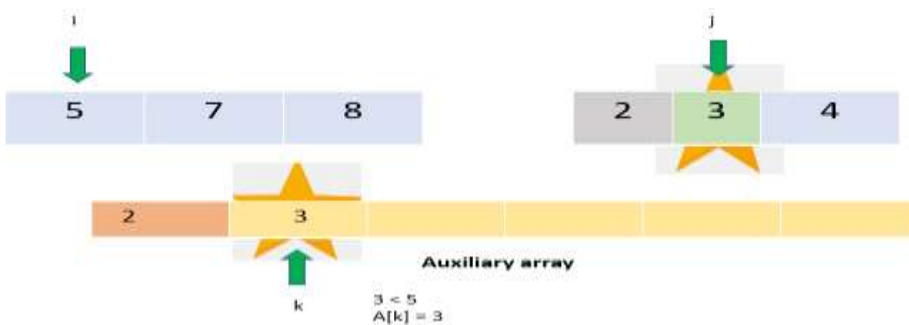
Step-01: Two sub array Left and Right is created, number of elements contained in Left sub array is given by index i and number of elements contained in Right sub array is given by index j . Number of elements contained in Left array will be $L[i] = q - p + 1$ and number of elements in right most array will be $R[j] = r - q$.



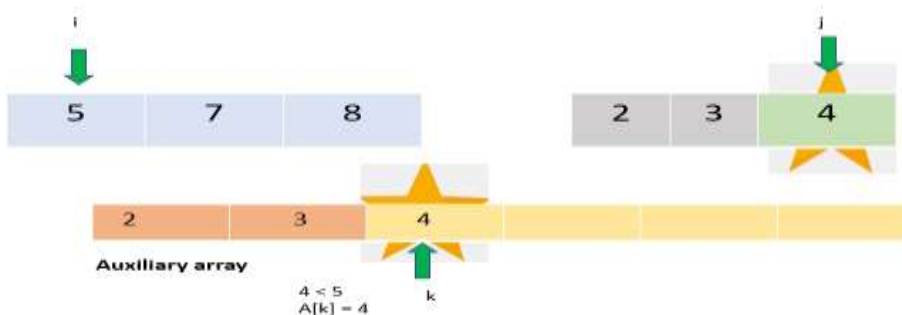
Step-02: Initially i , j and k will be 0. As element in right array $R[0]$ is 2 which is less than $L[0]$ that is 5. So, $A[k]$ will hold 2.



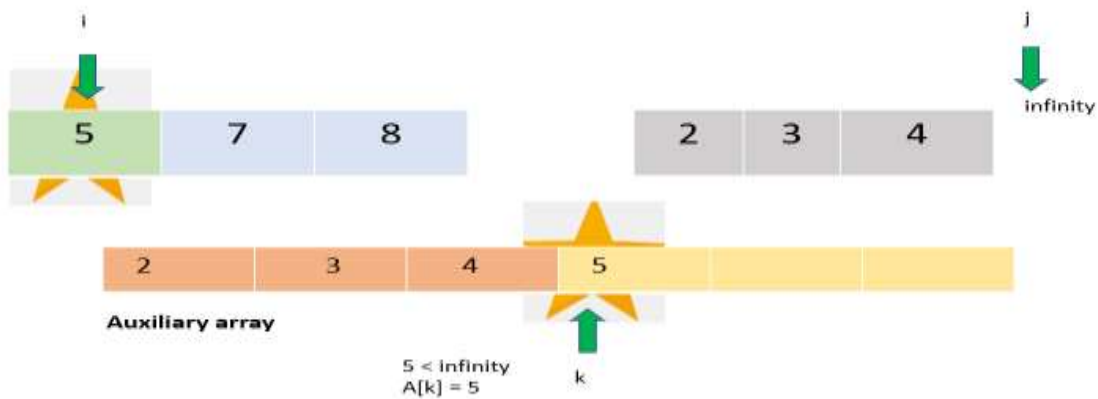
Step-03: Now $j = 1$, $i = 0$ and $k = 1$. Now $R[j]$ will point to 3 and $L[i]$ will point to 5, since 3 is less than 5 $A[k]$ will hold 3.



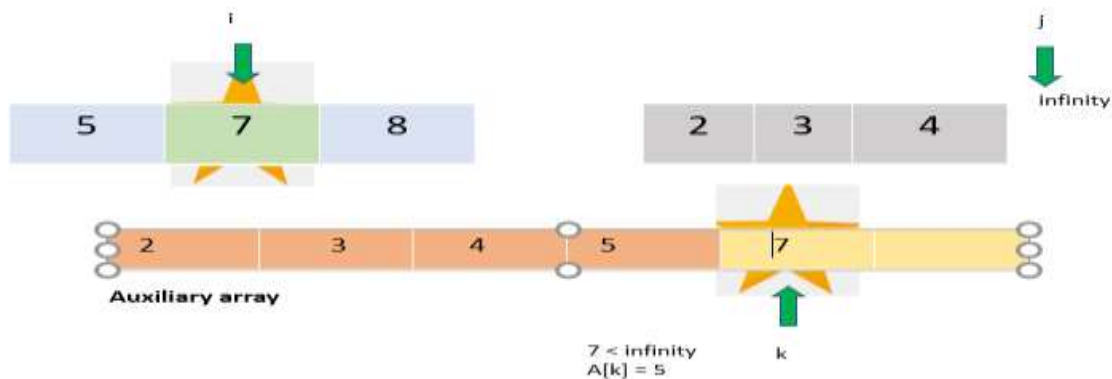
Step 4: Now $j = 2$, $k = 2$, $i = 0$. Since $R[j]$ will point to 4 and $L[i]$ is still at 5 so $B[2]$ will now contain 4.



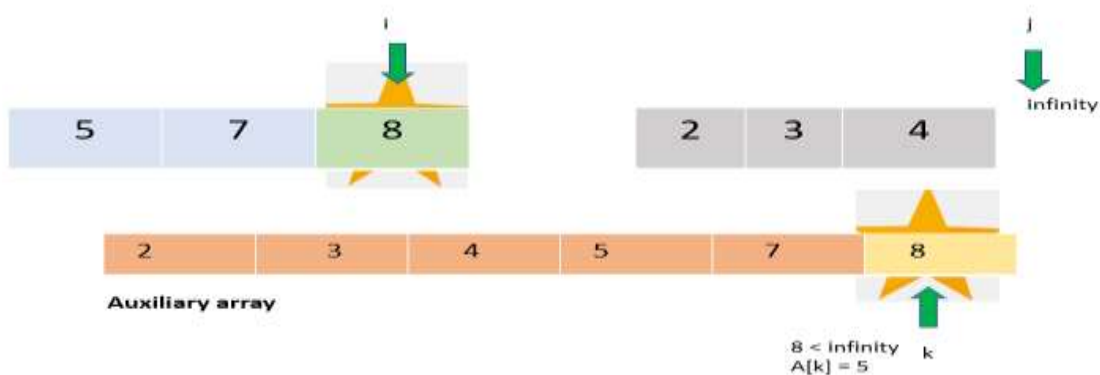
Step-05: Now $i = 1$, j points to sentinel which stores infinity, $k = 3$. Now $A[k]$ will be 5. Thus, we have-



Step 6: Now $i = 2$, j points to sentinel 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 sentinel 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


```

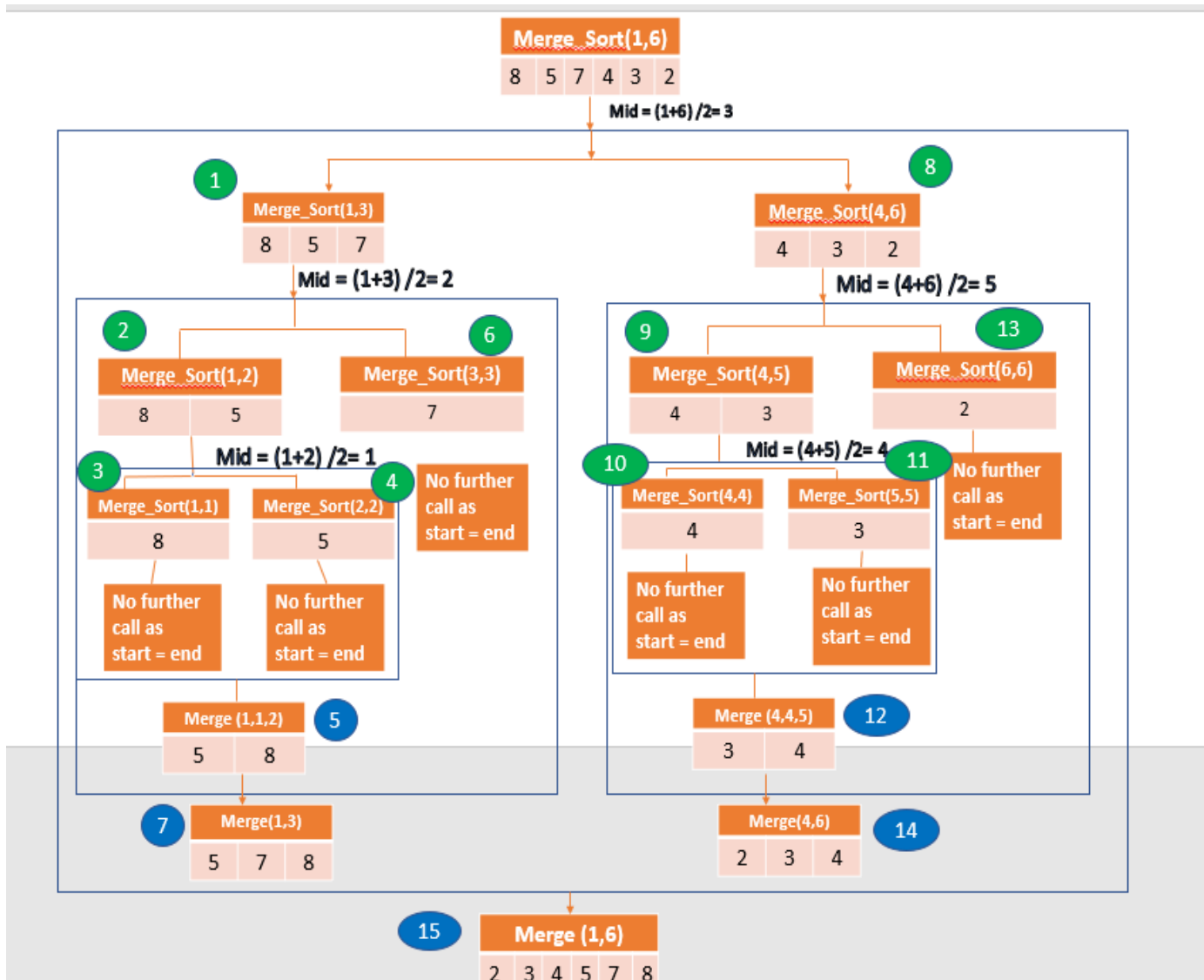
q =  $\lfloor (p + r) / 2 \rfloor$ 
MergeSort(A, p, r)
MergeSort(A, q+1, r)
Merge(A, p, q, r)

```

END;

To sort the entire sequence $A = _A[1], A[2], \dots, A[n]$, we make the initial call $\text{MERGESORT}(A, 1, \text{length}[A])$, where once again $\text{length}[A] = n$.

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.

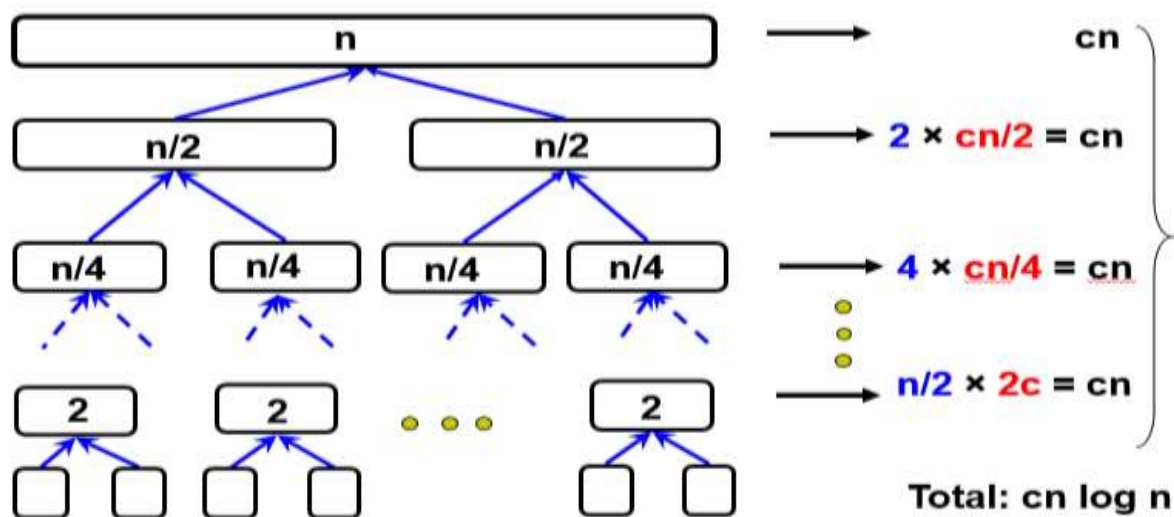
- **Combine:** Here we are sorting n elements present in array, so $C(n) = \Theta(n)$.

Hence the recurrence for the worst-case running time $T(n)$ of merge sort will be:

$$\begin{aligned} T(n) &= c \text{ If } n=1 \\ &= 2T(n/2) + cn \quad \text{If } n \text{ is greater than } 1 \end{aligned}$$

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 $O(n \log n)$.



The **time complexity of Merge Sort** is order of ($n \cdot \log n$) 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	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	$O(1)$
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	$O(N)$
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(N)$
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	$O(1)$

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

2.6.11. 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: $O(N \cdot \log N)$ 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 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 $j < \text{size of the second half}$. Add $(j - (\text{mid} + 1))$ 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 subsequences in 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 $2^{(\text{Number of elements less than arr[i]})}$.

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., 2^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: $O(N \log n)$ 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 its exact place the other in the same manner can be the pivot and find its 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 $A[j] \leq x$ THEN

$i = i+1$

 Exchange $A[i]$ with $A[j]$

 Exchange $A[i+1]$ with $A[r]$

 RETURN $i+1$

END;

2.7.4.Example to demonstrate working of Quick Sort:



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.



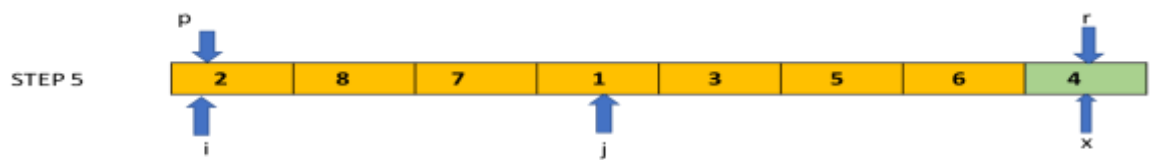
for $j=1$ $2 \leq 4$ True so i will increment by $i=1$ exchange will happen which is same for this.



for $j=2$ $8 \leq 4$ False so no change, j simply moves to next index



for $j=3$ $7 \leq 4$ False so no change, j simply moves to next index



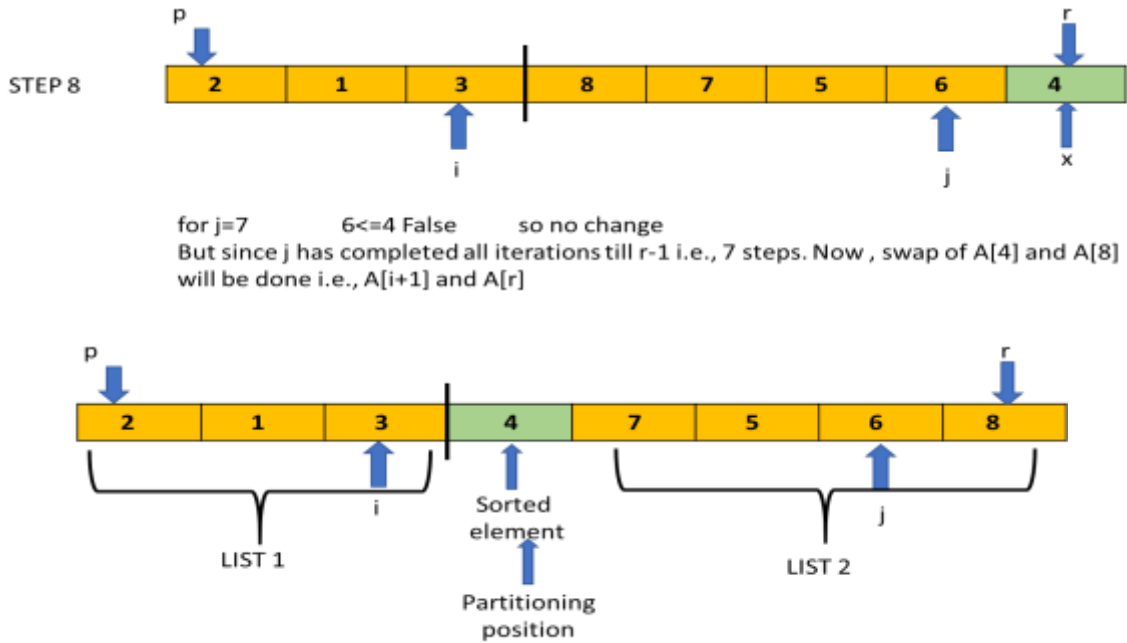
for $j=4$ $1 \leq 4$ True so i increments by one, $i=2$, exchange $A[2]$ and $A[4]$ i.e., $A[i]$ and $A[j]$



for $j=5$ $3 \leq 4$ True so i increments by one, $i=3$, exchange $A[3]$ and $A[5]$ i.e., $A[i]$ and $A[j]$



for $j=6$ $5 \leq 4$ False so no change



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  $p < r$ 
     $q = \text{Partition}(A, p, r)$ 
    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 $A[j] \leq x$	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:

$$F(n) = C1.1 + C2.1 + C3.n+1 + C4.n + C5.n + C6.1 + C7.1 + C8.1$$

$$F(n) = n(C3 + C4 + C5) + 1(C1 + C2 + C6 + C7 + C8)$$

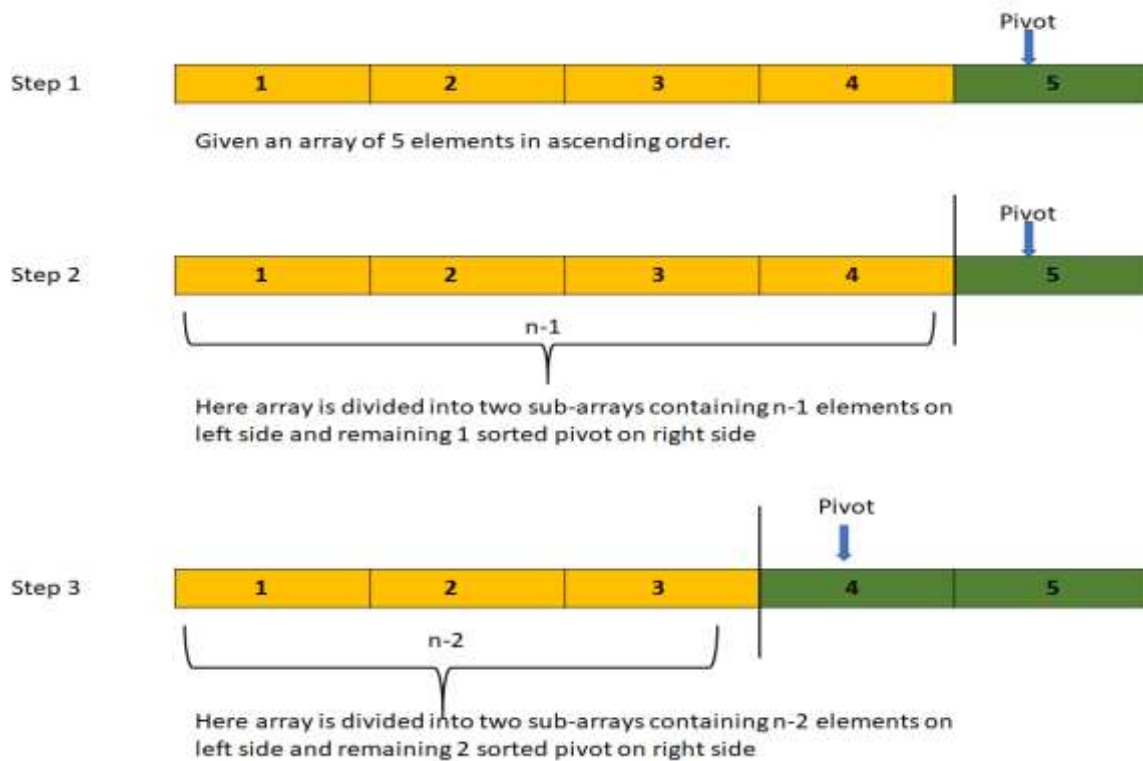
$$F(n) = a(n) + b$$

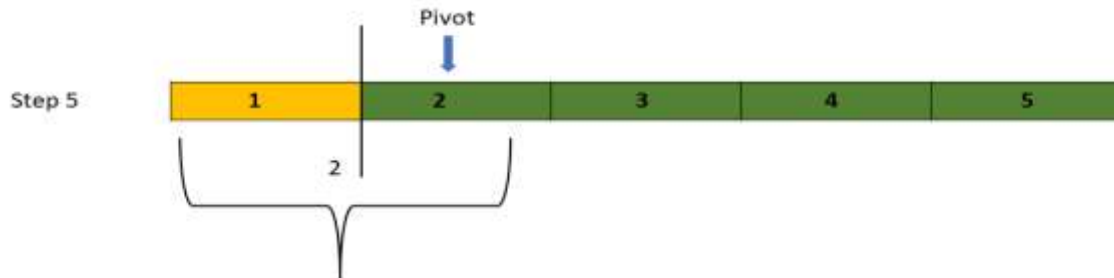
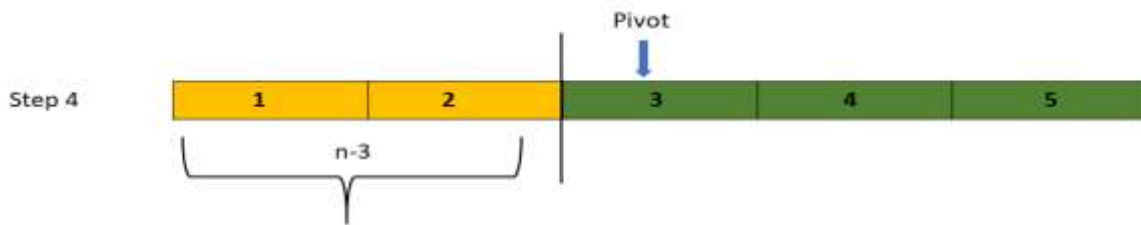
So, we can say that time complexity: $\theta(n)$ for partition algorithm. The space complexity of partition algorithm will be $\theta(1)$, as 5 extra variables are required.

2.7.5.Detailed Complexity Analysis of Quick Sort with the example:

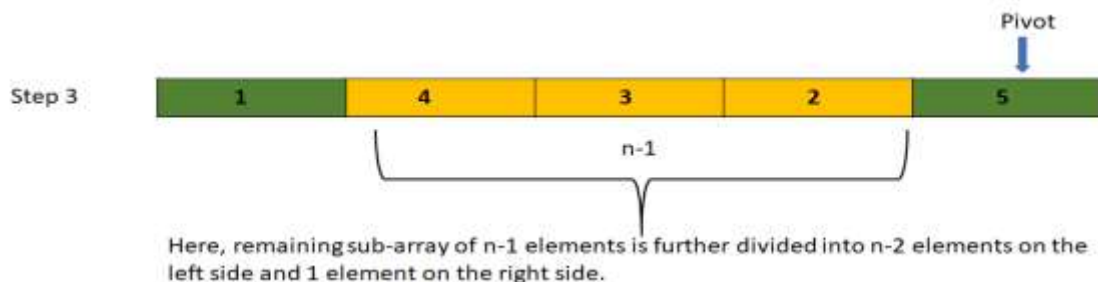
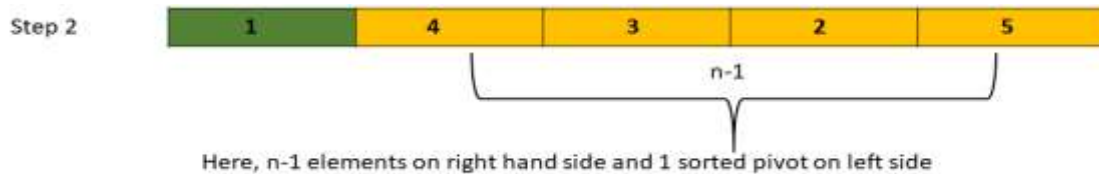
In the performance analysis of QuickSort selection of pivot element plays an important role. This can be classified as 3 cases in the form of input given.

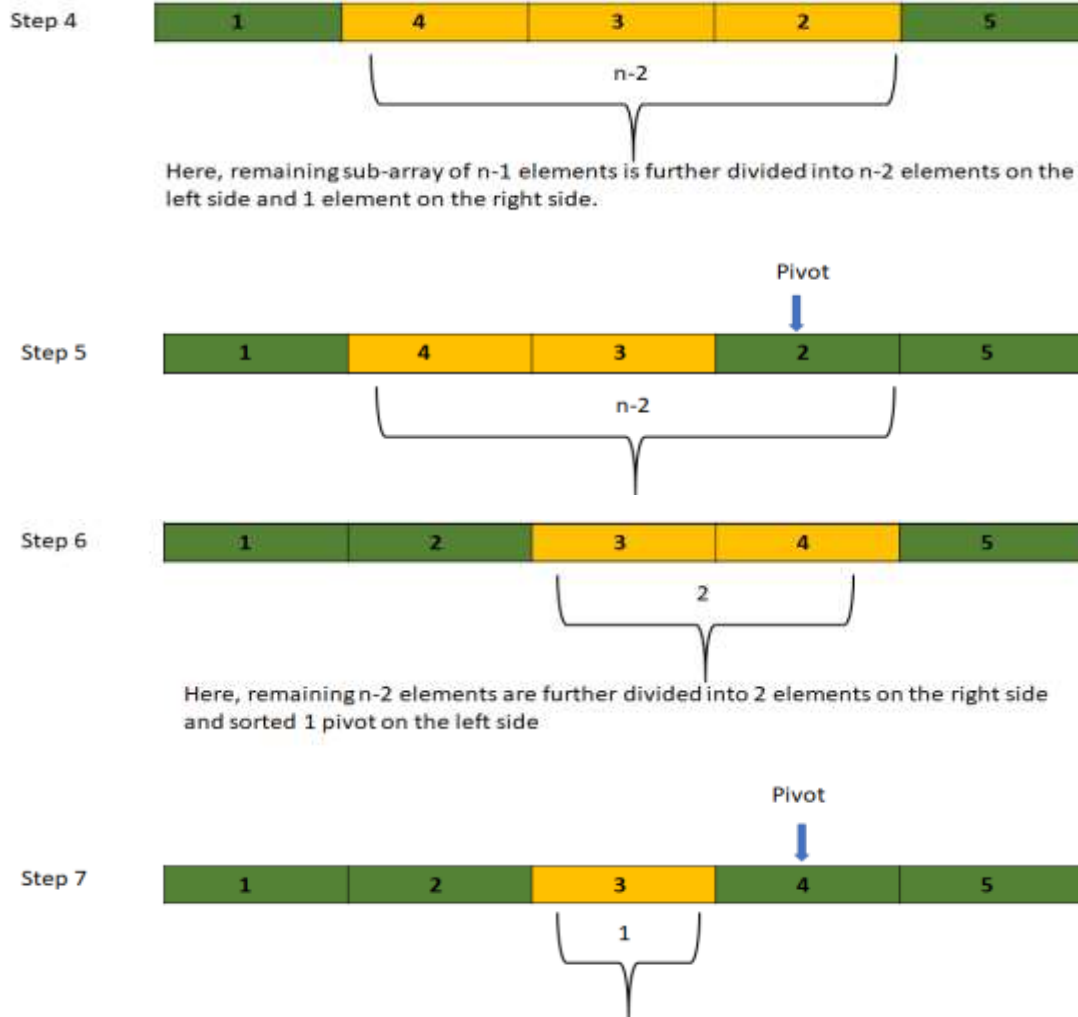
Case 1: a) When the input array is sorted in ascending order. Such a case experiences an unbalanced array split, with (n-1) elements on one side of an array and one as a sorted element (pivot element).





Case 1: b) When the input array is sorted in descending order. Given example below:



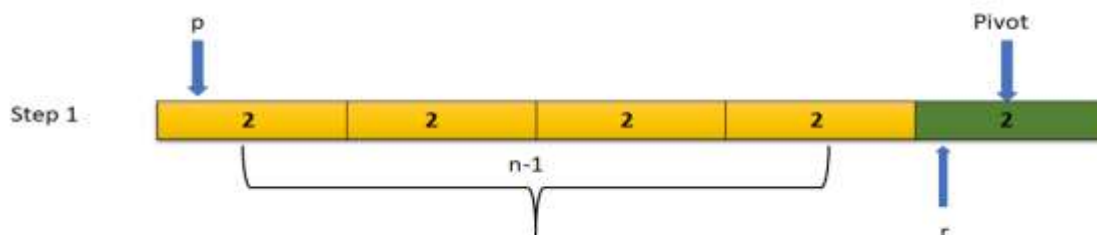


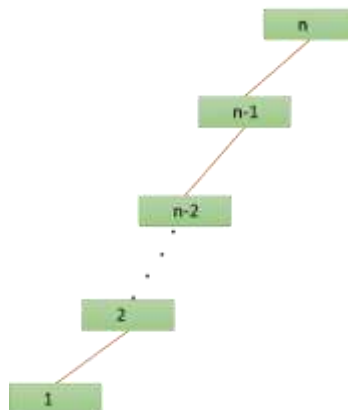
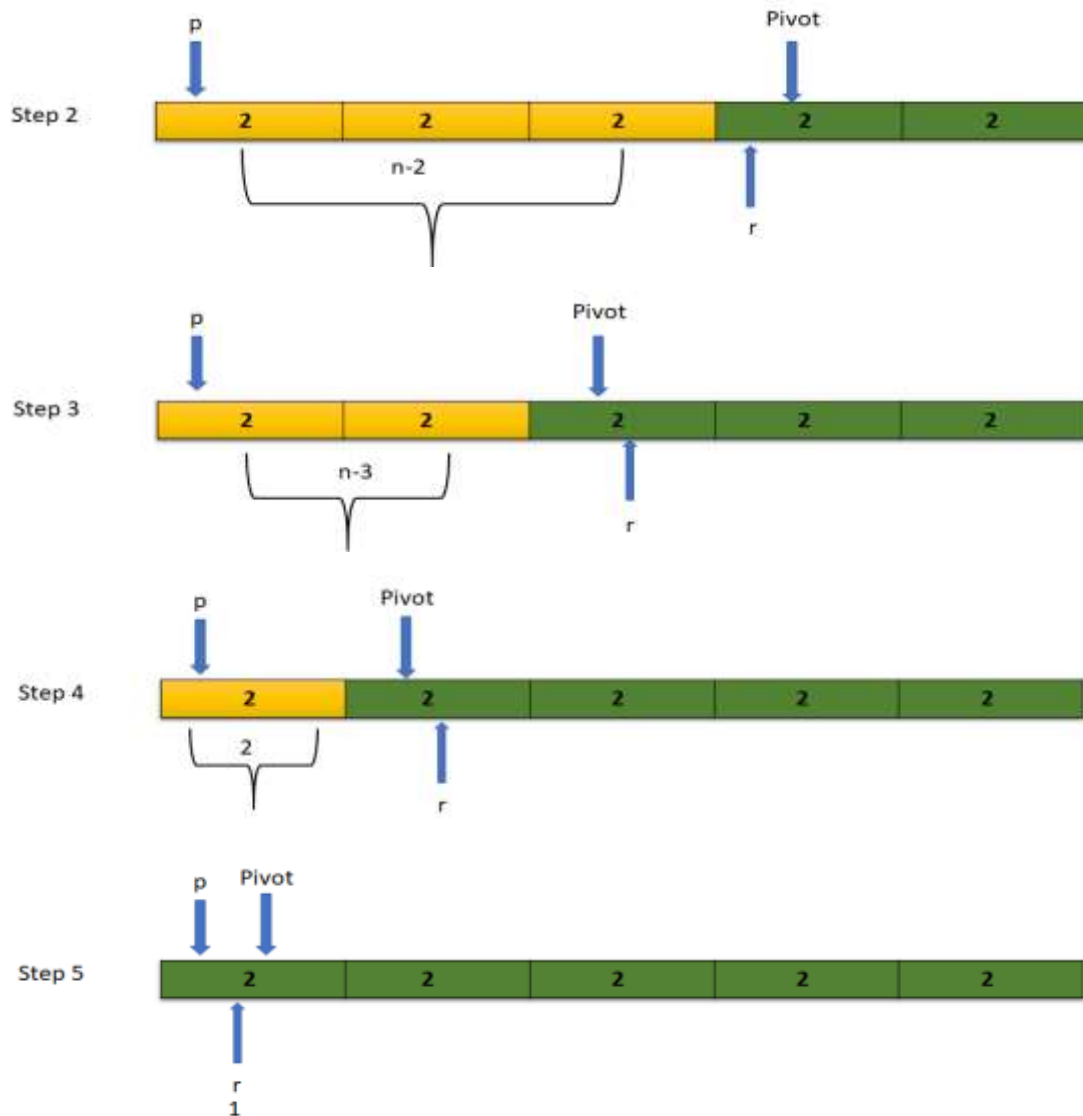
Here, worst case complexity of QuickSort can be explained either by the Recurrence relation or by iterative approach.

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 $[n + (n-1) + (n-2) + (n-3) + \dots + 2] = \frac{n(n+1)}{2} - 1 = O(n^2)$ **Worst case complexity.**

Case 1: c) When all elements in array are equal

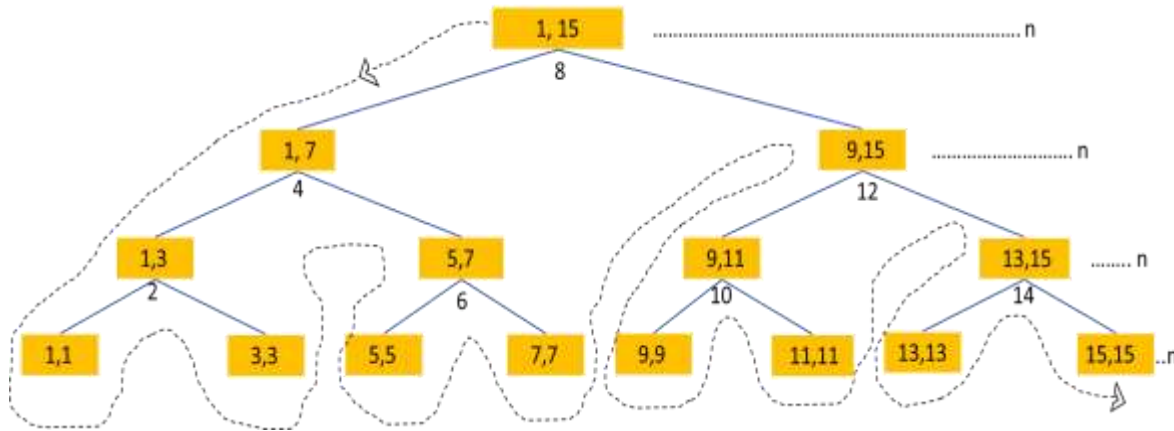




Thus, $[n + (n-1) + (n-2) + (n-3) + \dots + 1] = (n(n+1)/2 - 1) = O(n^2)$ worst case complexity.

Case 2: When the input array splits from the middle and partition occurs evenly from the middle at each level.

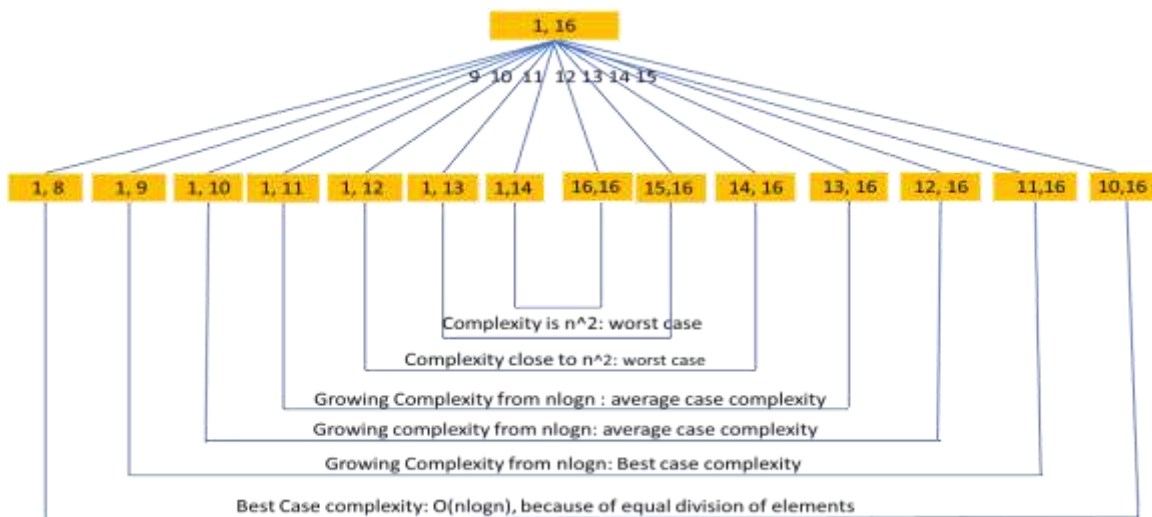
Suppose we have an array of 15 numbers $A[1:15]$. The best case is possible if:



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 $\Omega(n \log n)$. **Recurrence derived from the above mentioned approach is as follows: $T(n) = 2T(n/2) + \Theta(n)$** 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 $O(n \log n)$.

Case 3: Average Case Complexity:

$T(n) = 2T(n/2) + \Theta(n)$ 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 $O(n \log n)$. 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 **$T(n) = T(n-1) + \Theta(n)$** , 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 $T(n) = T(n/3) + T(2n/3) + \theta(n)$ which results in the complexity as order of $(n \log n)$. 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, $T(n) = T(n/4) + T(3n/4) + \theta(n)$, $T(n) = T(n/5) + T(4n/5) + \theta(n)$, $T(n) = T(n/6) + T(5n/6) + \theta(n)$ 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 $\theta(n)$ and Space Complexity is $\theta(1)$.

Partitioning Algorithm of Three Way Quick Sort:

ALGORITHM Partition (arr[], left, right, i, j)

BEGIN:

```
IF right – left <= 1 THEN
    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}

																			Pivot
24	97	40	67	88	85	15	66	53	44	26	48	16	52	45	23	90	18	49	80
Quick Sort is applied when length of the array is 10 or greater																			
																			Pivot
24	40	67	15	66	53	44	26	48	16	52	45	23	18	49	80	90	88	85	97
Quick Sort is applied when length of the array is 10 or greater															Insertion Sort				
																			Pivot
24	40	15	44	26	48	16	45	23	18	49	66	53	67	52	80	85	88	90	97
Quick Sort is applied when length of the array is 10										Insertion Sort					Sorted Array				
15	16	18	44	26	48	40	45	23	24	49	52	53	66	67	80	85	88	90	97
Insertion		Insertion Sort								Sorted Array									
15	16	18	23	24	26	40	44	45	48	49	52	53	66	67	80	85	88	90	97
Sorted Array																			

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 4th 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 3rd and 4th 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 $\theta(n \log n)$ 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[i] is special if there exists a contiguous sub-array that contains exactly k numbers that are strictly greater than A[i]. 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 $\leq E$. 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 then 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 a_i . Second friend got a deck with m cards. The i -th his card has strength b_i .

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.

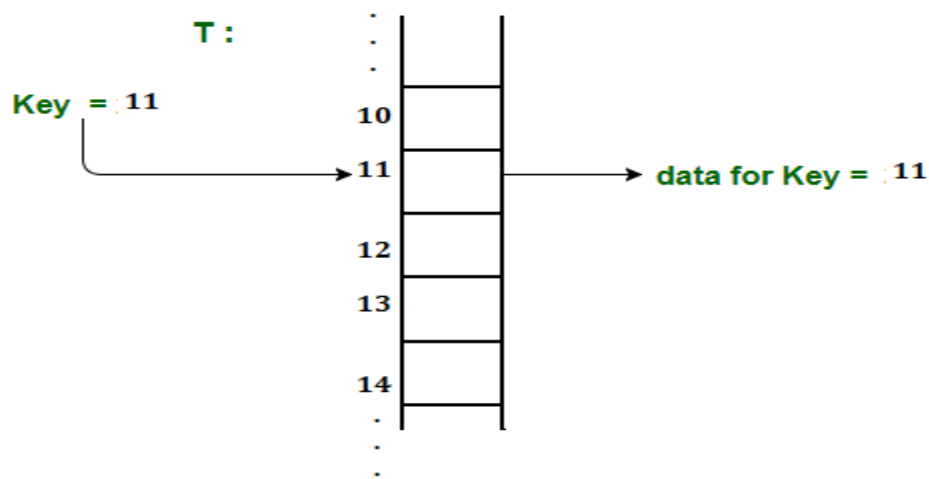
[Practice Problem \(hackerearth.com\)](https://hackerearth.com)

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:

- $A[1..n]$,
 $A[j] \in \{1, 2, \dots, k\}$

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 $\Theta(k)$, the second for loop takes time $\Theta(n)$, third for loop takes time $\Theta(k)$, and the fourth for loop takes time $\Theta(n)$. Thus, the overall time is $\Theta(k+n)$. In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$.

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

2.8.6.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 $h_1 - h_2$

 //two unique heights let h_1 and h_2 and also $h_1 > h_2$

IF $c == 3$ THEN

 IF $h_3 - h_2 == h_2 - h_1$ THEN // $h_1 < h_2 < h_3$

 WRITE("Height becomes equal")

 RETURN $h_3 - h_2$

 ELSE

 WRITE("Height did not become equal")

 RETURN

Time Complexity- $\theta(n)$

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.

2.9.1 Analogy:- 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

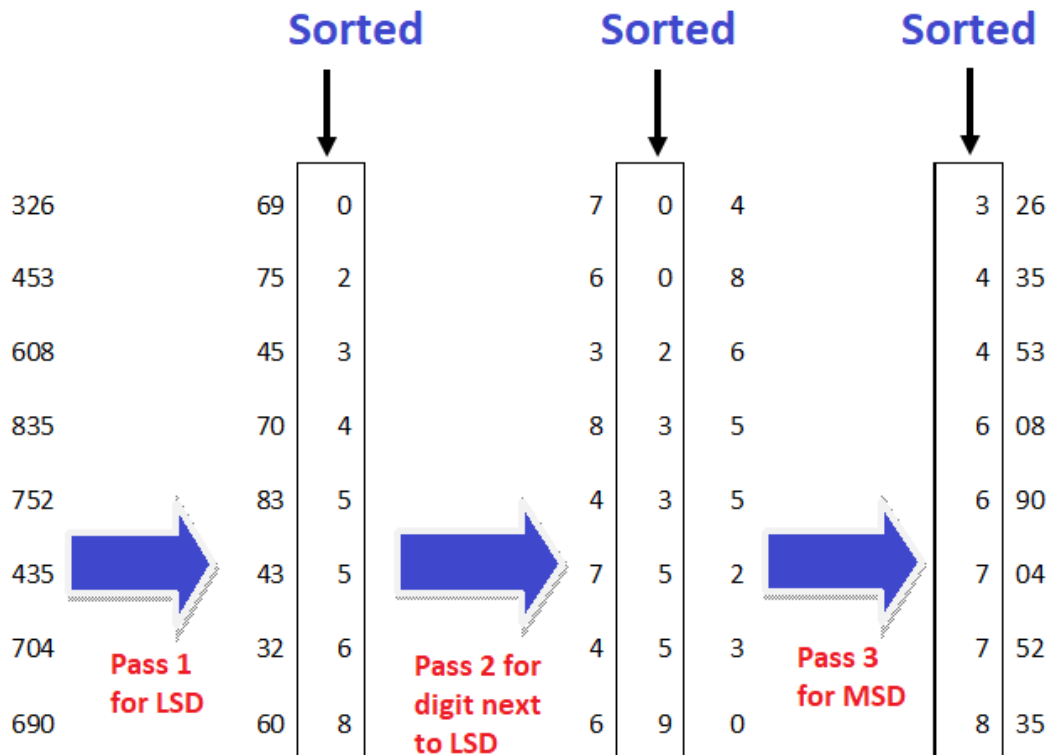
END;

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 i^{th} 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.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

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. **Radix Sort | CodeChef:** 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.

($1 \leq n \leq 106, 1 \leq k \leq m \leq 106, n * m \leq 5 * 10^7$)

Then the description of the strings follows. The format is not trivial. The i-string ($1 \leq i \leq n$) is written as the i^{th} symbols of the second, ..., $(m+1)^{\text{th}}$ 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 k 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-** In 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:

- Average Case complexity is $\theta(n)$
- Best case complexity is $\Omega(n)$
- Worst case time complexity is $O(n^2)$

2. **Space Complexity** of bucket sort is $\theta(n + k)$. n is number of elements and k is number of buckets.

Cases:

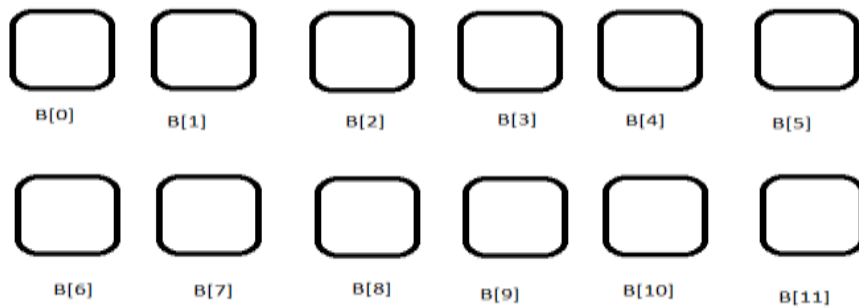
- Average case & Best Case:** when all element distribution is uniform in buckets.
- Worst Case:** when n or approximate n elements lie in one Bucket. Then it will work as insertion sort.

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)

$B[\text{int } n * a[i]] = a[i]$

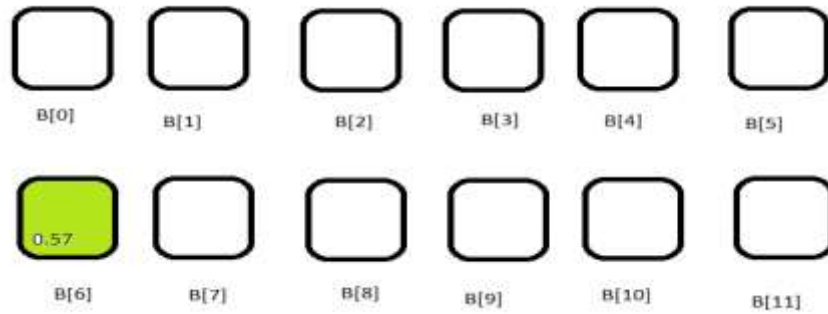


0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9
------	------	------	------	------	------	------	------	------	------	------	-----



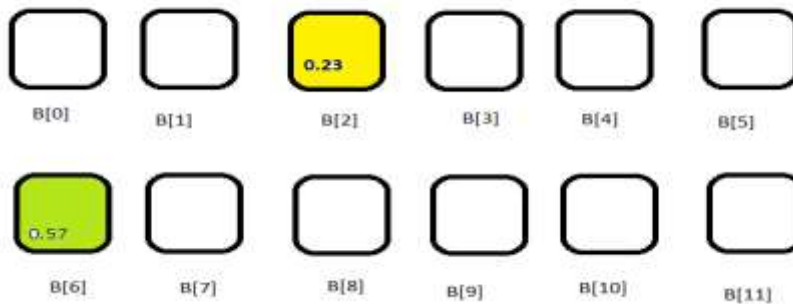
1st Element $B[\text{int } 12 * 0.57] \leftarrow .57$

$B[6] = 0.57$



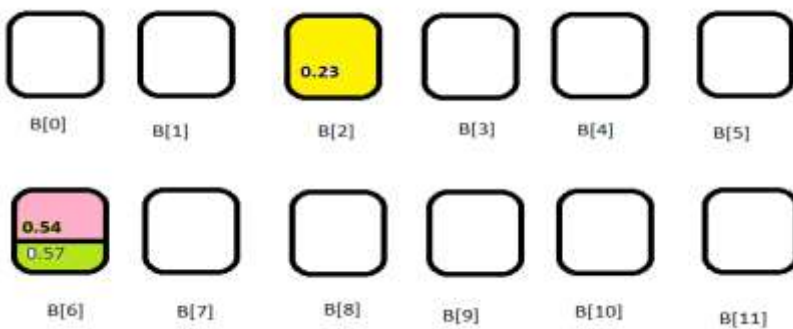
0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9
------	------	------	------	------	------	------	------	------	------	------	-----

↑
2nd Element $B[\text{int } 12 \cdot 0.23] \leftarrow 0.23$
 $B[2] \leftarrow 0.23$



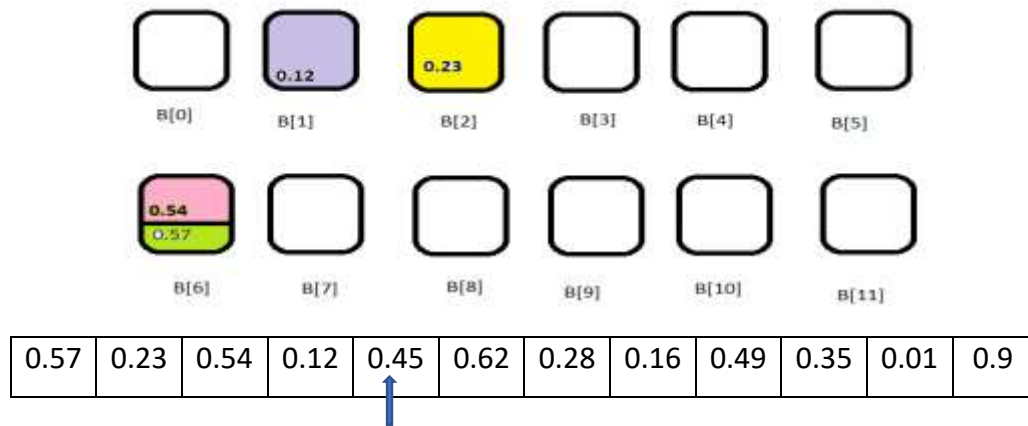
0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9
------	------	------	------	------	------	------	------	------	------	------	-----

↑
3rd Element $B[\text{int } 12 \cdot 0.54] \leftarrow 0.54$
 $B[6] \leftarrow 0.54$



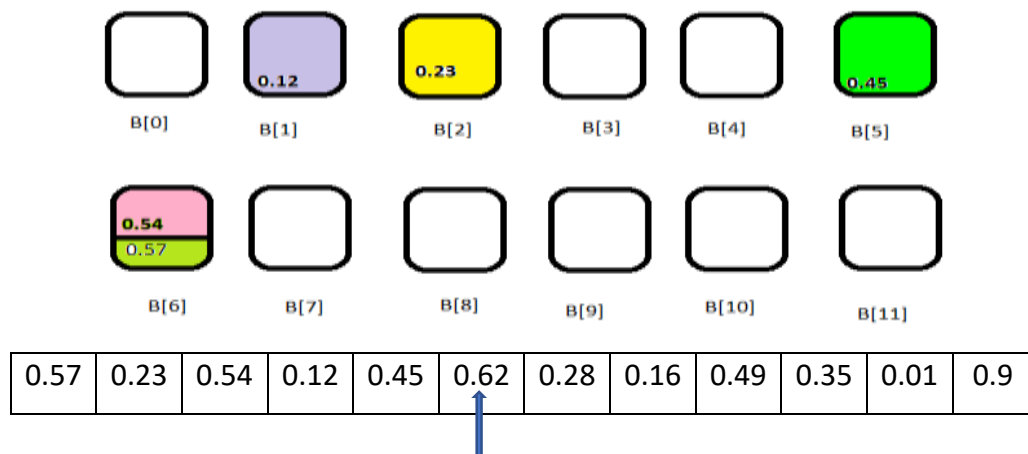
0.57	0.23	0.54	0.12	0.45	0.62	0.28	0.16	0.49	0.35	0.01	0.9
------	------	------	------	------	------	------	------	------	------	------	-----

↑
4th Element $B[\text{int } 12 \cdot 0.12] \leftarrow 0.12$
 $B[1] = 0.12$



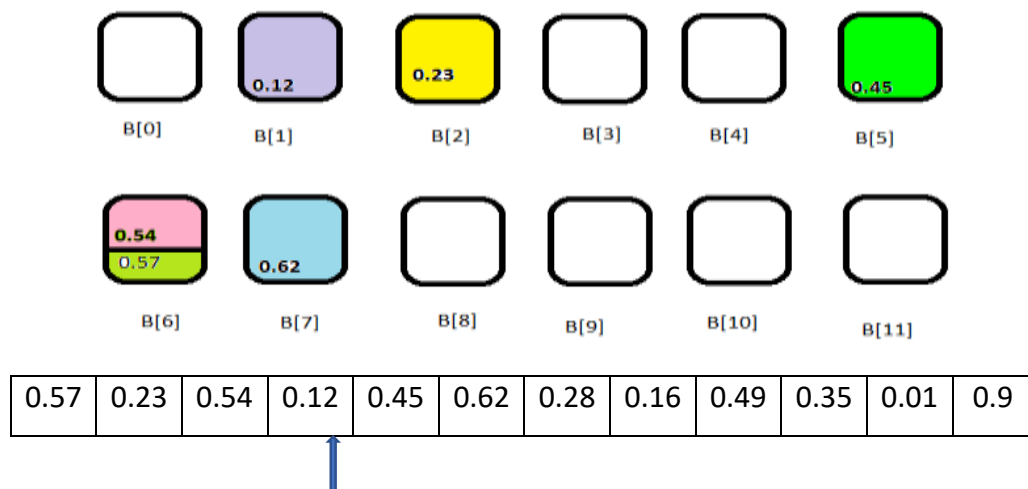
5th Element $B[\text{int } 12 * 0.45] \leftarrow 0.45$

$B[5] = 0.45$



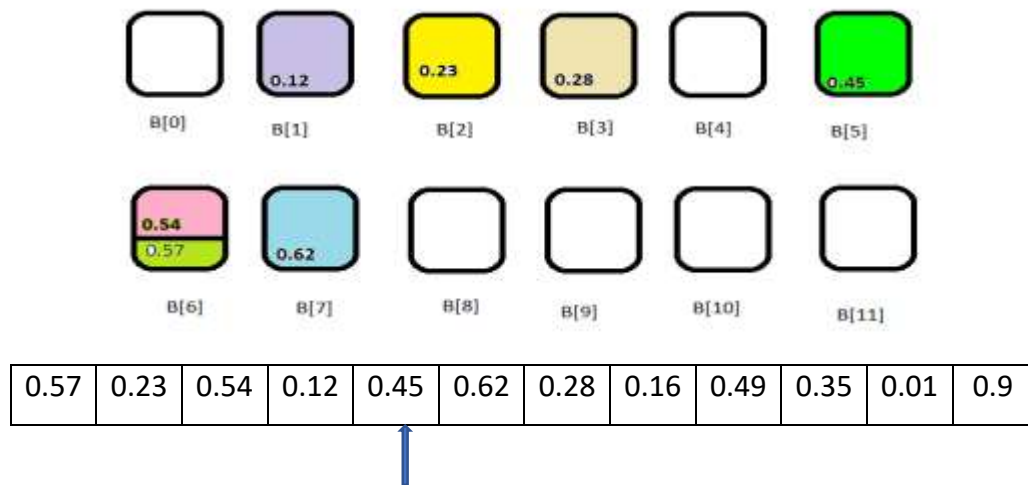
6th Element $B[\text{int } 12 * 0.62] \leftarrow 0.62$

$B[7] = 0.62$



7th Element $B[\text{int } 12 * 0.28] \leftarrow 0.28$

$B[3] = 0.28$



8th Element $B[\text{int } 12 \times 0.16] \leftarrow 0.16$

$B[1] = 0.16$



9th Element $B[\text{int } 12 \times 0.49] \leftarrow 0.49$

$B[5] = 0.49$



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 $B[\text{int } 12 \times 0.35] \leftarrow 0.35$

$B[4] = 0.35$



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 $B[\text{int } 12 \times 0.01] \leftarrow 0.01$

$B[0] = 0.01$

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.

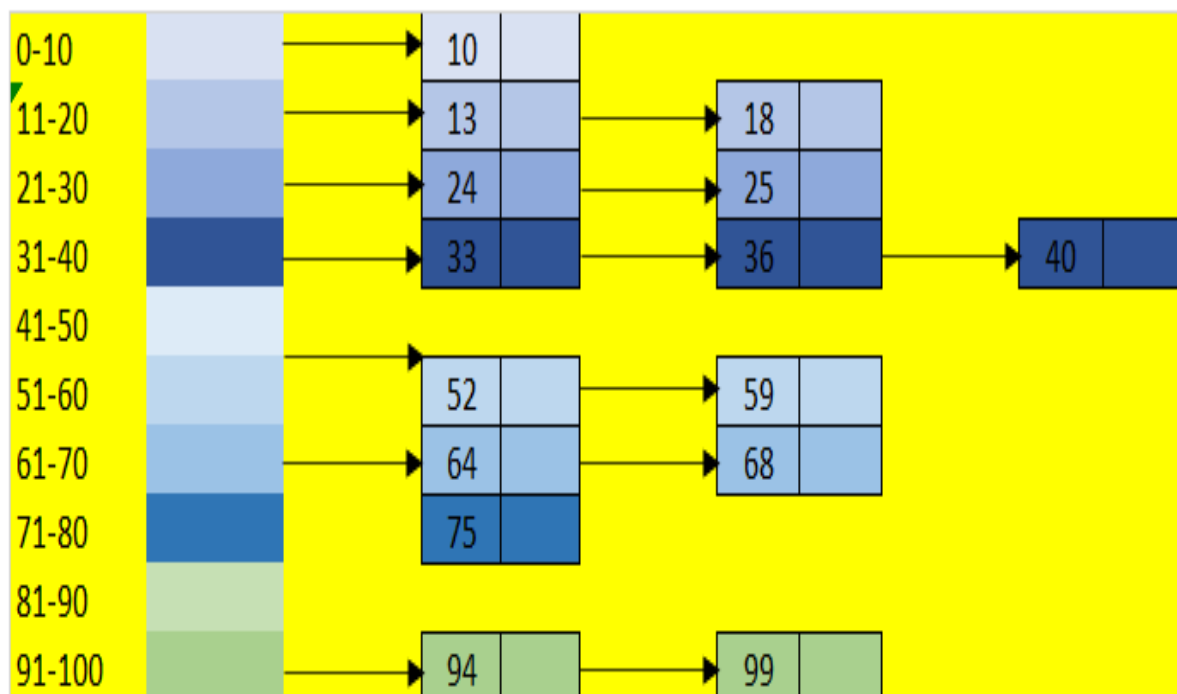
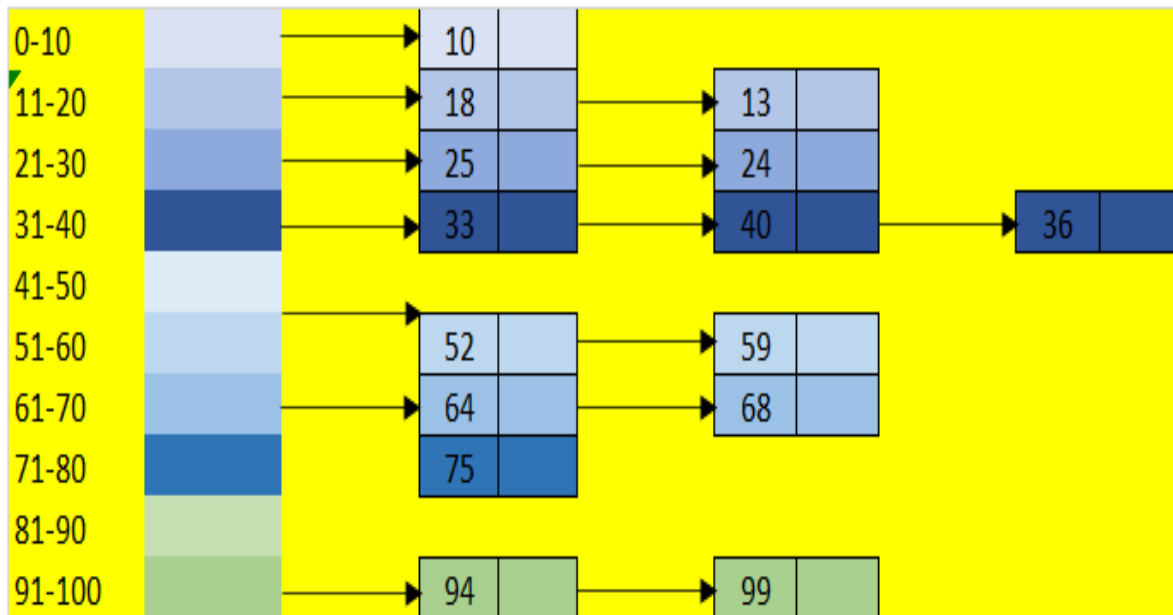
Algorithm 2–

- 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
$$\text{Range} = (\text{max} - \text{min}) / n \quad // n \text{ is the number of bucket}$$
3. Create n Buckets of Calculated Range
4. Distribute the elements in the buckets.
5. Bucket index = $(\text{arr}[i] - \text{min}) / \text{range}$
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 = $(10.5 - 0.5) / 5 = 2$

9.6	0.5	10.5	3.04	1.2	5.4	6.6	2.47	3.24	2.28
0.5	1.2	2.47	2.28	3.04	3.24	5.4	6.6	9.6	11
0.5	1.2	2.28	2.47	3.04	3.24	5.4	6.6	9.6	11
0.5	1.2	2.28	2.47	3.04	3.24	5.4	6.6	9.6	11

University Question:

Q1. What is recurrence relation? How is a recurrence solved using master's theorem? (5 marks)

Q2. What is asymptotic notation? Explain omega(Ω) notations. (5 marks)

Q3. Solve the recurrence **$T(n) = 4T(n/2) + n^2$** (2 marks)

Q4. Explain how algorithms performance is analyzed? (2 marks)

Q5. Write an algorithm for counting sort? Illustrate the operation of countingsort on the following array:

$A = \{2, 5, 3, 0, 2, 3, 0, 3\}$. (7 Marks)

Q6. Solve the following recurrence relation: (10 marks)

i. **$T(n) = T(n-1) + n^4$**

ii. $T(n) = T(n/4) + T(n/2) + n^2$

Q7. Write an algorithm for insertion sort. Find the time complexity of insertion sort in all cases. (5 marks)

Q8. Write Merge Sort Algorithm. And sort the following sequence {23,11,5,15,68,31,4, 17} using merge sort. (10 Marks)

Q9. What do you understand by stable and unstable sorting? Sort the following sequence {25, 57, 48, 36, 12, 91, 86, 32} by using heap sort. (10 marks)

Q10. Discuss the basic steps in the complete development of an algorithm. (2 marks)

Q11. Explain and compare best and worst time complexity of Quick Sort. (5 Marks)

Q12. Rank the following by growth rate: (2 Marks)

$n, 2^{\lg \sqrt{n}}, \log n, \log(\log n), \log^2 n, (\lg n)^{\lg n}, 4, (3/2)^n, n!$

Q13. (i) Solve the recurrence $T(n)=2T(n/2)+n^2+2n+1$

(ii) Prove that worst case running time of any comparison sort is $\Omega(n \log n)$ (10 marks)

Q14. (10 marks)

Among Merge sort, Insertion sort and quick sort which sorting technique is the best in worst case. Apply the best one among these algorithms to Sort the list E, X, A, M, P, L, E in alphabetic order.

Q15. (7 marks)

Solve the recurrence using recursion tree method:

$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

Q16. The recurrence $T(n) = 7T(n/2) + n^2$ describe the running time of an algorithm A. A competing algorithm A' has a running time of $T'(n) = aT'(n/4) + n^2$. What is the largest integer value for a' is asymptotically faster than A? (7 Marks)

Q17. Write an algorithm for bubble sort. Illustrate the operation of bubble sort on the following array {20, 14,12,15,25,30,6}. Also explain the condition when the time complexity of bubble sort will be $O(n)$. (write pseudo code) (10 marks)

Q18. What is advantage of binary search over linear search? Also state the limitation of binary search (5 marks)

Q19. Write an algorithm of shell sort. Also explain why this is called extension of insertion sort using an example. (5 Marks)

Q20. Explain growth of functions. Mention all the asymptotic notations. How running time and complexity of an algorithm are related to each other. Elucidate with the help of asymptotic notations (7 marks)

Q21. Explain and write bucket sort. Write complexity and draw step by step execution with appropriate data structure to illustrate the operation of BUCKETSORT on the array $A = \{.79, .13, .16, .64, .39, .20, .89, .53, .71, .42\}$. (10 marks)

- Q22. Explain and write radix sort. Write complexity and draw step by step execution with appropriate data structure to illustrate the operation of RADIXSORT on the list of English words {COW,DOG,SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR,DIG, BIG, TEA, NOW, FOX}. (10 marks)
- Q23. Find Θ notation for the function $f(n)=27n^2+16n+25$. (2 marks)
- Q24. Let $f(n)$ and $g(n)$ be asymptotic positive functions. Prove or disprove the following conjectures:
- $f(n)=O(g(n))$ implies $g(n)=O(f(n))$
 - $f(n)+g(n)=\Theta(\min(f(n),g(n)))$ (5 marks)
- Q25. Write Master's method for solving recurrence relations of different type.(2 marks)
- Q26. Prove that building of MAX HEAP takes linear time. (5 marks)
- Q27. What is the effect of calling Max-Heapify (A,i) when the element $A[i]$ is larger than its children?(5 marks)
- Q28. Which of the following sorting algorithm(s) are stable: insertion sort, merge sort, heap sort, and quick sort? Argue that any comparison based sorting algorithm can be made stable without effecting the running time by more than a constant factor.(10 marks)
- Q29. Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor \text{length}[A]/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor \text{length}[A]/2 \rfloor$?(10 marks)
- Q30. Suppose that the splits at every level of quicksort are in the proportion $1 - \alpha$ to α , where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1 - \alpha)$. (Don'tworry about integer round-off.)(10 marks)
- Q31. Suppose that the for loop header in line 9 of the COUNTING-SORT procedure is rewritten as for $j \leftarrow 1$ to $\text{length}[A]$.Show that the algorithm still works properly. Is the modified algorithm stable? (10 marks)