Unit-1

1 .What is an algorithm? What do you mean by correctalgorithm? What do you mean by instance of a problem? Listout the criteria that all algorithms must satisfy. On what baseswill you consider algorithm A is better than algorithm B?

ANS:-

An algorithm is a procedure used for solving a problem or performinga computation. Algorithms act as an exact list of instructions that conduct specified actions step by step in either hardware- or software-based routines.

Algorithms are widely used throughout all areas of IT. In mathematics and computer science, an algorithm usually refers to a small procedure that solves a recurrent problem. Algorithms are also used as specifi-cations for performing data processing and play a major role in automated systems.

An algorithm is correct only if it produces correct result for all input instances. – If the algorithm gives an incorrect answer for one or more input instances, it is an incorrect algorithm.

A specific selection of values for the parameters is called an instance of the problem. For example, the input parameter to a sorting function might be an array of integers. A particular array of integers, with a given size and specific values for each position in the array, would be an instance of the sorting problem. Different instances might generate the same output. However, any problem instance must always result in the same output every time the function is computed using that particular input.

All algorithms must satisfy the following criteria:

Zero or more input values3

One or more output values

Clear and unambiguous instructions

Atomic steps that take constant time

No infinite sequence of steps (help, the halting problem)

Feasible with specified computational device

In asymptotic analysis, we consider growth of algorithm in terms of input size.

An algorithm A is said to be asymptotically better than B if A takes smaller time than B for all input sizes n larger than a value n0 where n0 > 0.

Q.2. What is an Algorithm? What do you mean by linear inequalities and linear equations? Explain asymptotic notation with the help of example.

ANS:-

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

An algorithm to add two numbers:

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

Two algebraic expressions or real numbers related by the symbol ≤, ≥, < and > form an inequality. If the expressions are linear then the inequality is called linear inequality. Eg : 4(x – 2) < 10, 10 < 45, x >= -3, etc.

Symbols of linear inequality:

- Not Equal to (≠)
- Greater Than (>)
- Less Than (<)
- Greater Than or Equal To (≥)
- Less Than or Equal To (≤)

The Greater than (>) and Less than (<) are called **Strict Inequalities** as they depict that a number is strictly less or more than the other.
The Greater Than or Equal To (≥) and Less Than or Equal To (≤) are called **Slack Inequalities** or non-strict inequalities as they depict that the value is included in the solution.

Properties of Linear Inequality

- We can add or subtract a real number from both sides of the inequality
- We can multiply or divide both the sides of the inequality by a positive number
- If we multiply or divide both the sides of the inequality by a negative number then the inequality sign gets reversed

Types of Linear Inequality

- Linear Inequality in one variable: If the linear function involves a single variable then it is linear inequality in one variable. Eg: 2x – 45 > 7
- Linear inequality in two variables: If the linear function involves two variables then it is linear inequality in two variables. Eg : 4x + 7y > 9

i. Asymptotic analysis is used to evaluate the performance of an algorithm in terms of input size.

ii. The basic idea of asymptotic analysis is to measure the efficiency of algorithms that doesn't depend on machine specific constants.

iii. The mathematical tools to represent time complexity of algorithms for asymptotic analysis are called as asymptotic notations.

2. Description:

i. There are 3 notations to measure time complexity of a program
namely Big−O,Big−ΩandBig−θBig−O,Big−ΩandBig−θ.
ii. Big-O

- This notation defines an upper bound of an algorithm.
- The function f(n)=O(g(n)) if and only if fn<=c.g(n) for all n>=no where c and no are constants.
- Here, g(n) is known as upper bound on values of f(n).
- E.g.f(n)=3n+3,g(n)=4nE.g.f(n)=3n+3,g(n)=4n.

iii. Big-Ω

- Ω notation provides an asymptotic lower bound.
- The function f(n)= Ω g(n) if f(n)>=c.g(n) for all n>=no where c and no are constants.
- Here, g(n) is known as lower bound on values of f(n).
- E.g.f(n)=3n+2andg(n)=3nE.g.f(n)=3n+2andg(n)=3n.

iv. Big-θ

- The theta notation bounds a function from above and below, so it defines exact asymptotic behaviour. Hence, it is also known as tightly bound.
- The function f(n)= θ (g(n)) if c1.g(n) <= f(n) <= c2.g(n) for all n>=no where c1, c2 and no are constants.
- E.g.f(n)=3n+2,g(n)=n,C1=3andC2=4

Q.3. Why do we use asymptotic notations in the study of Algorithms? Briefly describe the commonly used asymptotic Notations.

ANS:-

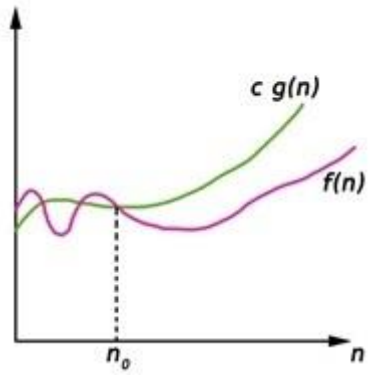
Asymptotic Notations, also known as 'algorithm's growth rate' are languages which allow analyzing the running time of an algorithm. This is calculated by identifying its behavior as the input size for the algorithm changes.

Asymptotic analysis of an algorithm refers to defining the mathematical boundation or the framing of its run-time performance.

By using asymptotic analysis, the best case, average case, and worst case scenario of an algorithm can be concluded.

Big Oh Notation

Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor.

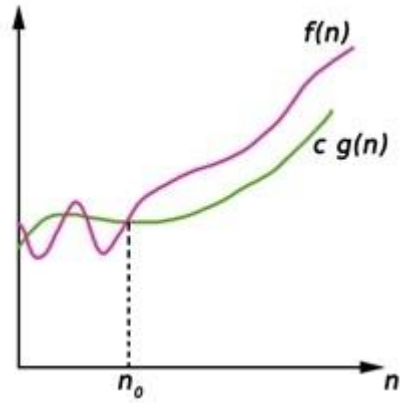


We write f(n) = O(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ f(n) ≤ c g(n), for all n ≥ n0}

Big Omega Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

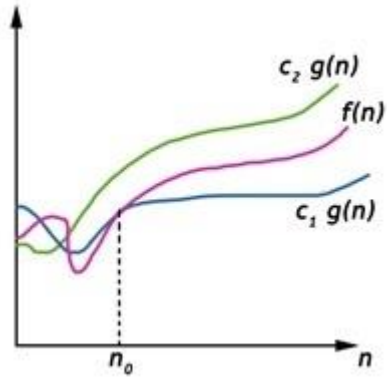


We write f(n) = Ω(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ c g(n) ≤ f(n), for all n ≥ n0}

Big Theta Notation

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.



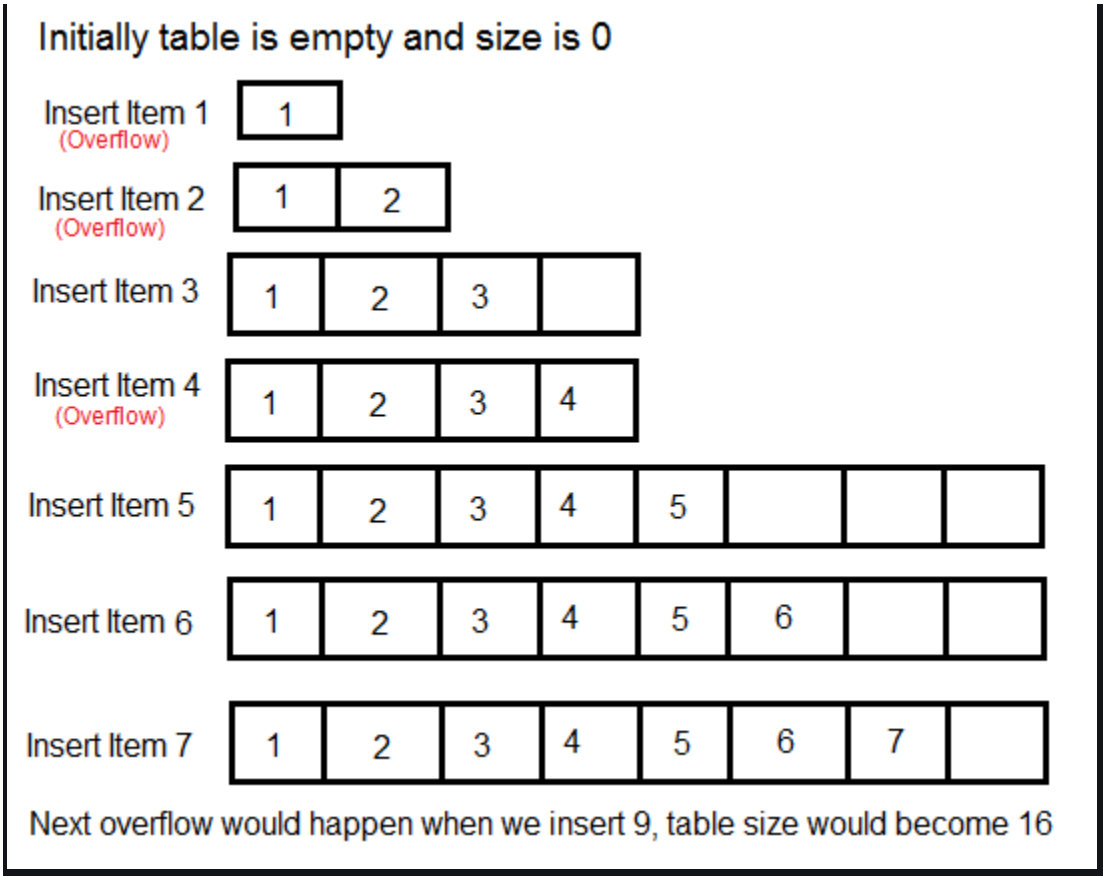
We write f(n) = Θ(g(n)), If there are positive constantsn0 and c1 and c2 such that, to the right of n0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n), for all n ≥ n0}

Q.5. What is an amortized analysis? Explain accounting method and aggregate analysis with suitable example.

ANS:-

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets, and Splay Trees.
Let us consider an example of simple hash table insertions. How do we decide on table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but the space required becomes high.



The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase the size of the table whenever it becomes full. Following are the steps to follow when the table becomes full.
1) Allocate memory for larger table size, typically twice the old table.
2) Copy the contents of the old table to a new table.
3) Free the old table.
If the table has space available, we simply insert a new item in the available space.

What is the time complexity of n insertions using the above scheme?
If we use simple analysis, the worst-case cost of insertion is O(n). Therefore, the worst-case cost of n inserts is n * O(n) which is $O(n^2)$. This analysis gives an upper bound, but not a tight upper bound for n insertions as all insertions don't take Θ(n) time.

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1...)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1...)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + ...)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

So using Amortized Analysis, we could prove that the Dynamic Table scheme has O(1) insertion time which is a great result used in hashing. Also, the concept of the dynamic table is used in vectors in C++ and ArrayList in Java.

Q.6. Explain following terms with example. 1. Set 2. Relation 3. Function

ANS:-

Set - Definition

A set is an unordered collection of different elements. A set can be written explicitly by listing its elements using a set bracket. If the order of the elements is changed or any element of a set is repeated, it does not make any changes in the set.

Some Example of Sets

- A set of all positive integers
- A set of all the planets in the solar system
- A set of all the states in India
- A set of all the lowercase letters of the alphabet

Representation of a Set

Sets can be represented in two ways −

- Roster or Tabular Form
- Set Builder Notation

Roster or Tabular Form

The set is represented by listing all the elements comprising it. The elements are enclosed within braces and separated by commas.

Example 1 − Set of vowels in English alphabet, A = { a,e,i,o,u }

Example 2 − Set of odd numbers less than 10, B = { 1,3,5,7,9 }

Some Important Sets

N − the set of all-natural numbers = {1, 2, 3, 4, ...}

Z − the set of all integers = {....., -3, -2, -1, 0, 1, 2, 3, ...}

$\mathbf{Z^+}$ − the set of all positive integers

Q − the set of all rational numbers

R − the set of all real numbers

W − the set of all whole numbers

Relation

A relation from set A to set B is a subset of the cartesian product set A x B. The subset is made up by describing a relationship between the first element and the second element of elements in A x B.

Example: R = {(1,2), (2, -3), (3,5)}

Here in the above example, set of all first elements i.e {1,2,5} is called Domain while the set of all second elements i.e {2,-3,5} is called the range of the relation.

Types of Relation

There are 8 main types of relations which include:

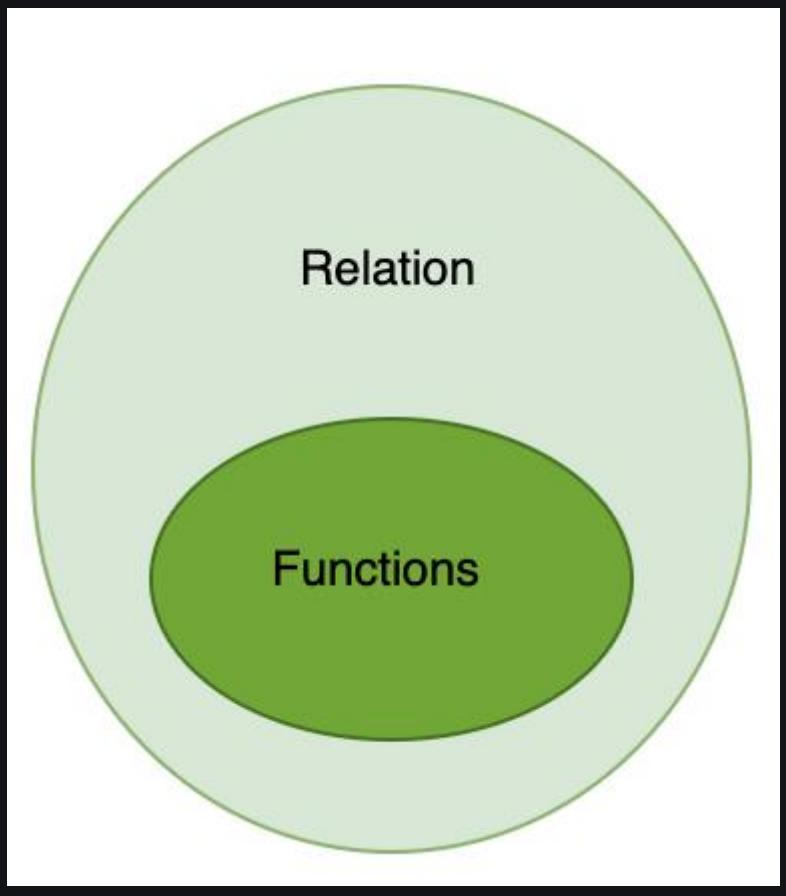
1. **Empty Relation-** There is no relation between any elements of a set.
2. **Universal Relation-** Every element of the set is related to each other.
3. **Identity Relation-** In an identity relation, every element of a set is related to itself only.
4. **Inverse Relation-** Inverse relation is seen when a set has elements that are inverse pairs of another set.

5. **Reflexive Relation-** In a reflexive relation, every element maps to itself.
6. **Symmetric Relation-** In asymmetric relation, if a=b is true then b=a is also true.
7. **Transitive Relation-** For transitive relation, if (x, y) ∈ R, (y, z) ∈ R, then (x, z) ∈ R.
8. **Equivalence Relation-** A relation that is symmetric, transitive, and reflexive at the same time.

Function

A function is a special kind of relation. It is a relation in which each domain value maps only to one range value. It is denoted by **ƒ:X⇒Y**

What this means that it is a function from X to Y. It takes input from set X and gives the unique value from set Y as output. "X" is called the domain of the function while "Y" is called the co-domain.



All functions are relations but all relations are not functions.

As described earlier in the introduction, it can be thought of as a block/machine which runs on some formula or rule. It takes input and spits out the output.

Q.8. Define an amortized analysis. Briefly explain its different Techniques. Carry out aggregate analysis for the problem of implementing a k-bit binary counter that counts upward from 0.

ANS:-

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time that is lower than the worst-case time of a particularly expensive operation.
The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets, and Splay Trees.
Let us consider an example of simple hash table insertions. How do we decide on table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but the space required becomes high.

Q.9. Define following terms (i) Quantifier (ii) Algorithm (iii) Big 'Oh' Notation (iv) Big 'Omega' Notation (v) 'Theta' Notation.

ANS:-

Quantifiers

Quantifier is used to quantify the variable of predicates. It contains a formula, which is a type of statement whose truth value may depend on values of some variables. When we assign a fixed value to a predicate, then it becomes a proposition. In another way, we can say that if we quantify the predicate, then the predicate will become a proposition. So quantify is a type of word which refers to quantifies like **"all"** or **"some"**.

There are mainly two types of quantifiers that are universal quantifiers and existential quantifiers. Besides this, we also have other types of quantifiers such as nested quantifiers and Quantifiers in Standard English Usages. Quantifier is mainly used to show that for how many elements, a described predicate is true. It also shows that for all possible values or for some value(s) in the universe of discourse, the predicate is true or not.

Example 1:

"x ≤ 5 ∧ x > 3"

This statement is false for x= 6 and true for x = 4.

Algorithm

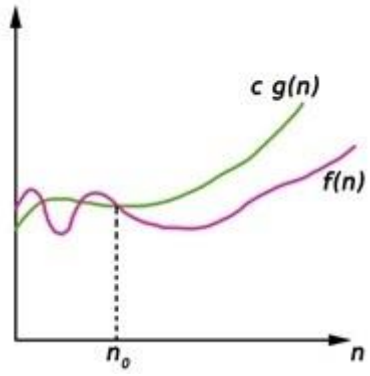
In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output. For example,

An algorithm to add two numbers:

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

Big Oh Notation

Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor.

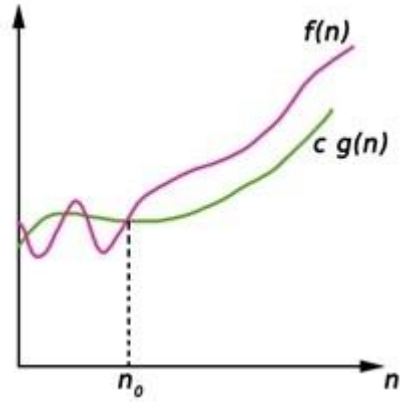


We write f(n) = O(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ f(n) ≤ c g(n), for all n ≥ n0}

Big Omega Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

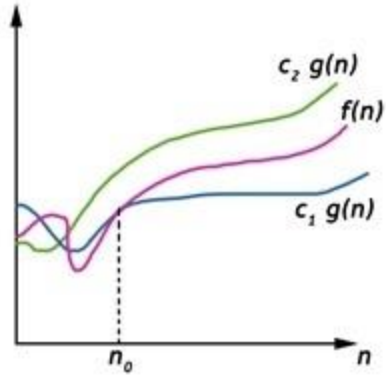


We write f(n) = Ω(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ c g(n) ≤ f(n), for all n ≥ n0}

Big Theta Notation

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.



We write f(n) = Θ(g(n)), If there are positive constantsn0 and c1 and c2 such that, to the right of n0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n), for all n ≥ n0}

Q.10. SHORT QUESTIONS (i) what is an Algorithm? (ii) what is worst case time complexity? (iii) Big Oh notation.

ANS:-

(i)what is algorithm?

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

An algorithm to add two numbers:

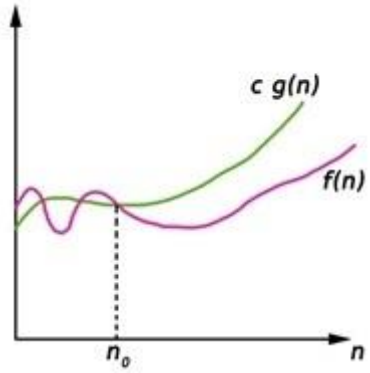
1. Take two number inputs
2. Add numbers using the + operator
3. Display the result

(ii) what is worst case time complexity?

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be **O(n)**.

(iii)Big Oh notation.

Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor.



We write f(n) = O(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n0 such that $0 \leq f(n) \leq c\ g(n)$, for all $n \geq n0$}

Q.11. Define Algorithm, Time Complexity and Space Complexity.

ANS:-

Analysis of algorithms

Algorithm analysis is an important part of computational complexities. The complexity theory provides the theoretical estimates for the resources needed by an algorithm to solve any computational task. Analysis of the algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of the analysis of the algorithm is the required time or performance.

Complexities of an Algorithm

The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n). The complexity of an algorithm can be divided into two types. The **time complexity** and the **space complexity**.

Time Complexity of an Algorithm

The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm. This calculation is totally independent of implementation and programming language.

Space Complexity of an Algorithm

Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm. The memory space is generally considered as the primary memory.

Unit-2

Q.1. What is an algorithm? Explain various properties of an algorithm.

ANS:-

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

Q.2. What do you mean by asymptotic notations? Explain.

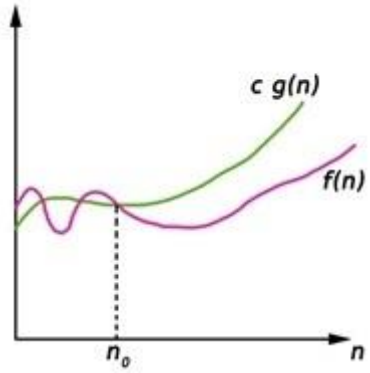
ANS:-

Asymptotic Notations

Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

Big Oh Notation

Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor.

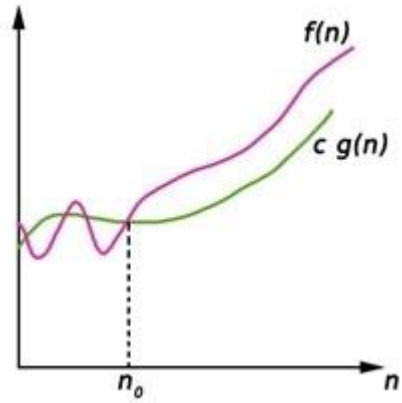


We write f(n) = O(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ f(n) ≤ c g(n), for all n ≥ n0}

Big Omega Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

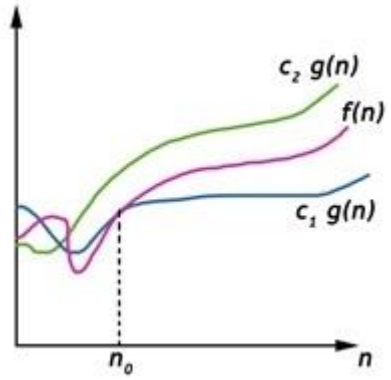


We write f(n) = Ω(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ c g(n) ≤ f(n), for all n ≥ n0}

Big Theta Notation

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.



We write f(n) = Θ(g(n)), If there are positive constantsn0 and c1 and c2 such that, to the right of n0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n), for all n ≥ n0}

Q.3. Write a program/algorithm of selection sort methods. What is complexity of the method?

ANS:-

```
void SelectionSort(int a[])
{
    int i, max;
    for(i=0;i<n;i++)
    {
        max=FindMax(a,n-i-1);
        swap(a,max,n-i-1);
    }
}

int FindMax(int a[], int high)
{
   int i, index;
   index=high;
   for(i=0;i<high;i++)
   {
     if(a[i]>a[index])
        index=i;
   }
   return index;
```

}

</high;i++)
</n;i++)

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

To assess the complexity, the order (approximation) of the count of operation is always considered instead of counting the exact steps.

The complexity can be found in any form such as constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. It is nothing but the order of constant, logarithmic, linear and so on, the number of steps encountered for the completion of a particular algorithm. To make it even more precise, we often call the complexity of an algorithm as "running time".

Q.4. Explain different asymptotic notations in brief.

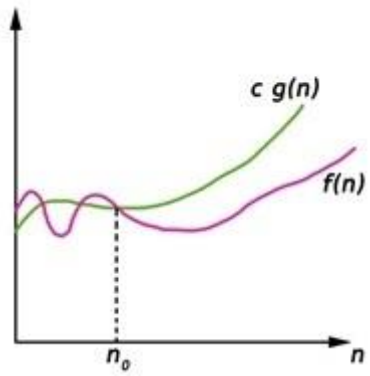
ANS:-

Asymptotic Notations

Asymptotic notations are used to represent the complexities of algorithms for asymptotic analysis. These notations are mathematical tools to represent the complexities. There are three notations that are commonly used.

Big Oh Notation

Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor.

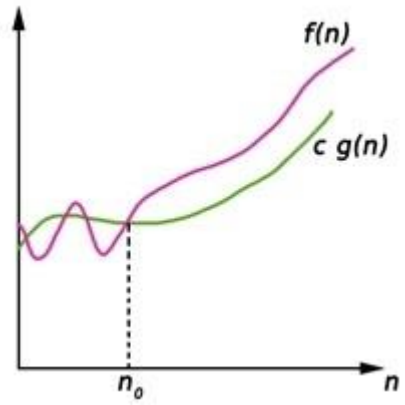


We write f(n) = O(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ f(n) ≤ c g(n), for all n ≥ n0}

Big Omega Notation

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

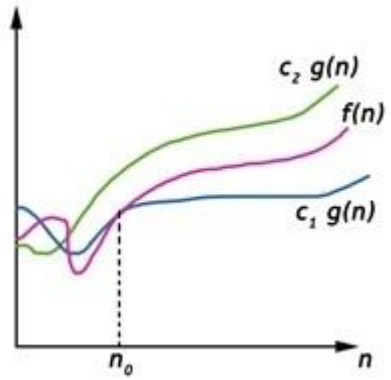


We write f(n) = Ω(g(n)), If there are positive constantsn0 and c such that, to the right of n0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n0 such that 0 ≤ c g(n) ≤ f(n), for all n ≥ n0}

Big Theta Notation

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.



We write f(n) = Θ(g(n)), If there are positive constantsn0 and c1 and c2 such that, to the right of n0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n0 such that 0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n), for all n ≥ n0}

Q.5. What is an amortized analysis? Explain aggregate method of amortized analysis using simple example.

ANS:-

Amortize Analysis

This analysis is used when the occasional operation is very slow, but most of the operations which are executing very frequently are faster. Data structures we need amortized analysis for Hash Tables, Disjoint Sets etc.

In the Hash-table, the most of the time the searching time complexity is O(1), but sometimes it executes O(n) operations. When we want to search or insert an element in a hash table for most of the cases it is constant time taking the task, but when a collision occurs, it needs O(n) times operations for collision resolution.

Aggregate Method

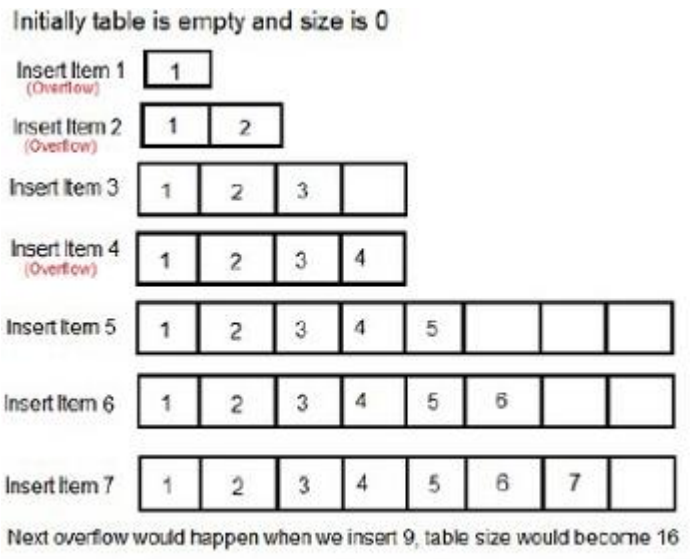
The aggregate method is used to find the total cost. If we want to add a bunch of data, then we need to find the amortized cost by this formula.

For a sequence of n operations, the cost is −

$$\frac{Cost(n\ operations)}{n} = \frac{Cost(normal\ operations) + Cost(Expensive\ operations)}{n}$$

Example on Amortized Analysis

For a dynamic array, items can be inserted at a given index in O(1) time. But if that index is not present in the array, it fails to perform the task in constant time. For that case, it initially doubles the size of the array then inserts the element if the index is present.



For the dynamic array, let = cost of *ith* insertion.

$$\text{So } ci = 1 + \begin{cases} i-1, \text{ if } i-1 \text{ is power of } 2 \\ 0, \qquad\qquad \text{Otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} \leq \frac{n + \sum_{j=1}^{\lfloor log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n}$$

Q.6. Explain why analysis of algorithm is important? Explain: Worst case, Best case, Average case complexity.

ANS:-

Algorithm analysis is important in practice because the accidental or unintentional use of an inefficient algorithm can significantly impact system performance. In time-sensitive applications, an algorithm taking too long to run can render its results outdated or useless. An inefficient algorithm can also end up requiring an uneconomical amount of computing power or storage in order to run, again rendering it practically useless.

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be

executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be O(n).

2. Best Case Analysis (Very Rarely used)

In the best case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Ω(1)

3. Average Case Analysis (Rarely used)

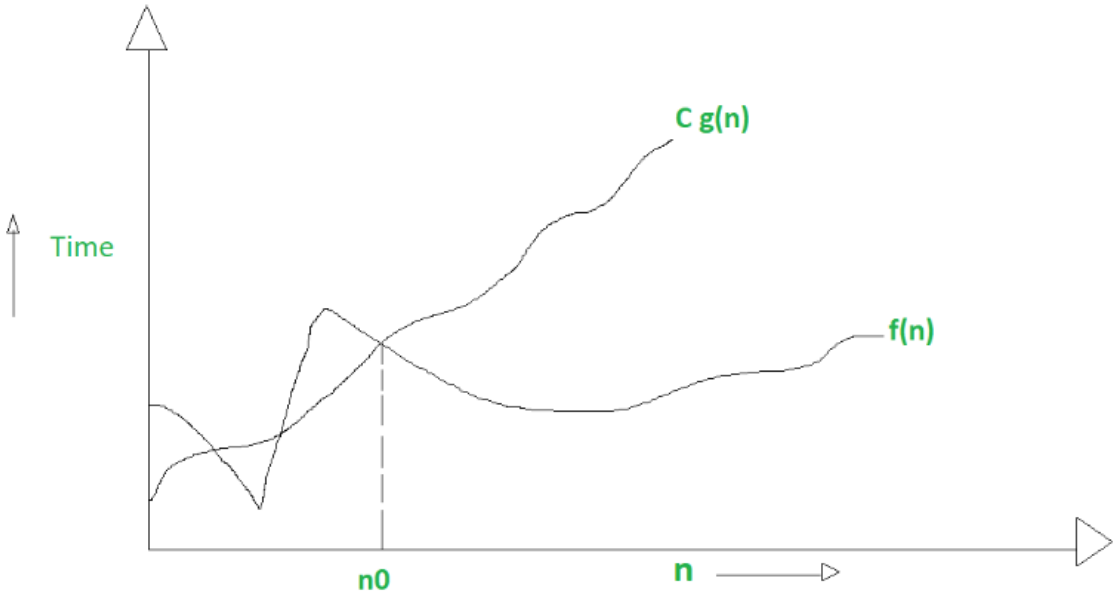
In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Q.7. Define: Big Oh, Omega and Big Theta notation.

ANS:-

1. Big oh notation (O):
It is define as upper bound and upper bound on an algorithm is the most amount of time required (the worst case performance).
Big oh notation is used to describe **asymptotic upper bound**.
Mathematically, if **f(n)** describes the running time of an algorithm; **f(n)** is **O(g(n))** if there exist positive constant **C** and **n0** such that,

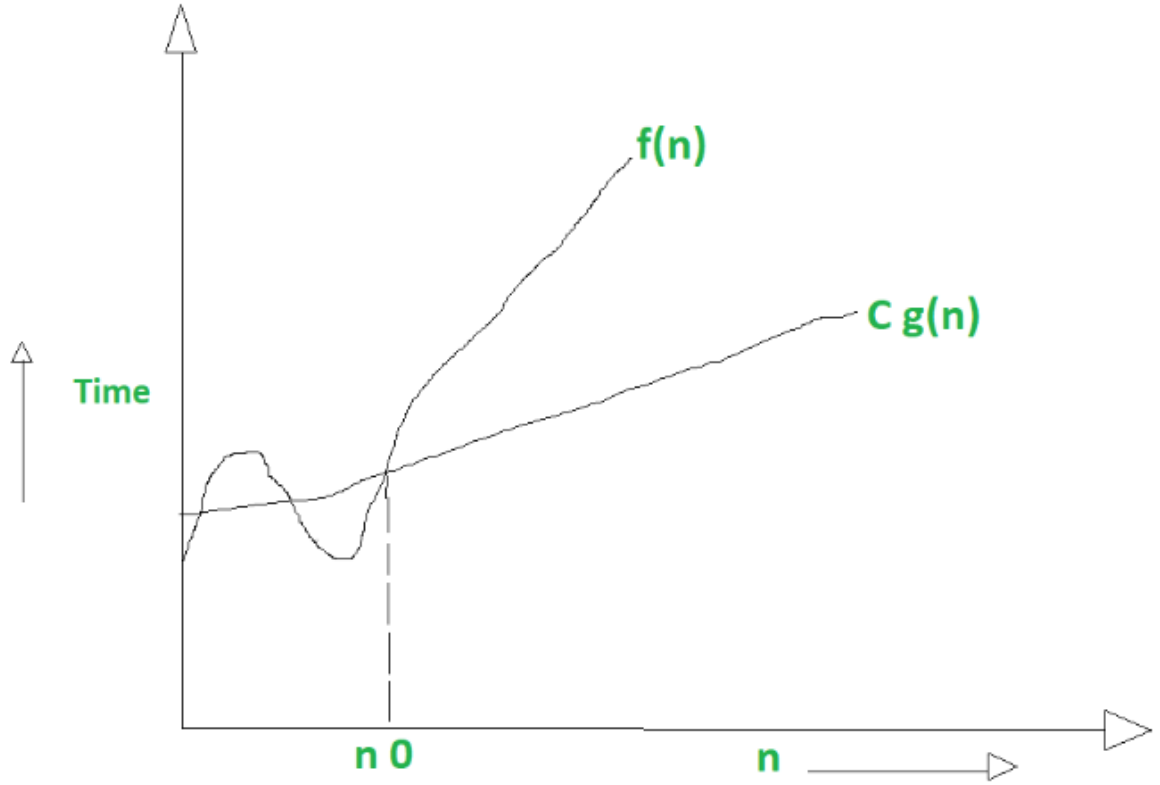
n = used to give upper bound an a function.
If a function is **O(n)**, it is automatically **O(n-square)** as well.
Graphic example for **Big oh (O) :**



2. Big Omega notation (Ω) :
It is define as lower bound and lower bound on an algorithm is the least amount of time required (the most efficient way possible, in other words best case).
Just like **O notation** provide an **asymptotic upper bound**, **Ω notation** provides **asymptotic lower bound**.

Let **f(n)** define running time of an algorithm;
f(n) is said to be **Ω(g (n))** if there exists positive constant **C** and **(n0)** such that

n = used to given lower bound on a function
If a function is **Ω(n-square)** it is automatically **Ω(n)** as well.
Graphical example for **Big Omega (Ω):**

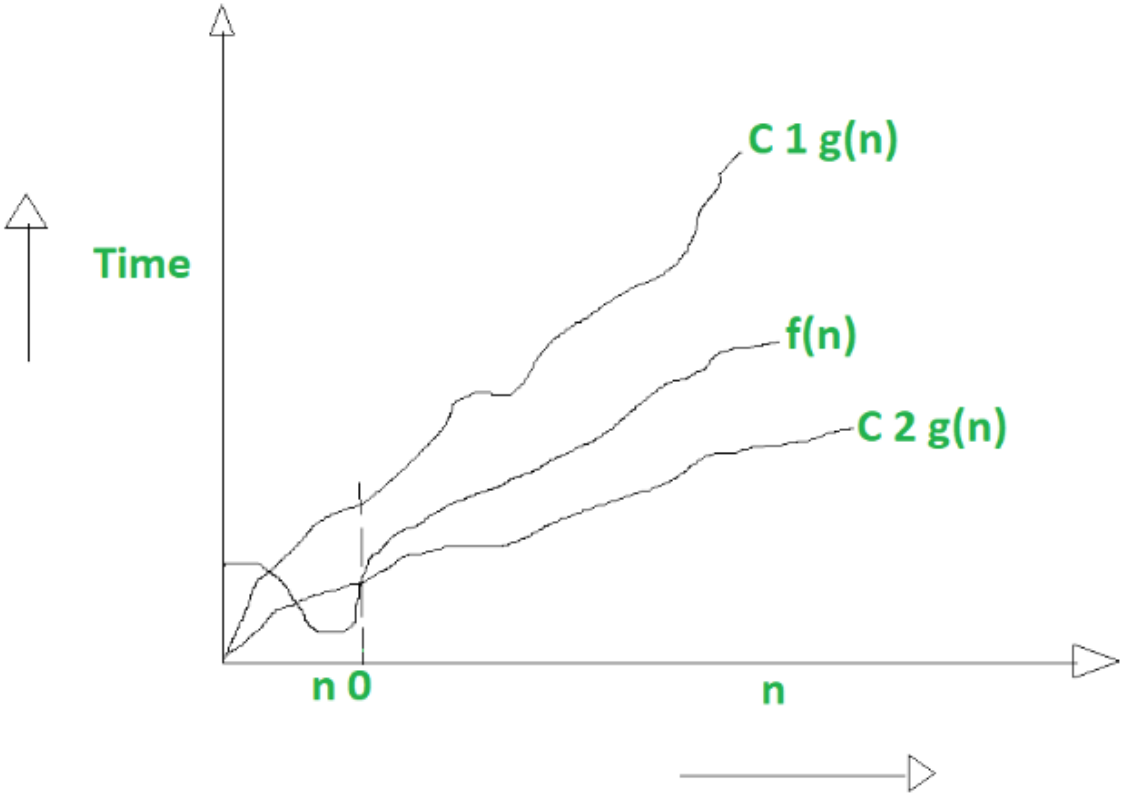


3. Big Theta notation (Θ) :
It is define as tightest bound and tightest bound is the best of all the worst case times that the algorithm can take.

Let **f(n)** define running time of an algorithm.
f(n) is said to be **Θ(g(n))** if **f(n)** is **O(g(n))** and **f(n)** is **Ω(g(n)).**

The equation simply means there exist positive constants C1 and C2 such that f(n) is sandwich between C2 g(n) and C1g(n).

Graphic example of **Big Theta (Θ)**:



Q.8. What is Recursion? Give the implementation of Tower of Hanoi Problem using recursion.

ANS:-

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

Tower of Hanoi is a mathematical puzzle where we have three rods (**A**, **B**, and **C**) and **N** disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod **A**. The objective of the puzzle is to move the entire stack to another rod (here considered **C**), obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Tower of Hanoi using Recursion:

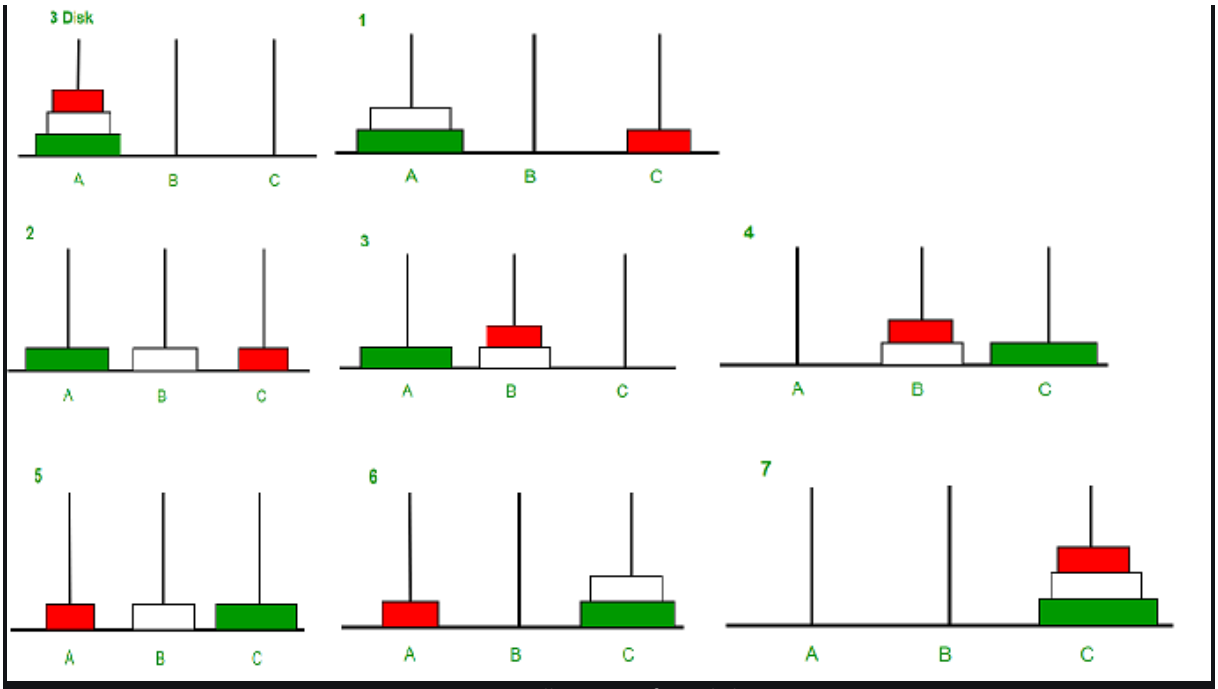


Image illustration for 3 disks

Follow the steps below to solve the problem:

- Create a function **towerOfHanoi** where pass the **N** (current number of disk), **from_rod**, **to_rod**, **aux_rod.**
- Make a function call for N – 1 th disk.
- Then print the current the disk along with **from_rod** and **to_rod**
- Again make a function call for N – 1 th disk.

Q.9. Explain why analysis of algorithm is important?

ANS:-

Why Analysis of Algorithms is important?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.

- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Algorithm Analysis:

1. Best case
2. Worst case
3. Average case

- **Best case:** Define the input for which algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
- **Worst Case**: Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is not present at all then the worst case occurs.
- **Average case**: In the average case take all random inputs and calculate the computation time for all inputs.
  And then we divide it by the total number of inputs.

Average case = all random case time / total no of case

Q.10. Explain bubble sort algorithm. Derive the algorithmic complexity in best case, worst case and average case analysis.

ANS:-

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

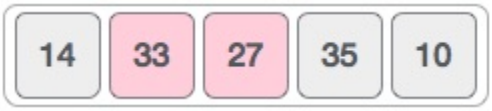
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

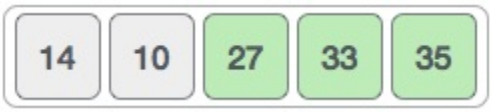
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

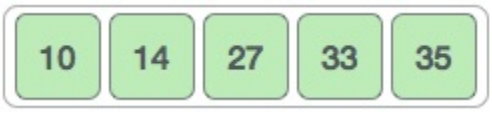
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list

end BubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows –

```
procedure bubbleSort( list : array of items )

  loop = list.count;

  for i = 0 to loop-1 do:
    swapped = false
```

```
    for j = 0 to loop-1 do:

      /* compare the adjacent elements */
      if list[j] > list[j+1] then
        /* swap them */
        swap( list[j], list[j+1] )
        swapped = true
      end if

    end for

    /*if no number was swapped that means
    array is sorted now, break the loop.*/

    if(not swapped) then
      break
    end if

  end for

end procedure return list
```

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search( ) function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be **O(n)**.

2. Best Case Analysis (Very Rarely used)

In the best case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Ω(1)

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Q.12. Explain the heap sort in detail. Give its complexity.

ANS:-

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

Working of Heap sort Algorithm

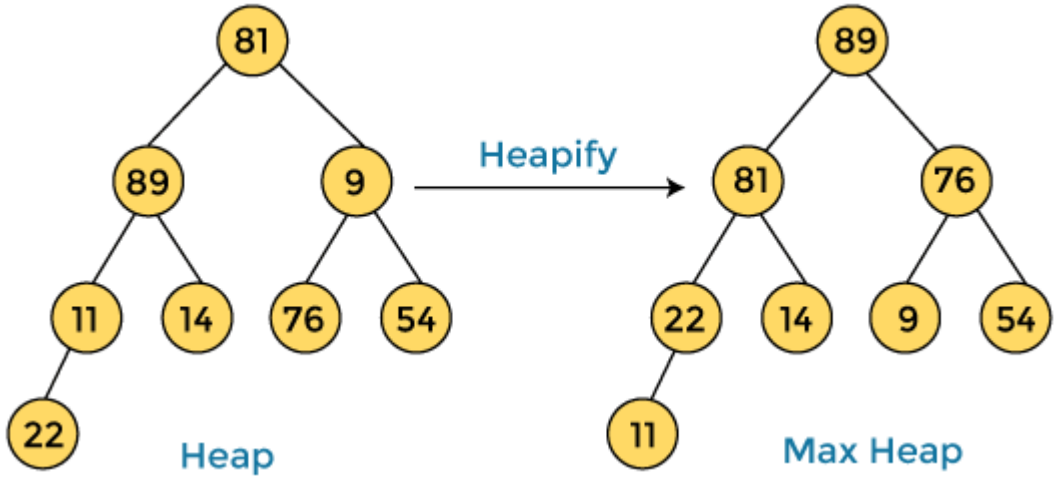
Now, let's see the working of the Heapsort Algorithm.

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

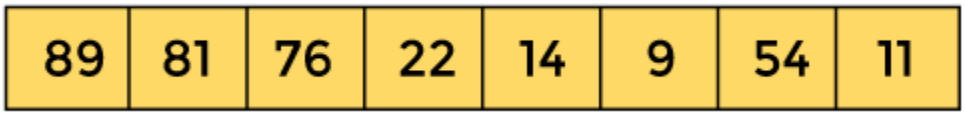
- The first step includes the creation of a heap by adjusting the elements of the array.
- After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

Now let's see the working of heap sort in detail by using an example. To understand it more clearly, let's take an unsorted array and try to sort it using heap sort. It will make the explanation clearer and easier.

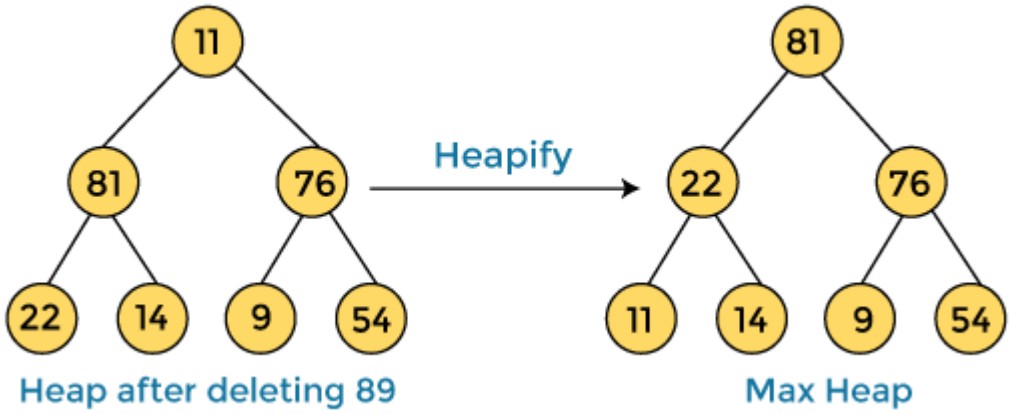
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

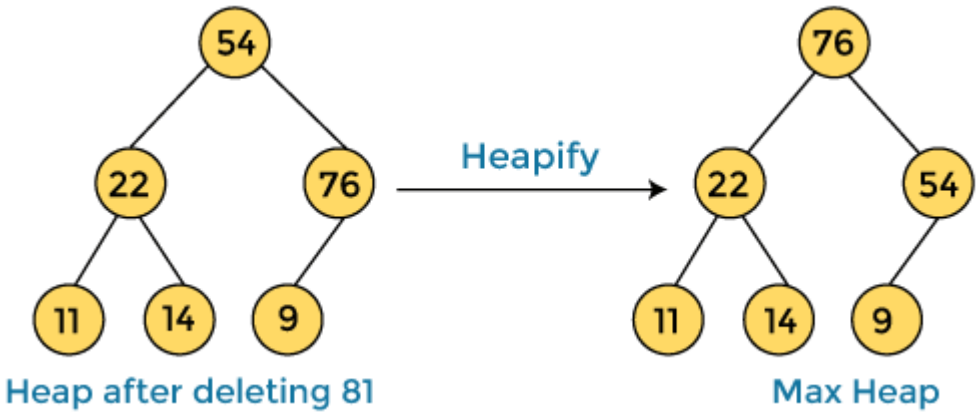


Next, we have to delete the root element **(89)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



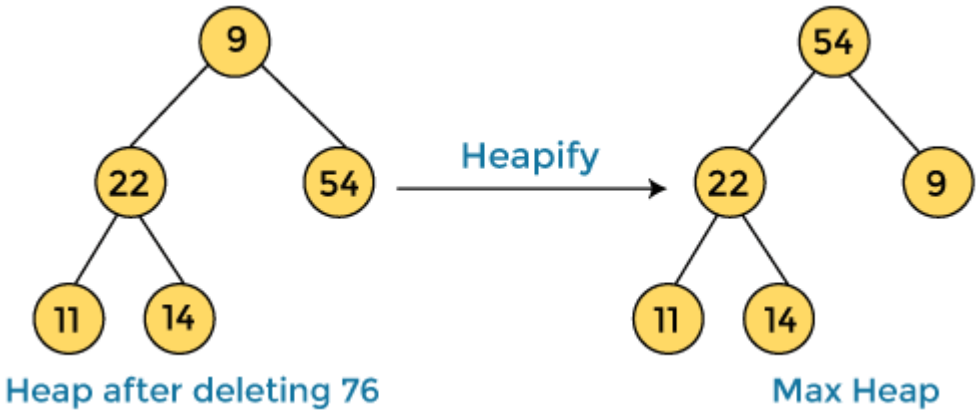
After swapping the array element **89** with **11,** and converting the heap into max-heap, the elements of array are -

In the next step, again, we have to delete the root element **(81)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(54).** After deleting the root element, we again have to heapify it to convert it into max heap.



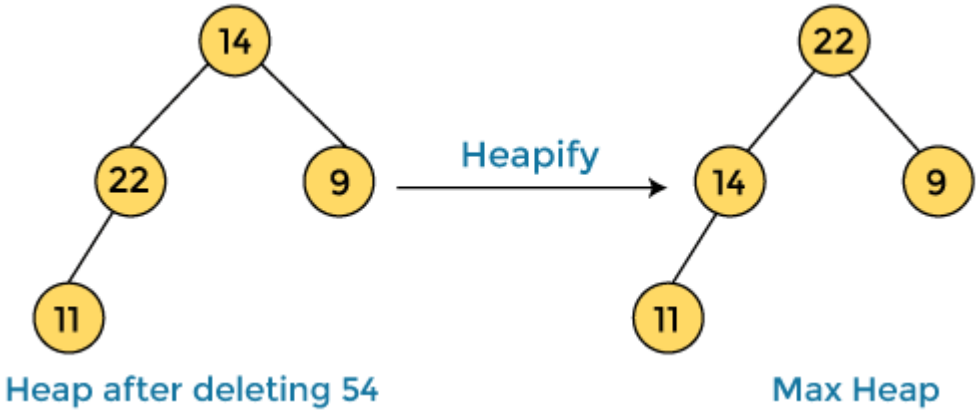
After swapping the array element **81** with **54** and converting the heap into max-heap, the elements of array are -

In the next step, we have to delete the root element **(76)** from the max heap again. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

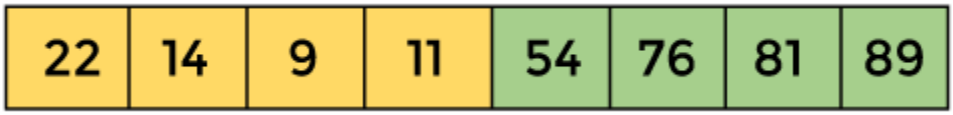


After swapping the array element **76** with **9** and converting the heap into max-heap, the elements of array are -

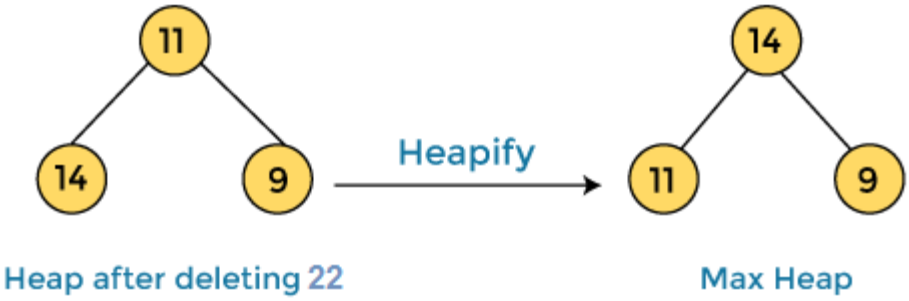
In the next step, again we have to delete the root element **(54)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(14).** After deleting the root element, we again have to heapify it to convert it into max heap.



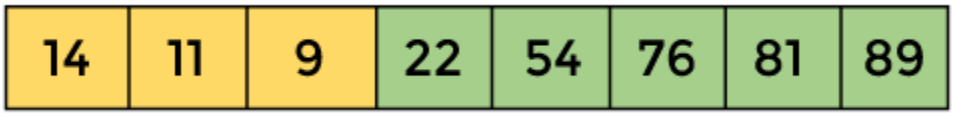
After swapping the array element **54** with **14** and converting the heap into max-heap, the elements of array are -



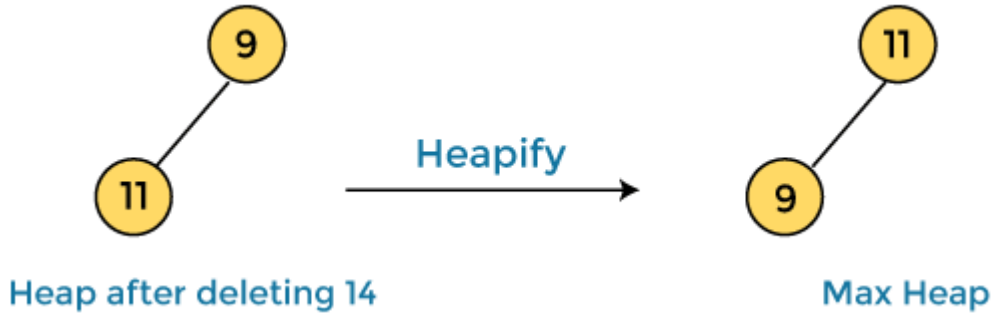
In the next step, again we have to delete the root element **(22)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(11).** After deleting the root element, we again have to heapify it to convert it into max heap.



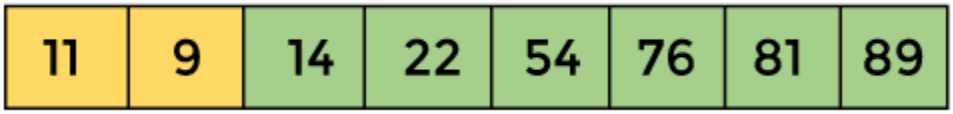
After swapping the array element **22** with **11** and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element **(14)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

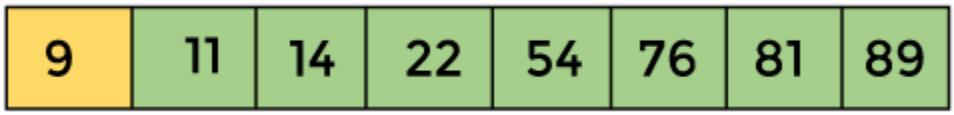


After swapping the array element **14** with **9** and converting the heap into max-heap, the elements of array are -



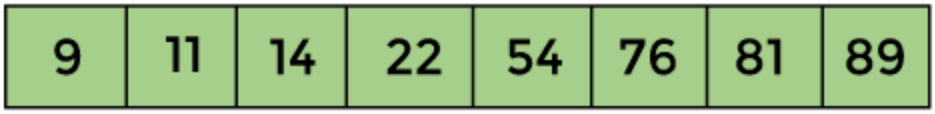
In the next step, again we have to delete the root element **(11)** from the max heap. To delete this node, we have to swap it with the last node, i.e. **(9).** After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element **11** with **9,** the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -



Now, the array is completely sorted.

1. Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of heap sort is **O(n logn).**
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of heap sort is **O(n log n).**
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of heap sort is **O(n log n).**

The time complexity of heap sort is **O(n logn)** in all three cases (best case, average case, and worst case). The height of a complete binary tree having n elements is **logn.**

2. Space Complexity

Space Complexity	O(1)
Stable	N0

- The space complexity of Heap sort is O(1).

Q.13. Sort the letters of word “DESIGN” in alphabetical order using bubble sort.

ANS:-

Q.14. Write an algorithm for insertion sort. Analyze insertion sort algorithm for best case and worst case.

ANS:-

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

The numbers, which are needed to be sorted, are known as **keys**. Here is the algorithm of the insertion sort method.

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
  key = A[j]
  i = j – 1
  while i > 0 and A[i] > key
    A[i + 1] = A[i]
    i = i -1
  A[i + 1] = key
```

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **$O(n^2)$**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the ascending order of an array into the descending order. In this algorithm, every individual element is compared with the rest of the elements, due to which n-1 comparisons are made for every n^{th} element.

Q.15. Explain Counting sort with example.

ANS:-

Counting sort is a sorting technique that is based on the keys between specific ranges. In coding or technical interviews for software engineers, sorting algorithms are widely asked. So, it is important to discuss the topic.

This sorting technique doesn't perform sorting by comparing elements. It performs sorting by counting objects having distinct key values like hashing. After that, it performs some arithmetic operations to calculate each object's index position in the output sequence. Counting sort is not used as a general-purpose sorting algorithm.

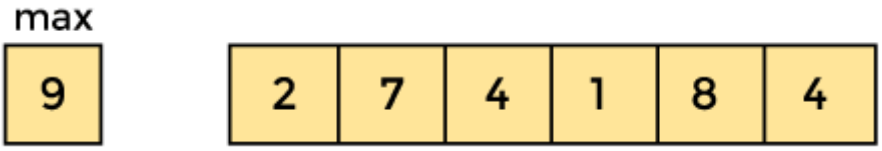
Working of counting sort Algorithm

Now, let's see the working of the counting sort Algorithm.

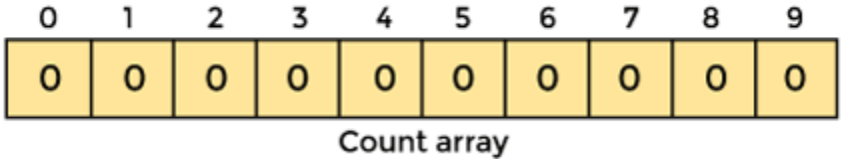
To understand the working of the counting sort algorithm, let's take an unsorted array. It will be easier to understand the counting sort via an example.

Let the elements of array are -

1. Find the maximum element from the given array. Let **max** be the maximum element.

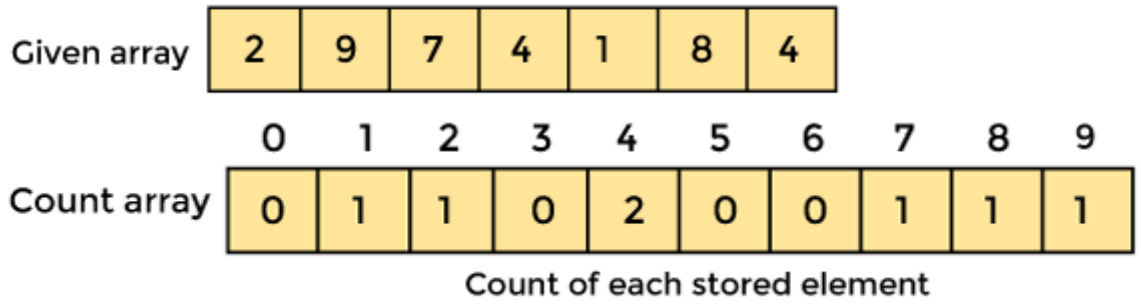


2. Now, initialize array of length **max + 1** having all 0 elements. This array will be used to store the count of the elements in the given array.

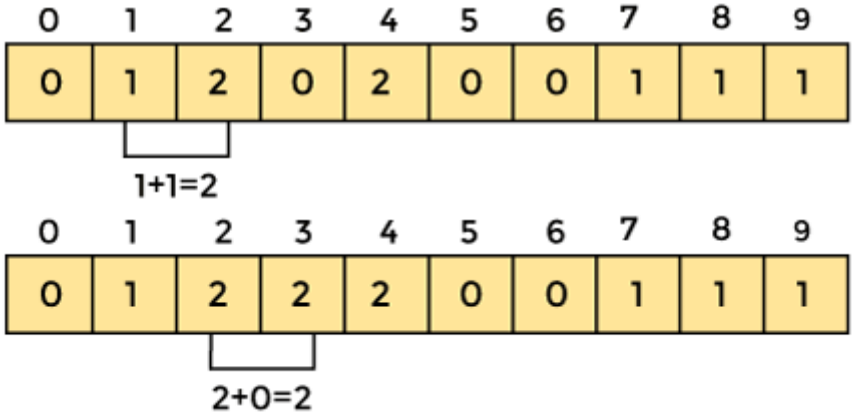


3. Now, we have to store the count of each array element at their corresponding index in the count array.

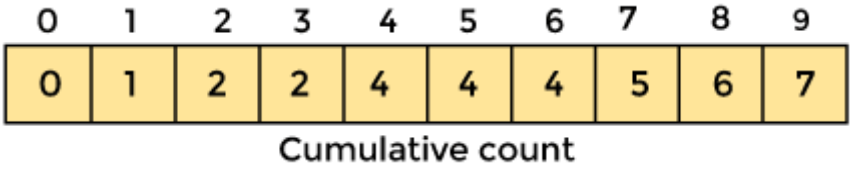
The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4th position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3rd position.



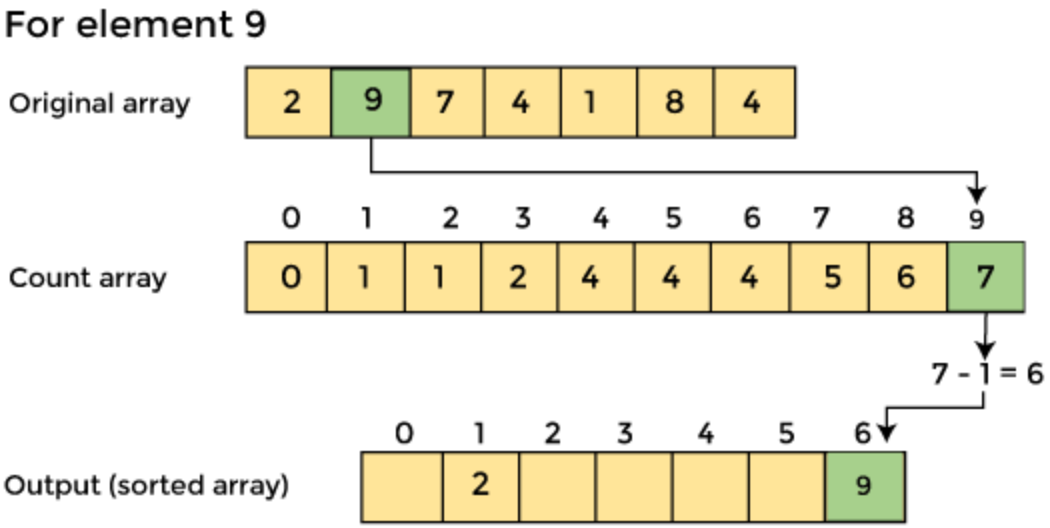
Now, store the cumulative sum of **count** array elements. It will help to place the elements at the correct index of the sorted array.



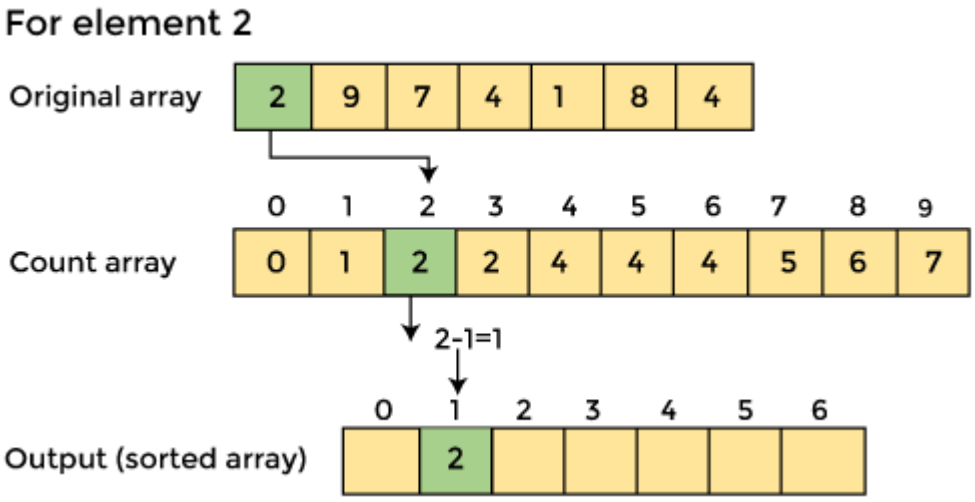
Similarly, the cumulative count of the count array is -



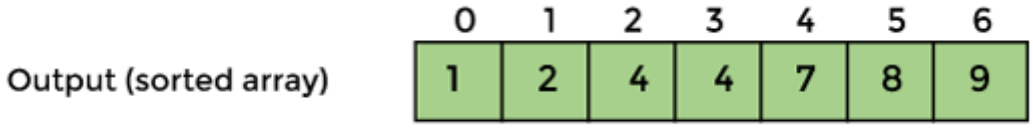
4. Now, find the index of each element of the original array



After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.



Similarly, after sorting, the array elements are -



Now, the array is completely sorted.

Counting sort complexity

Now, let's see the time complexity of counting sort in best case, average case, and in worst case. We will also see the space complexity of the counting sort.

1. Time Complexity

Case	Time	Complexity
Best Case	O(n + k)	
Average Case	O(n + k)	
Worst Case	O(n + k)	

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of counting sort is **O(n + k)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of counting sort is **O(n + k)**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of counting sort is **O(n + k)**.

Q.16. Sort the following data with Heap Sort Method: 20, 50, 30, 75, 90, 60, 25, 10, and 40. And explain it.

ANS:-

Q.17. What is an amortized analysis? Explain aggregate method of amortized analysis using suitable example.

ANS:-

Amortize Analysis

This analysis is used when the occasional operation is very slow, but most of the operations which are executing very frequently are faster. Data structures we need amortized analysis for Hash Tables, Disjoint Sets etc.

In the Hash-table, the most of the time the searching time complexity is O(1), but sometimes it executes O(n) operations. When we want to search or insert an element in a hash table for most of the cases it is constant time taking the task, but when a collision occurs, it needs O(n) times operations for collision resolution.

Aggregate Method

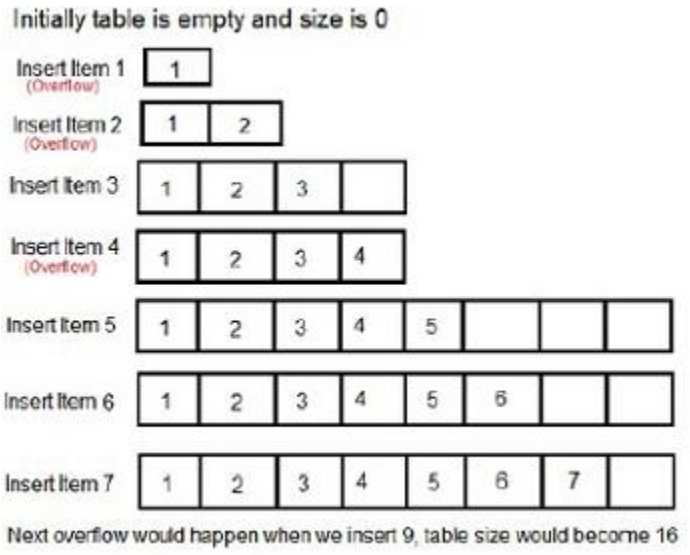
The aggregate method is used to find the total cost. If we want to add a bunch of data, then we need to find the amortized cost by this formula.

For a sequence of n operations, the cost is −

$$\frac{Cost(n\ operations)}{n} = \frac{Cost(normal\ operations) + Cost(Expensive\ operations)}{n}$$

Example on Amortized Analysis

For a dynamic array, items can be inserted at a given index in O(1) time. But if that index is not present in the array, it fails to perform the task in constant time. For that case, it initially doubles the size of the array then inserts the element if the index is present.



For the dynamic array, let = cost of *ith* insertion.

$$\text{So } ci = 1 + \begin{cases} i-1, & if\ i-1\ is\ power\ of\ 2 \\ 0, & Otherwise \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} \leq \frac{n + \sum_{j=1}^{\lfloor log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n}$$

Q.18. Explain Selection Sort Algorithm and give its best case, worst case and average case complexity with example.

ANS:-

Selection Sort

The selection sort enhances the bubble sort by making only a single swap for each pass through the rundown. In order to do this, a selection sort searches for the biggest value as it makes a pass and, after finishing the pass, places it in the best possible area. Similarly, as with a bubble sort, after the first pass, the biggest item is in the right place. After the second pass, the following biggest is set up. This procedure proceeds and requires n-1 goes to sort n item since the last item must be set up after the (n-1) th pass.

ALGORITHM: SELECTION SORT (A)

1. 1. k ← length [A]
2. 2. **for** j ←1 to n-1
3. 3. smallest ← j
4. 4. **for** I ← j + 1 to k
5. 5. **if** A [i] < A [smallest]
6. 6. then smallest ← i
7. 7. exchange (A [j], A [smallest])

How Selection Sort works

1. In the selection sort, first of all, we set the initial element as a **minimum**.

2. Now we will compare the minimum with the second element. If the second element turns out to be smaller than the minimum, we will swap them, followed by assigning to a minimum to the third element.
3. Else if the second element is greater than the minimum, which is our first element, then we will do nothing and move on to the third element and then compare it with the minimum.
   We will repeat this process until we reach the last element.
4. After the completion of each iteration, we will notice that our minimum has reached the start of the unsorted list.
5. For each iteration, we will start the indexing from the first element of the unsorted list. We will repeat the Steps from 1 to 4 until the list gets sorted or all the elements get correctly positioned.
   Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

   A [] = (7, 4, 3, 6, 5).
   A [] =

1st Iteration:

Set minimum = 7

- Compare a_0 and a_1

As, $a_0 > a_1$, set minimum = 4.

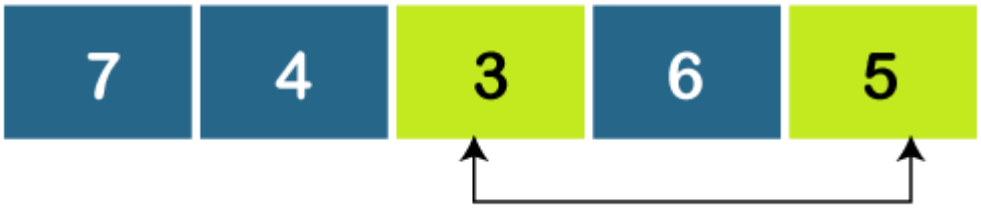
- Compare a_1 and a_2

As, $a_1 > a_2$, set minimum = 3.

- Compare a_2 and a_3

As, $a_2 < a_3$, set minimum= 3.

- Compare a_2 and a_4



As, $a_2 < a_4$, set minimum =3.

Since 3 is the smallest element, so we will swap a_0 and a_2.

2nd Iteration:

Set minimum = 4

- Compare a_1 and a_2

As, $a_1 < a_2$, set minimum = 4.

- Compare a_1 and a_3

As, A[1] < A[3], set minimum = 4.

- Compare a_1 and a_4

Again, $a_1 < a_4$, set minimum = 4.

Since the minimum is already placed in the correct position, so there will be no swapping.

3rd Iteration:

Set minimum = 7

- Compare a_2 and a_3

As, $a_2 > a_3$, set minimum = 6.

- Compare a_3 and a_4

As, $a_3 > a_4$, set minimum = 5.

Since 5 is the smallest element among the leftover unsorted elements, so we will swap 7 and 5.

4th Iteration:

Set minimum = 6

- Compare a_3 and a_4

As $a_3 < a_4$, set minimum = 6.

Since the minimum is already placed in the correct position, so there will be no swapping.

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of **$O(n^2)$** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is **$O(n^2)$**, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.

- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

Q.19. Sort the letters of word “EDUCATION” in alphabetical order using insertion sort.

ANS:-

Q.20. Apply the bubble sort algorithm for sorting {U,N,I,V,E,R,S}.

ANS:-

Q.21. Let f(n) and g(n) be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that max(f(n), g(n)) = Θ (f(n) + g(n)).

ANS:-

To prove this, we have to show that there exists constants c_1, c_2, n_0 > 0$c_1, c_2, n_0 > 0$ such that for all n \ge n_0$n \ge n_0$,
0 \le c_1 (f(n) + g(n)) \le \max(f(n), g(n)) \le c_2 (f(n) + g(n))$0 \le c_1 (f(n) + g(n)) \le \max(f(n), g(n)) \le c_2 (f(n) + g(n))$
As the functions are asymptotically non-negative, we can assume that for some n_0 > 0$n_0 > 0$, f(n) \ge 0$f(n) \ge 0$ and g(n) \ge 0$g(n) \ge 0$. Therefore, n \ge n_0$n \ge n_0$,
f(n) + g(n) \ge \max(f(n), g(n))$f(n) + g(n) \ge \max(f(n), g(n))$
Also note that, f(n) \le \max(f(n), g(n))$f(n) \le \max(f(n), g(n))$ and g(n) \le \max(f(n), g(n))$g(n) \le \max(f(n), g(n))$

\begin{aligned} f(n) + g(n) &\le 2 \max(f(n), g(n)) \\ \frac 1 2 (f(n) + g(n)) &\le \max(f(n), g(n)) \end{aligned}$f(n)+g(n)21(f(n)+g(n))$ $\le 2\max(f(n),g(n)) \le \max(f(n),g(n))$
Therefore, we can combine the above two inequalities as follows:

0 \le \frac 1 2 (f(n) + g(n)) \le \max(f(n), g(n)) \le (f(n) + g(n)) \text { for } n \ge n_00$\le 21(f(n)+g(n)) \le \max(f(n),g(n)) \le (f(n)+g(n))$ for $n \ge n_0$
So, \max(f(n), g(n)) = \Theta(f(n) + g(n))$\max(f(n),g(n)) = \Theta(f(n)+g(n))$ because there exists c_1 = 0.5c_1 =0.5 and c_2 = 1c_2=1.

Q.22. What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2n on the same machine?

ANS:-

For inputs of size n, running time of algorithm A is 100n2 and of B is 2n. For A to run faster than B, 100n2 must be smaller than 2n.

Calculate: A (quadratic time complexity) will run much faster than B (exponential time complexity) for very large values of n. Let's start checking from n=1 and go up for values of n which are power of 2.

n=1⇒ 100×1^2 = 100 > 2^1

n=2⇒ 100×2^2 = 400 > 2^2

n=4⇒ 100×4^2 = 1600 > 2^4

n=8⇒ 100×8^2 = 6400 > 2^8

n=16⇒ 100×16^2 = 25600 < 2^16

Somewhere between 8 and 16, A starts to run faster than B. Let's do what we were doing but now we are going to try middle value of the range, repeatedly (binary search).

n=8+16/ 2 = 12⇒100×12^2=14400 > 2^12

n=12+16/ 2=14⇒100×14^2=19600 > 2^14

n=14+16/ 2=15⇒100×15^2=22500 < 2^15

So, at n=15, A starts to run faster than B.

Q.23. Explain Tower of Hanoi Problem, Derive its recursion equation and computer it's time complexity.

ANS:-

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.
Pseudo Code

Analysis of Recursion

Recursive Equation : ——-equation-1

Solving it by Backsubstitution :

————equation-2

————equation-3

Put the value of T(n-2) in the equation–2 with help of equation-3

——equation-4

Put the value of T(n-1) in equation-1 with help of equation-4

After Generalization :

Base condition T(1) =1
n – k = 1
k = n-1
put, k = n-1

It is a GP series, and the sum is

, or you can say which is exponential

for 5 disks i.e. n=5 It will take 2^5-1=31 moves.

Unit-3

Q.1. Prove that Greedy Algorithms does not always give optimal solution. What are the general characteristics of Greedy Algorithms? Also compare GreedyAlgorithms

with Dynamic Programming and Divide and Conquer methods to find out major difference between them.

ANS:-

A greedy algorithm picks the best *local* choice at every point it has to make a decision.

Overall, though, a combination of the best *local* choices most likely is not the best *global* solution for the problem. This is because they can make commitments to certain choices too early which prevent them from finding the best overall solution later.

Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum

Greedy Algorithms work step-by-step, and always choose the steps which provide immediate profit/benefit. It chooses the "locally optimal solution", without thinking about future consequences. Greedy algorithms may not always lead to the optimal global solution, because it does not consider the entire data.

- To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Divide and Conquer	Dynamic Programming
The algorithms that follow divide and conquer paradigm are mostly recursive in nature.	The algorithms that follow the dynamic programming paradigmare mostly non-recursive.
Divide and conquer algorithms are less efficient as compared to the dynamic programming algorithms.	Dynamic programming algorithms are more efficient.
In this paradigm, all the data has to be computed even if it was processed.	Each processed steps are saved to be reused in the future if needed.
It combines the solutions of the sub-problems to obtain a solution of the main problem.	It uses the result of the sub-problems to find the optimal solution of the main problem.
Divide and conquer algorithms generally consumes more time for execution.	Dynamic programming algorithms consumes less time in general.
Sub-problems are independent.	Sub-problems are interdependent.
Example of a divide and conquer algorithm is Binary search.	Example of a dynamic programming algorithm is Fibonacci number.

Q.2. Justify the general statement that “if a problem can be split using Divide and Conquer strategy in almost equal portions at each stage, then it is a good can did ate for recursive implementation, but if it cannot be easily be so divided in equal portions, then it better be implemented iteratively”. Explain with an example.

ANS:-

Q.3. Write an algorithm for binary search. Calculate the time complexity for each case.

ANS:-

Begin
Set beg = 0

```
Set end = n-1
Set mid = (beg + end) / 2
while ( (beg <= end) and (a[mid] ≠ item) ) do
if (item < a[mid]) then
Set end = mid - 1
else
Set beg = mid + 1
endif
Set mid = (beg + end) / 2
endwhile
if (beg > end) then
Set loc = -1
else
Set loc = mid
endif
End
```

Explanation

Binary Search Algorithm searches an element by comparing it with the middle most element of the array.

Then, following three cases are possible-

Case-01

If the element being searched is found to be the middle most element, its index is returned.

Case-02

If the element being searched is found to be greater than the middle most element, then its search is further continued in the right sub array of the middle most element.

Case-03

If the element being searched is found to be smaller than the middle most element, then its search is further continued in the left sub array of the middle most element.

This iteration keeps on repeating on the sub arrays until the desired element is found or size of the sub array reduces to zero.

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have-

Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.

Here, n is the number of elements in the sorted linear array.

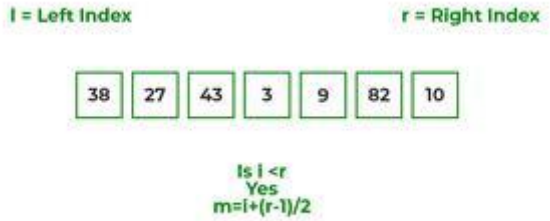
This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

Q.4. Write an algorithm for merge sort with divide and conquer strategy.Analyze each case.List best case worst case and average case complexity

ANS:-

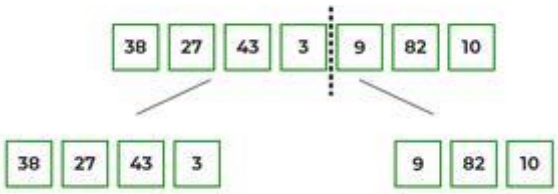
To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

- At first, check if the left index of array is less than the right index, if yes then calculate its mid point

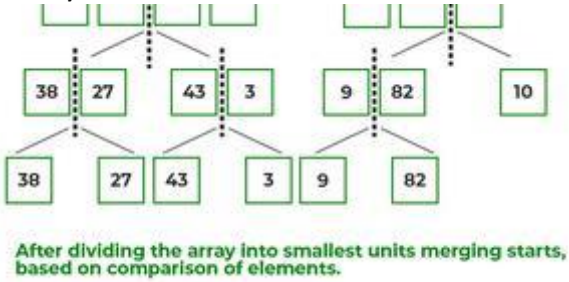


- Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.
- Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

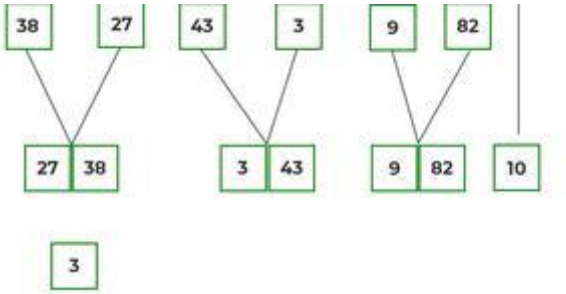
- Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



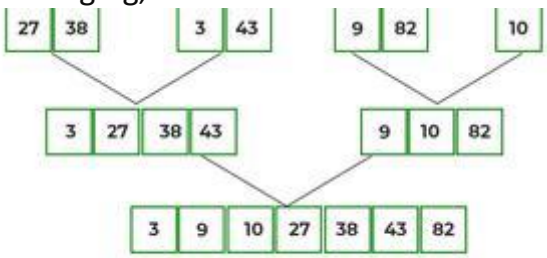
- Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



- After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
- Firstly, compare the element for each list and then combine them into another list in a sorted manner.

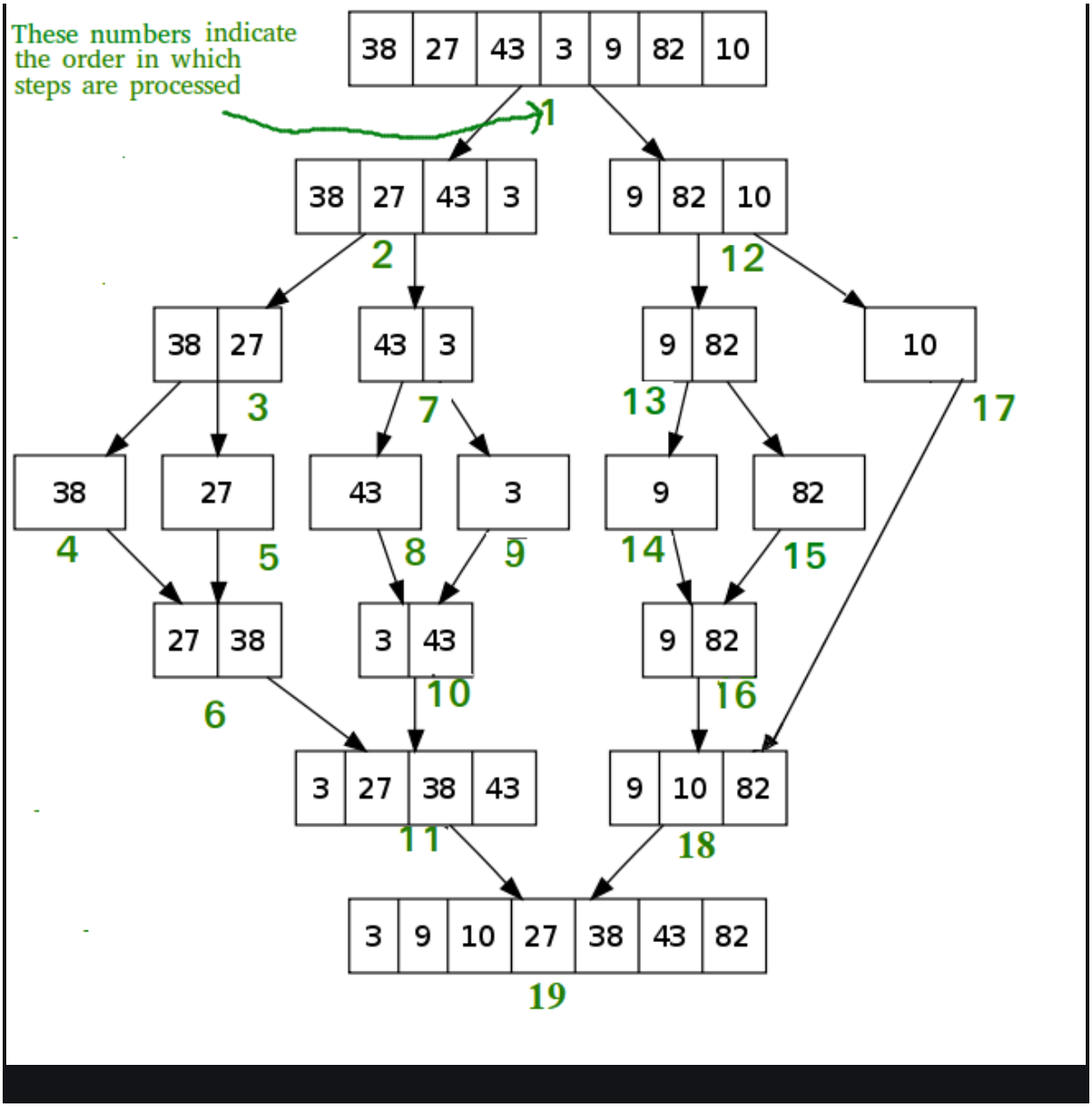


- After the final merging, the list looks like this:



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Worst Case Time Complexity [Big-O]: **O(n*log n)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n)**

- Time complexity of Merge Sort is O(n*Log n) in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.

- It requires **equal amount of additional space** as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.

Q.5. Write an algorithm for quick sort with divide and conquer strategy.Analyze each case. In whichcase it performs similar to selection sort?

ANS:-

Algorithm:

```
1. QUICKSORT (array A, start, end)
2. {
3.  1 if (start < end)
4.  2 {
5. 3 p = partition(A, start, end)
6. 4 QUICKSORT (A, start, p - 1)
7. 5 QUICKSORT (A, p + 1, end)
8. 6 }
9. }
```

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

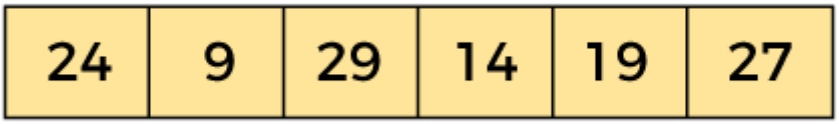
```
1.  PARTITION (array A, start, end)
2.  {
3.   1 pivot ? A[end]
4.   2 i ? start-1
5.   3 for j ? start to end -1 {
6.   4 do if (A[j] < pivot) {
7.   5 then i ? i + 1
8.   6 swap A[i] with A[j]
9.   7  }}
10.  8 swap A[i+1] with A[end]
11.  9 return i+1
12.  }
```

Working of Quick Sort Algorithm

Now, let's see the working of the Quicksort Algorithm.

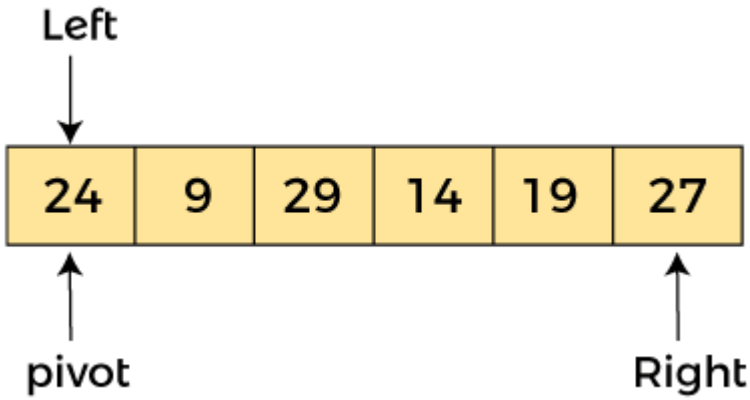
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

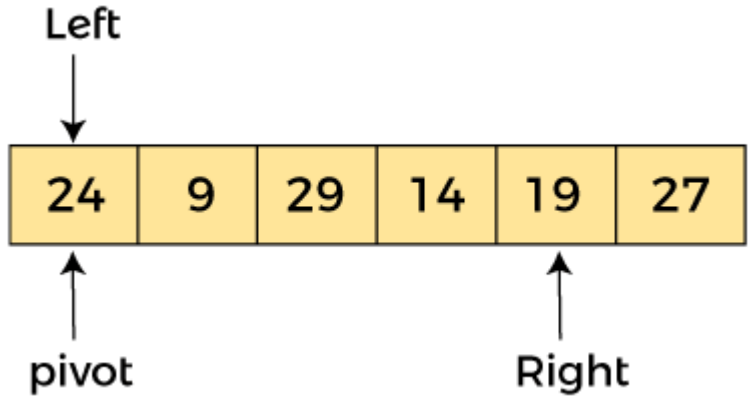


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

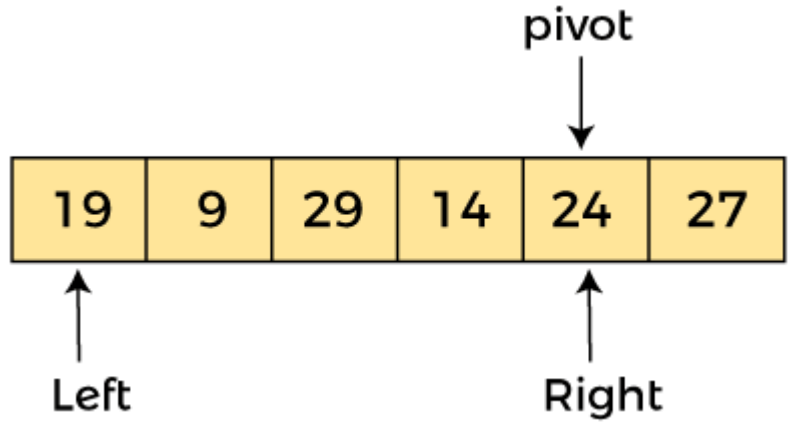


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



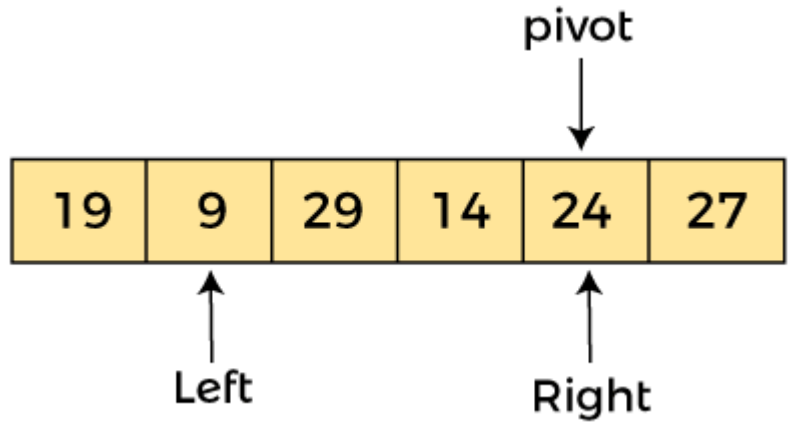
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

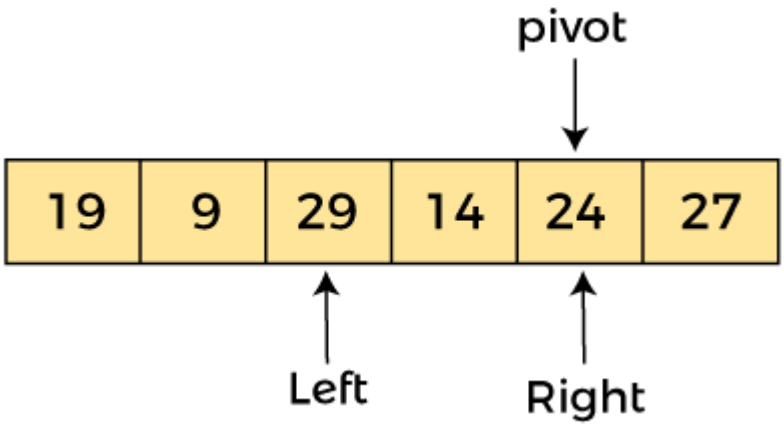


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

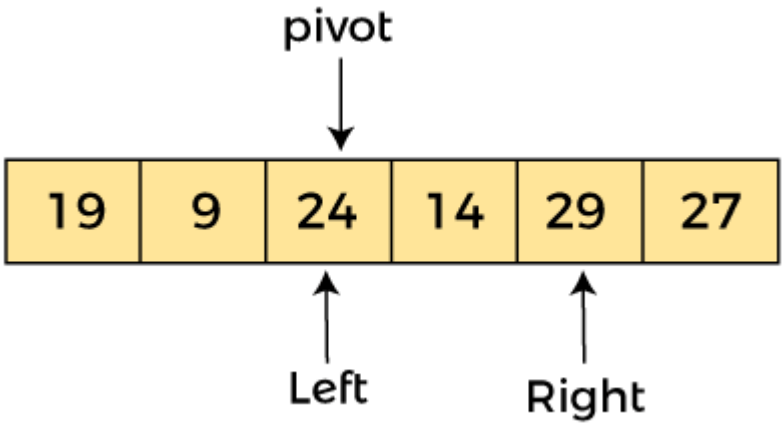
As a[pivot] > a[left], so algorithm moves one position to right as -



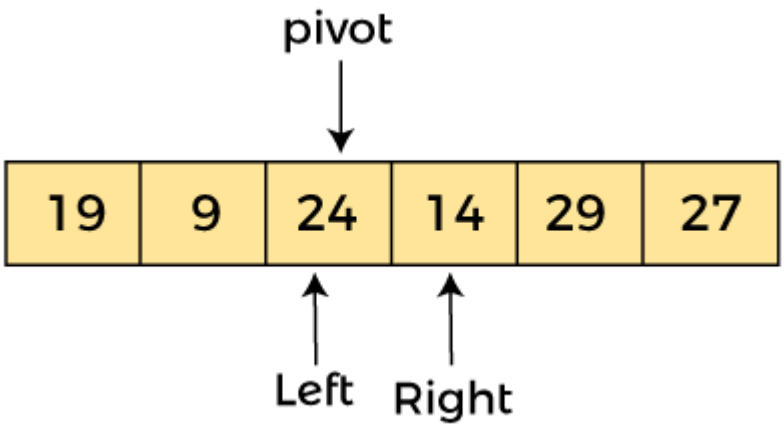
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



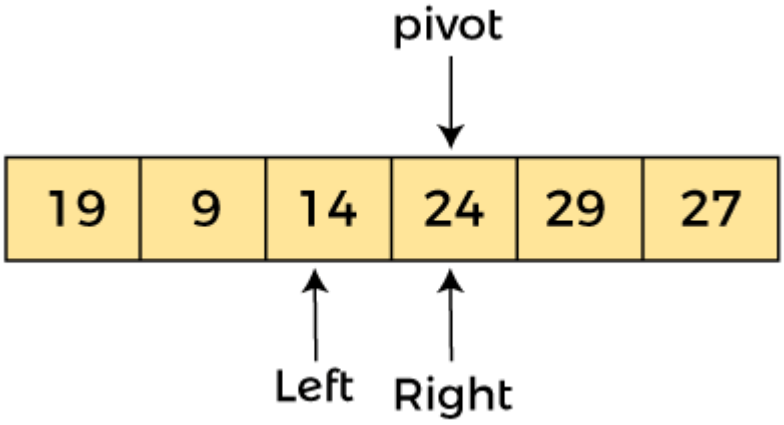
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



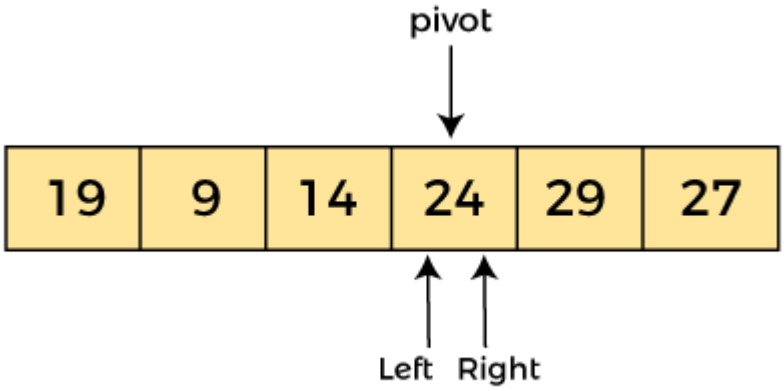
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



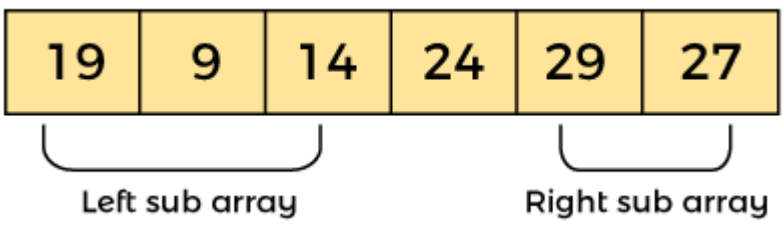
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



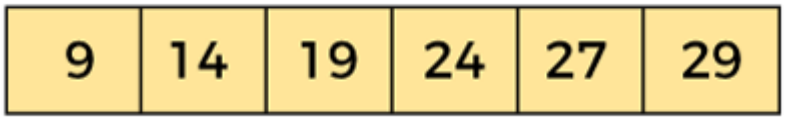
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



Quicksort complexity

Now, let's see the time complexity of quicksort in best case, average case, and in worst case. We will also see the space complexity of quicksort.

1. Time Complexity

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

- **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n*logn)**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n*logn)**.
- **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**.

Though the worst-case complexity of quicksort is more than other sorting algorithms such as **Merge sort** and **Heap sort**, still it is faster in practice. Worst case in quick sort rarely occurs because by changing the choice of pivot, it can be implemented in different ways. Worst case in quicksort can be avoided by choosing the right pivot element.

Q.6. Differentiate divide and conquer with dynamic programming. Write recurrence for calculation for binominal coefficient.

ANS:-