



MASENO
UNIVERSITY
Fountain of Excellence

DESIGN AND ANALYSIS OF ALGORITHMS –CIT 301

Moses Wainaina

Contacts: 0707691430

Email: wainainamoses047@gmail.com

Course Outline

TOPIC 1: Introduction to algorithms

- Introduction to algorithms
- Introduction Design and analysis of algorithms
- Applications of Algorithms

TOPIC 2: Algorithm Design technique and algorithm analysis

- Introduction
- Design Techniques
- Analysis techniques
- Complexity theory; asymptotic analysis of upper and average complexity: least, average and worst analysis of algorithms
- amortized analysis: big 'O' and little 'o' notation: time and space trade –off in algorithm

TOPIC 3: Search Algorithm

- Linear Search Algorithms
- Binary Search Algorithms

TOPIC 4: Sorting

- Introduction
- Sorting Techniques
- Sorting Analysis

TOPIC 5: Trees and algorithm

- Introduction to tree and algorithms
- Application areas of trees and algorithms

TOPIC 6: Graphs and Algorithms

- Introduction graphs and algorithms
- Application areas of graphs and algorithms

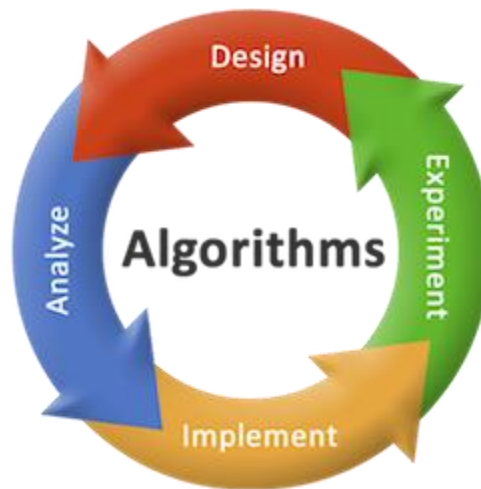
TOPIC 7: Short path algorithm

TOPIC 1: INTRODUCTION TO ALGORITHMS

Algorithm: A finite set of instruction that specifies a sequence of operation to be carried out in order to solve a specific problem or class of problems.

An algorithm can be defined as a well-defined computational procedure that takes some values, or the set of values, as an input and produces some value, or the set of values, as an output. An algorithm is thus a sequence of computational steps that transform the input into output.

Design, analysis, experiment and implementation



Analysis of algorithms is the process of finding the computational complexity of algorithms – the amount of time, storage, or other resources needed to execute them. Usually, this involves determining a function that relates the length of an algorithm's input to the number of steps it takes (its time complexity) or the number of storage locations it uses (its space complexity).

Worst-case: $f(n)$ defined by the maximum number of steps taken on any instance of size n .

Best-case: $f(n)$ defined by the minimum number of steps taken on any instance of size n .

Average case: $f(n)$ defined by the average number of steps taken on any instance of size n

Algorithm design refers to a method (Technique) or a mathematical process for problem-solving also known as algorithm engineering e.g. Divide and Conquer.

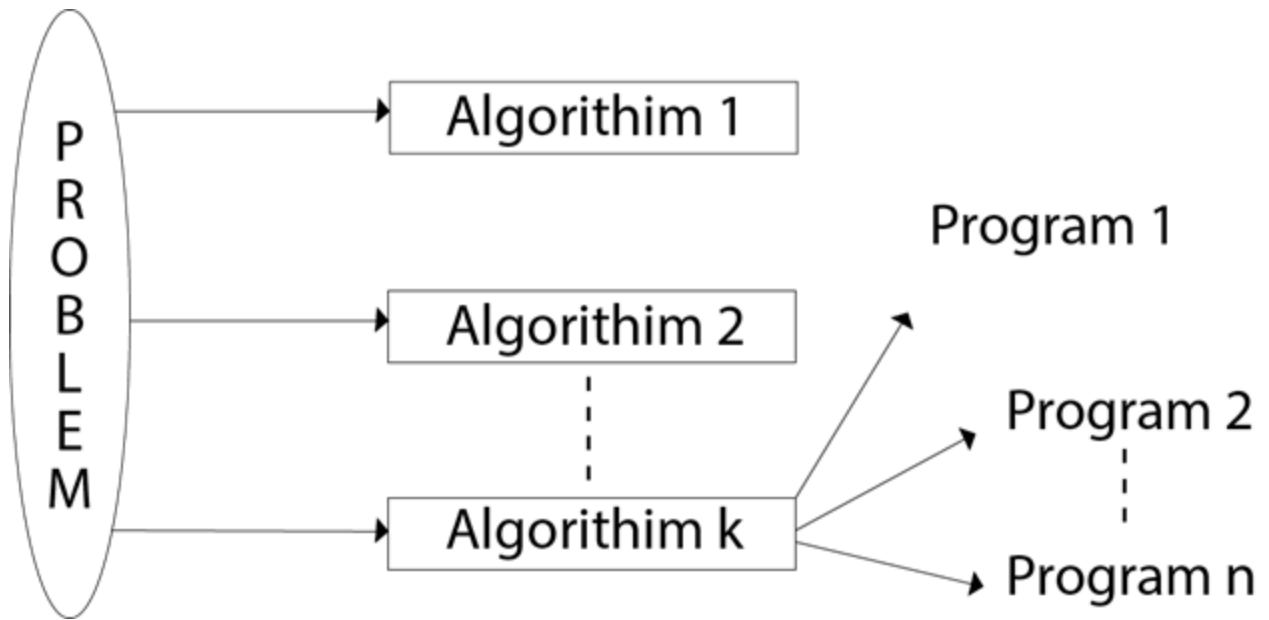
NEED OF ALGORITHM

1. To understand the basic idea of the problem
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. To understand the basic principles of designing the algorithms.
5. To compare the performance of the algorithm with respect to other techniques.
6. It is the best method of description without describing the implementation detail.
7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
8. A good design can produce a good solution.
9. To understand the flow of the problem.
10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)
11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
12. With the help of algorithm, we convert art into a science.
13. To understand the principle of designing.
14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design

ALGORITHM VS PROGRAM.

A finite set of instructions that specifies a sequence of operations to be carried out to solve a specific problem of a class of problem is called an algorithm.

On the other hand, the Program doesn't have to satisfy the finiteness condition. For example, we can think of an operating system that continues in a "wait" loop until more jobs are entered. Such a program doesn't terminate unless the system crashes.



Given a Problem to solve, the design Phase produces an algorithm, and the implementation phase then generates a program that expresses the designed algorithm. So, the concrete expression of an algorithm in a particular programming language is called a program.

An algorithm must have the following properties:

- a. **Correctness:** It should produce the output according to the requirement of the algorithm
- b. **Finiteness:** Algorithm must complete after a finite number of instructions have been executed. **An Absence of Ambiguity:** Each step must be defined, having only one interpretation.
- c. **Definition of Sequence:** Each step must have a unique defined preceding and succeeding step. The first step and the last step must be noted.
- d. **Input/output:** Number and classification of needed inputs and results must be stated.
Feasibility: It must be feasible to execute each instruction.
- e. **Flexibility:** It should also be possible to make changes in the algorithm without putting so much effort on it.
- f. **Efficient** - Efficiency is always measured in terms of time and space requires implementing the algorithm, so the algorithm uses a little running time and memory space as possible within the limits of acceptable development time.
- g. **Independent:** An algorithm should focus on what are inputs, outputs and how to derive output without knowing the language it is defined. Therefore, we can say that the algorithm is independent of language.

Applications of Algorithms

- a. Chatbots e.g facebook
- b. Computer networks
- c. Stock Market **Algorithm**
- d. Autopilot algorithm
- e. Facial Recognition **Algorithm.**
- f. Beauty Pageant **Algorithm.**
- g. Medical diagnosis
- h. Most program automation require algorithms
Data structures+ Algorithms=Programs(efficient)

TOPIC 2: ALGORITHM DESIGN TECHNIQUES AND ALGORITHM ANALYSIS

Algorithm Design Techniques

Algorithm design is a specific method to create a mathematical process in solving problems. Applied algorithm design is algorithm engineering.

One of the most important aspects of algorithm design is creating an algorithm that is efficient.

Techniques used in Designing Algorithm

- a. Simple recursive algorithms
- b. Backtracking algorithms
- c. Divide and conquer algorithms
- d. Dynamic programming algorithms
- e. Greedy algorithms
- f. Branch and bound algorithms
- g. Brute force algorithms

a) Simple recursive algorithms

A recursive algorithm calls itself with smaller input values and returns the result for the current input by carrying out basic operations on the returned value for the smaller input. Generally, if a problem can be solved by applying solutions to smaller versions of the same problem, and the smaller versions shrink to readily solvable instances, then the problem can be solved using a recursive algorithm.

NB: An algorithm is said to be recursive if it calls itself

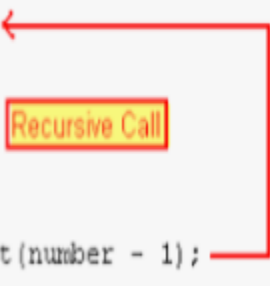
For example consider factorial of a number, $n! = n * (n-1) * (n-2) * \dots * 2 * 1$, and that $0! = 1$.

Function to calculate the factorial can be written as

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else return (n * factorial(n-1));
}
```

Example 2

```
//Very simple example
public int Fact(int number)
{
    if (number == 0)
        return 1;
    else
        return number * Fact(number - 1);
}
```



b) Backtracking Algorithm

Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

Backtracking Algorithm

Backtrack(x)

if x is not a solution

return false

if x is a new solution

add to list of solutions

backtrack(expand x)

Example Backtracking Approach

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

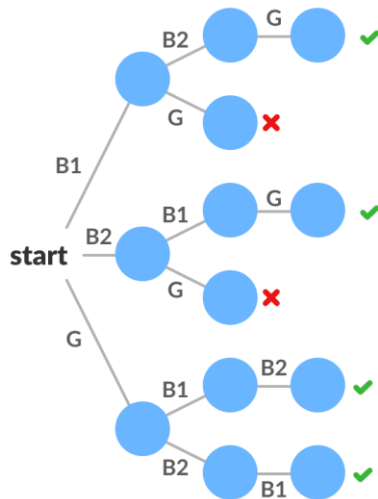
Constraint: Girl should not be on the middle bench.

Solution: There are a total of $3! = 6$ possibilities. We will try all the possibilities and get the possible solutions. We recursively try all the possibilities.

All the possibilities are:

B1	B2	G	B2	G	B1
B1	G	B2	G	B1	B2
B2	B1	G	G	B2	B1

The following [state space tree](#) shows the possible solutions.



NB: State Space Tree

A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state

Backtracking Algorithm Applications

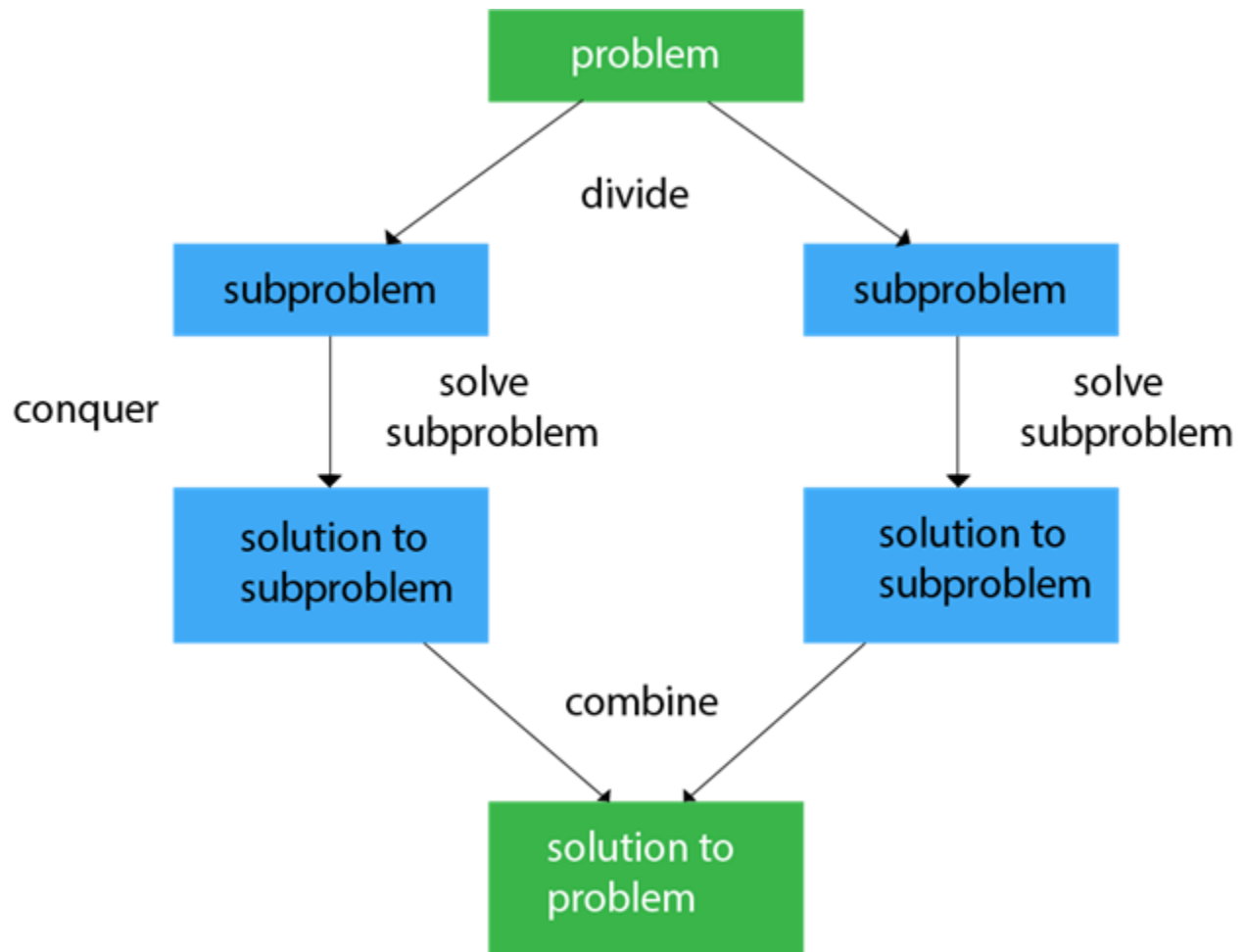
1. To solve the N Queen problem.
2. Maze solving problem.
3. The Knight's tour problem.

(Research More about these applications)

c. Divide and Conquer

It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

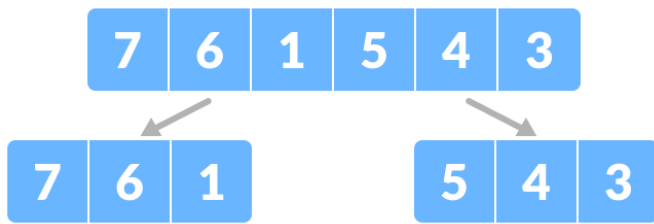
- Divide the original problem into a set of sub problems.
- Solve every sub problem individually, recursively.
- Combine the solution of the sub problems (top level) into a solution of the whole original problem



EXAMPLE: Let the given array be:

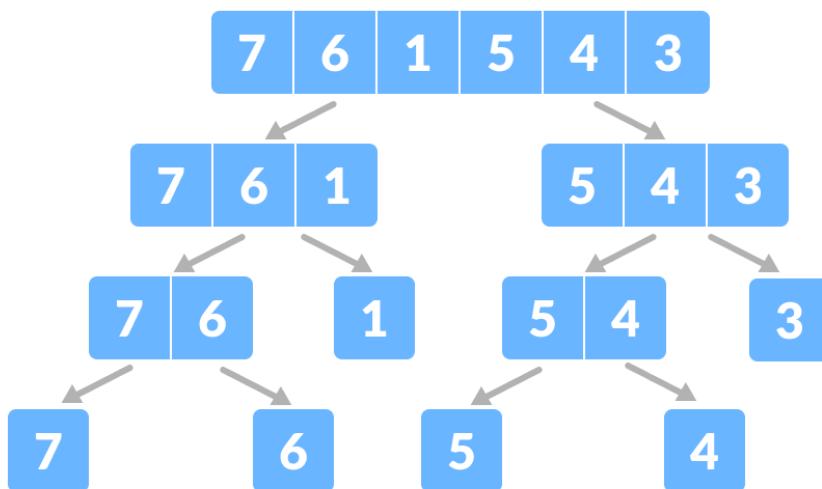
7	6	1	5	4	3
---	---	---	---	---	---

Array for merge sort

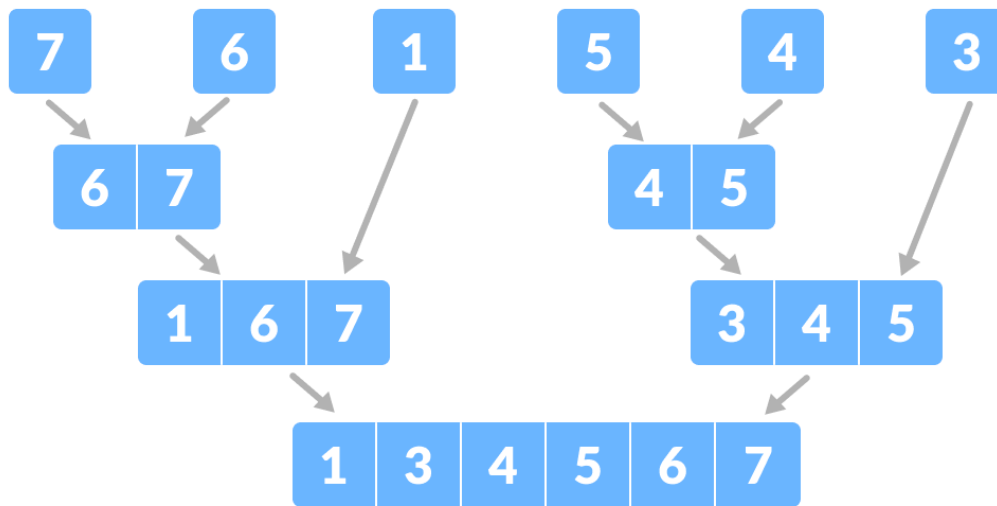


Divide the array into two halves

Again, divide each subpart recursively into two halves until you get individual elements.



Now, combine the individual elements in a sorted manner. Here, **conquer** and **combine** steps go side by side.



d. Dynamic Programming Algorithm

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping sub problems.

If any problem can be divided into sub problems, which in turn are divided into smaller sub problems, and if there are overlapping among these sub problems, then the solutions to these sub problems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

- Requires “optimal substructure” and “overlapping subproblems”
- Optimal substructure: Optimal solution contains optimal solutions to subproblems
- Overlapping subproblems: Solutions to subproblems can be stored and reused in a bottom-up fashion

Dynamic Programming Example

Let's find the fibonacci sequence upto 5th term. A fibonacci series is the sequence of numbers in which each number is the sum of the two preceding ones. For example, 0,1,1, 2, 3. Here, each number is the sum of the two preceding numbers.

Algorithm

Let n be the number of terms.

1. If $n \leq 1$, return 1.

2. Else, return the sum of two preceding numbers.

We are calculating the fibonacci sequence up to the 5th term.

1. The first term is 0.
2. The second term is 1.
3. The third term is sum of 0 (from step 1) and 1(from step 2), which is 1.
4. The fourth term is the sum of the third term (from step 3) and second term (from step 2)
i.e. $1 + 1 = 2$.
5. The fifth term is the sum of the fourth term (from step 4) and third term (from step 3)
i.e. $2 + 1 = 3$.

Hence, we have the sequence 0,1,1, 2, 3. Here, we have used the results of the previous steps as shown below. This is called a **dynamic programming approach**.

$$F(0) = 0$$

$$F(1) = 1$$

$$F(2) = F(1) + F(0)$$

$$F(3) = F(2) + F(1)$$

$$F(4) = F(3) + F(2)$$

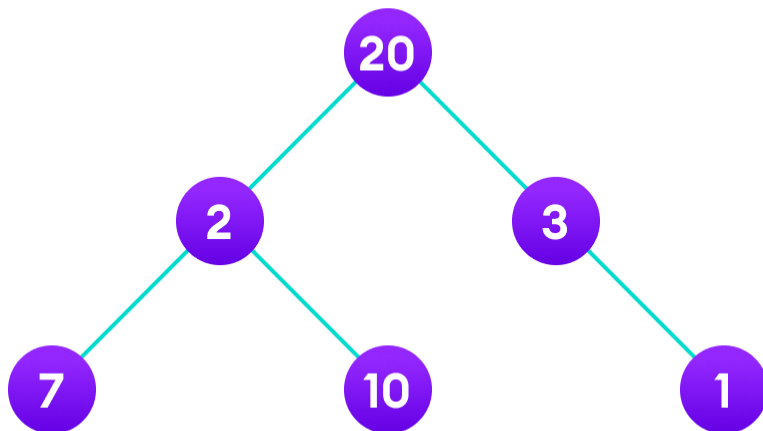
E. GREEDY ALGORITHMS

A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong. It works in a top-down approach.

This algorithm may not produce the best result for all the problems. It's because it always goes for the local best choice to produce the global best result.

For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.



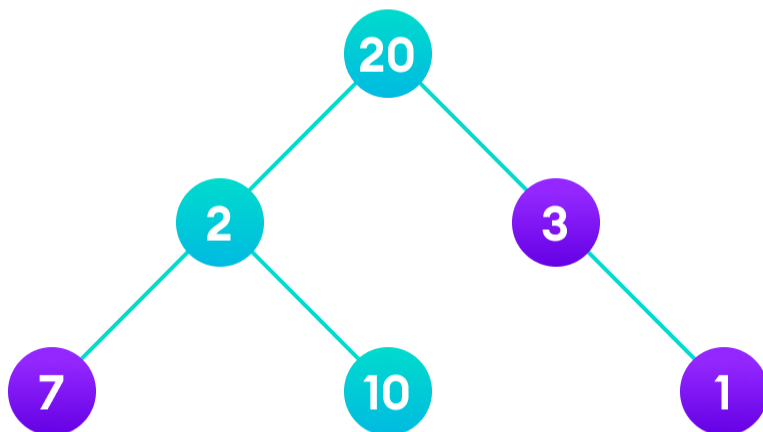
Greedy Approach

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.

2. Our problem is to find the largest path. And, the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.

3. Finally the weight of an only child of **3** is **1**. This gives us our final result $20 + 3 + 1 = 24$.

However, it is not the optimal solution. There is another path that carries more weight ($20 + 2 + 10 = 32$) as shown in the image below.

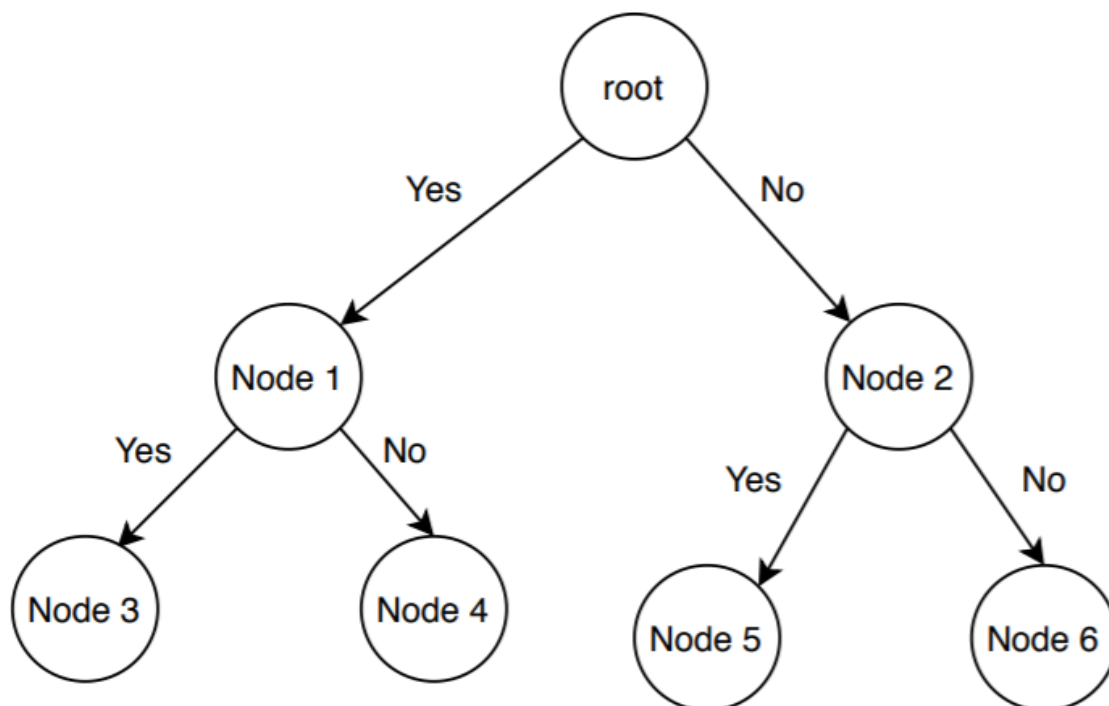


F. BRANCH AND BOUND ALGORITHMS

Branch and bound algorithms are generally used for optimization problems

- As the algorithm progresses, a tree of subproblems is formed
- The original problem is considered the “root problem”
- A method is used to construct an upper and lower bound for a given problem
- At each node, apply the bounding methods
 - If the bounds match, it is deemed a feasible solution to that particular subproblem
 - If bounds do *not* match, partition the problem represented by that node, and make the two subproblems into children nodes
- Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed

A branch and bound algorithm consist of stepwise enumeration of possible candidate solutions by exploring the entire search space. With all the possible solutions, we first build a rooted decision tree. The root node represents the entire search space:



Here, each child node is a partial solution and part of the solution set. Before constructing the rooted decision tree, we set an upper and lower bound for a given problem based on the optimal solution. At each level, we need to make a decision about which node to include in the solution set. At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.

G.BRUTE FORCE ALGORITHMS

A straightforward method of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

A Brute Force Algorithm is **the straightforward approach to a problem** i.e., the first approach that comes to our mind on seeing the problem

An example in computer science is the traveling salesman problem (TSP). Suppose a salesman needs to visit 10 cities across the country. How does one determine the order in which those cities should be visited such that the total distance traveled is minimized?

The brute force solution is simply to calculate the total distance for every possible route and then select the shortest one. This is not particularly efficient because it is possible to eliminate many possible routes through clever algorithms.

ALGORITHM ANALYSIS

Analysis of Algorithms is the area of computer science that provides tools to analyze the efficiency (time and space) of different methods of solutions. Algorithm is analyzed to discover its characteristics in order to evaluate its suitability for various applications

Time Complexity: Running time of a program as a function of the size of the input.

Space Complexity: Some forms of analysis could be done based on how much space an algorithm needs to complete its task.

Algorithm analysis techniques

1. Asymptotic Notations
2. Master theorem

1. Asymptotic Notations

Asymptotic analysis of an algorithm refers to defining the mathematical boundation /framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$. This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases.

Usually, the time required by an algorithm falls under three types –

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

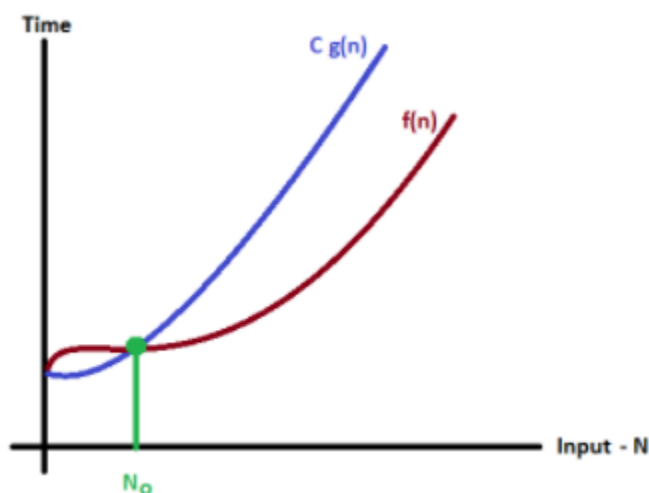
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation – worst case
- Ω Notation – best case
- θ Notation – average case

Big Oh Notation, O

The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



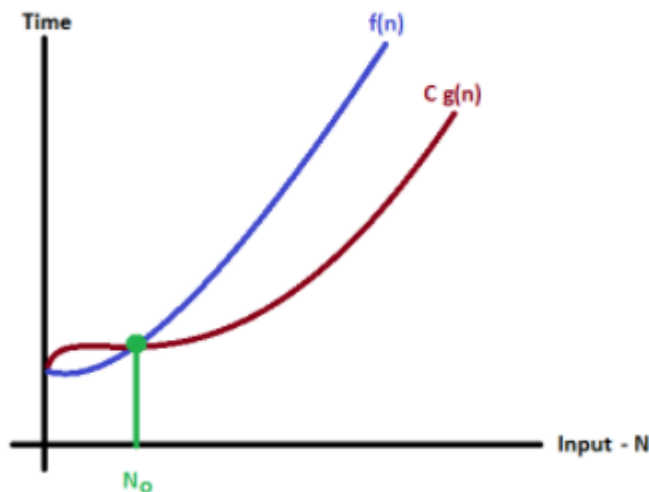
In above graph after a particular input value n_0 , always $C g(n)$ is greater than $f(n)$ which indicates the algorithm's upper bound.

NB: Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \leq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $O(g(n))$.

$$f(n) = O(g(n))$$

Omega Notation, Ω

The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



In above graph after a particular input value n_0 , always $C g(n)$ is less than $f(n)$ which indicates the algorithm's lower bound.

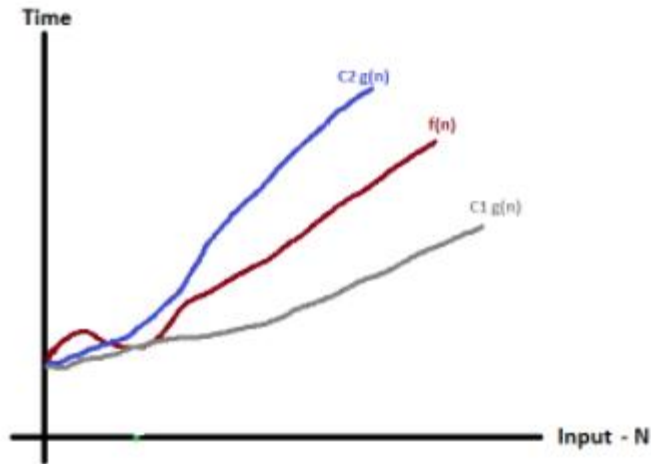
NB: Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \geq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Omega(g(n))$.

$$f(n) = \Omega(g(n))$$

Theta Notation, θ

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.



In above graph after a particular input value n_0 , always $C_1 g(n)$ is less than $f(n)$ and $C_2 g(n)$ is greater than $f(n)$ which indicates the algorithm's average bound.

NB: Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_0$, $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Theta(g(n))$.

$$f(n) = \Theta(g(n))$$

NB

O (log N) basically means time goes up linearly while the n goes up exponentially. So if it takes 1 second to compute 10 elements, it will take 2 seconds to compute 100 elements, 3 seconds to compute 1000 elements, and so on. It is $O(\log n)$ when we do divide and conquer type of algorithms (average case)

O(1) It means the running time of an algorithm is a constant. **It takes a constant time, like 14 nanoseconds**, or three minutes no matter the amount of data in the set. –(best case)

$O(n)$ denotes the **number of iterations, calculations or steps needed** at most (worst case), for the algorithm to reach its end-state, n being the objects given at the start of the algorithm

The master theorem

The master method is a formula for solving recurrence relations of the form:

$$T(n) = aT(n/b) + f(n),$$

where,

n = size of input

a = number of sub problems in the recursion

n/b = size of each sub problem. All sub problems are assumed to have the same size.

$f(n)$ = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Here, $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function.

An asymptotically positive function means that for a sufficiently large value of n , we have $f(n) > 0$.

The master theorem is used in calculating the time complexity of recurrence relations ([divide and conquer algorithms](#)) in a simple and quick way.

CASE 1

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

If the cost of solving the sub-problems at each level increases by a certain factor, the value of $f(n)$ will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level ie. $n^{\log_b a}$

$$\text{If } f(n) < n^{\log_b a}$$

then

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

Example:-

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

Here:-

$$a = 4$$

$$b = 2$$

$$f(n) = n$$

$$\therefore n^{\log_b a} = n^{\log_2 4}$$

$$= n^2$$

$$\therefore f(n) < n^{\log_b a}$$

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

$$\boxed{T(n) = \Theta(n^2)}$$

CASE 2

If the cost of solving the sub-problem at each level is nearly equal, then the value of $f(n)$ will be $n^{\log_b a}$. Thus, the time complexity will be $f(n)$ times the total number of levels ie. $n^{\log_b a} * \log n$

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

If $f(n) = n^{\log_b a}$

then

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

Example:-

$$T(n) = 3T\left(\frac{2n}{6}\right) + n$$

$$T(n) = 3T\left(\frac{2n/2}{6/2}\right) + n$$

$$= 3T\left(\frac{n}{3}\right) + n$$

Here :-

$$a = 3$$

$$b = 3$$

$$f(n) = n$$

$$\therefore n^{\log_b a} = n^{\log_3 3}$$

$$= n$$

$$\therefore f(n) = n$$

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

$$\boxed{T(n) = \Theta(n \log n)}$$

CASE 3:

1. If the cost of solving the subproblems at each level decreases by a certain factor, the value of $f(n)$ will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of $f(n)$.

• **Case 3.** If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ (and $af(\frac{n}{b}) \leq cf(n)$ for some $c < 1$ for all n sufficiently large), then $T(n) = \Theta(f(n))$.

$$\text{If } f(n) > n^{\log_b a}$$

then

$$T(n) = \Theta(f(n))$$

Example:-

$$T(n) = 16T\left(\frac{n}{8}\right) + n^2$$

Here:-

$$a = 16$$

$$b = 8$$

$$f(n) = n^2$$

$$\begin{aligned} \therefore n^{\log_b a} &= n^{\log_8 16} \\ &= n^{4/3} \end{aligned}$$

$\frac{\log_2 16}{\log_2 8}$
 $= \frac{4}{3}$

$$f(n) > n^{4/3}$$

$$\boxed{T(n) = n^2}$$

TOPIC 3: SEARCH ALGORITHM

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

A search algorithm is an algorithm which solves a search problem. Search algorithms work to retrieve information stored within some data.

There are two prominent search strategies are extensively used to find a specific item on a list

- i. Linear Search
- ii. Binary Search

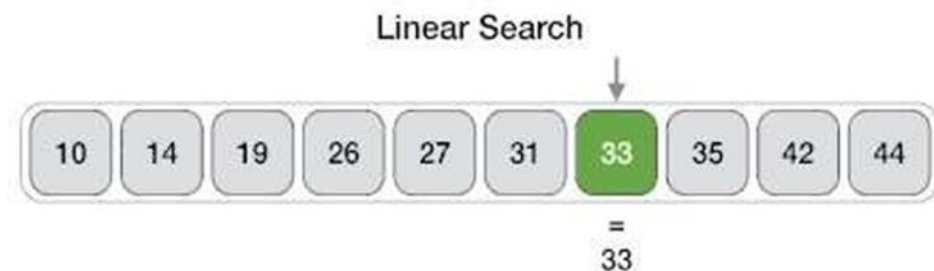
1. Linear Search

In linear search a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

It examines each element until it finds a match, starting at the beginning of the data set, until the end. The search is finished and terminated once the target element is located. If it finds no match, the algorithm must terminate its execution and return an appropriate result. The linear search algorithm is easy to implement and efficient in two scenarios:

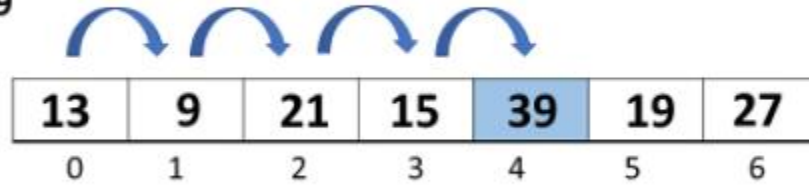
- When the list contains lesser elements
- When searching for a single element in an unordered array

Linear search is mostly used to search an unordered list in which the items are not sorted



Searched Element

39



Algorithm

Linear Search (Array Arr, Value a) // Arr is the name of the array, and a is the searched element.

Step 1: Set i to 0 // i is the index of an array which starts from 0

Step 2: if $i > n$ then go to step 7 // n is the number of elements in array

Step 3: if $Arr[i] = a$ then go to step 6

Step 4: Set i to $i + 1$

Step 5: Goto step 2

Step 6: Print element a found at index i and go to step 8

Step 7: Print element not found

Example of Linear Search Algorithm

Consider an array of size 7 with elements 13, 9, 21, 15, 39, 19, and 27 that starts with 0 and ends with size minus one, 6.

Search element = 39

13	9	21	15	39	19	27
0	1	2	3	4	5	6

Step 1: The searched element 39 is compared to the first element of an array, which is 13.

39						
13	9	21	15	39	19	27
0	1	2	3	4	5	6

The match is not found; you now move on to the next element and try to implement a comparison.

Step 2: Now, search element 39 is compared to the second element of an array, 9.

39						
13	9	21	15	39	19	27
0	1	2	3	4	5	6

As both are not matching, you will continue the search.

Step 3: Now, search element 39 is compared with the third element, which is 21.

39						
13	9	21	15	39	19	27
0	1	2	3	4	5	6

Again, both the elements are not matching, you move onto the next following element.

Step 4; Next, search element 39 is compared with the fourth element, which is 15.

39						
13	9	21	15	39	19	27
0	1	2	3	4	5	6

As both are not matching, you move on to the next element.

Step 5: Next, search element 39 is compared with the fifth element 39.

39						
13	9	21	15	39	19	27
0	1	2	3	4	5	6

A perfect match is found, you stop comparing any further elements and terminate the Linear Search Algorithm and display the element found at location 4.

Followed by the practical implementation, you will move on to the complexity of the linear search algorithm.

The Complexity of Linear Search Algorithm

You have three different complexities faced while performing Linear Search Algorithm, they are mentioned as follows.

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

Best Case Complexity

- The element being searched could be found in the first position.
- In this case, the search ends with a single successful comparison.

- Thus, in the best-case scenario, the linear search algorithm performs $O(1)$ operation.

Worst Case Complexity

- The element being searched may be at the last position in the array or not at all.
- In the first case, the search succeeds in 'n' comparisons.
- In the next case, the search fails after 'n' comparisons.
- Thus, in the worst-case scenario, the linear search algorithm performs $O(n)$ operations.

Average Case Complexity

When the element to be searched is in the middle of the array, the average case of the Linear Search Algorithm is $O(n)$.

Space Complexity of Linear Search Algorithm

The linear search algorithm takes up no extra space; its space complexity is $O(1)$ for an array of n elements.

A linear complexity— $O(n)$ denotes the **number of iterations, calculations or steps needed** at most (worst case), for the algorithm to reach its end-state, n being the objects given at the start of the algorithm. If the array has 10 items, the function calls `System.out.println` 10 times. If it has 1,000 items, it calls it 1,000 times.

Application of Linear Search Algorithm

The linear search algorithm has the following applications:

- Linear search can be applied to both single-dimensional and multi-dimensional arrays.
- Linear search is easy to implement and effective when the array contains only a few elements.
- Linear Search is also efficient when the search is performed to fetch a single search in an unordered-List.

2. Binary Search

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

How Binary Search Works?

Consider this list of ordered numbers:

Position in data set	0	1	2	3	4	5	6	7	8
Data value	1	3	4	5	7	9	11	14	16

Consider searching for the value 11.

The midpoint is found by adding the lowest position to the highest position and dividing by 2.

$$\text{Highest position (8) + lowest position (0) = 8}$$

$$8/2 = 4$$

NOTE - if the answer is a decimal, round up. For example, 3.5 becomes 4. An alternative is to round down, but be consistent.

Check at position 4, which has the value 7.

7 is less than 11, so the bottom half of the list - including the midpoint - is discarded.

Position in data set	0	1	2	3	4	5	6	7	8
Data value	1	3	4	5	7	9	11	14	16

The new lowest position is 5.

$$\text{Highest position (8) + lowest position (5) = 13}$$

$13/2 = 6.5$, which rounds up to 7

Check at position 7, which has the value 14.

14 is greater than 11, so the top half of the list (including the midpoint) is discarded.

Position in data set	0	1	2	3	4	5	6	7	8
Data value	1	3	4	5	7	9	11	14	16

The new highest position is 6.

Highest position (6) + lowest position (5) = 11

$11/2 = 5.5$, which rounds up to 6

Check at position 6.

The value held at position 6 is 11 - a match. The search ends.

Binary Search complexity

Time Complexity

Best Case Complexity - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **$O(1)$**

Worst Case Complexity - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **$O(\log n)$** .

Binary Search Algorithm Advantages-

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.

TOPIC 4: SORTING

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios:

- **Telephone Directory** – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

In-place Sorting VS Not-in-place Sorting

Sorting algorithms that **do not requires extra space for comparison** are known as in **place sorting** while Sorting algorithms that requires extra space for comparison are known as in out of place sorting.

Adaptive and Non-Adaptive Sorting Algorithm

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.

Typical complexities of algorithm

A logarithmic complexity- $O(\log N)$ basically means time goes up linearly while the n goes up exponentially. To perform operation of N elements, it often takes the logarithmic base as 2. So if it takes 1 second to compute 10 elements, it will take 2 seconds to compute 100 elements, 3 seconds to compute 1000 elements, and so on. It is $O(\log n)$ when we do divide and conquer type of algorithms (average case),

Logarithmic time complexities usually apply to algorithms that divide problems in half every time

Constant Complexity - $O(1)$ It means the running time of an algorithm is a constant. **It takes a constant time, like 14 nanoseconds**, or three minutes no matter the amount of data in the set. – (best case)

It describes algorithms that take the same amount of time to compute regardless of the input size.

For instance, if a function takes the same time to process ten elements and 1 million items, then we say that it has a constant growth rate or $O(1)$

A linear complexity— $O(n)$ denotes the number of iterations, calculations or steps needed at most (worst case), for the algorithm to reach its end-state, n being the objects given at the start of the algorithm. If the array has 10 items, the function calls `System.out.println` 10 times. If it has 1,000 items, it calls it 1,000 times.

Linear time complexity $O(n)$ means that the algorithms take proportionally longer to complete as the input grows.

Examples of linear time algorithms:

- Get the max/min value in an array.
- Find a given element in a collection.

$O(n^2)$ - Quadratic Complexity - If our array has n items, our outer loop runs n times and our inner loop runs n times *for each iteration of the outer loop*, giving us n^2 total calls to `System.out.println`. Thus this function runs in $O(n^2)$ time (or *quadratic time*). If the array has 10 items, we have to print 100 times. If it has 1,000 items, we have to print 1,000,000 times.

A function with a quadratic time complexity has a growth rate of n^2 . If the input is size 2, it will do four operations. If the input is size 8, it will take 64, and so on.

examples of quadratic algorithms:

- Check if a collection has duplicated values.
- Sorting items in a collection using bubble sort, insertion sort, or selection sort.

$O(n^3)$ - Cubic Complexity: For N inputs data size, it execute the order of N^3 steps on N element to solve a given problem

For example, if there exist 100 elements, it is going to execute 1000000 steps

$O(n \log n)$ - Linearithmic Complexity: The [Merge sort](#) worst case complexity is $O(n \log n)$.

Linearithmic time complexity it's slightly slower than a linear algorithm. It undergoes the execution of the order $N \cdot \log(N)$ on N number of elements to solve the given problem

For a given 1000 elements, the linear complexity will execute 10000 steps for solving a given problem

$O(2^n)$ - Exponential Complexity: The algorithm takes twice as long for every new element added, so even small increases in n dramatically increase the running time.

Exponential (base 2) running time means that the calculations performed by an algorithm double every time as the input grows.

Examples of exponential runtime algorithms:

- Power Set: finding all the subsets on a set.
- Fibonacci.

Types of Sort Algorithm

a. BUBBLE SORT 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?

- Bubble sort uses multiple passes (scans) through an array.
- In each pass, bubble sort compares the adjacent elements of the array.
- It then swaps the two elements if they are in the wrong order.
- In each pass, bubble sort places the next largest element to its proper position.
- In short, it bubbles down the largest element to its correct position.

Algorithm of Bubble sort

```
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
```

Example

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



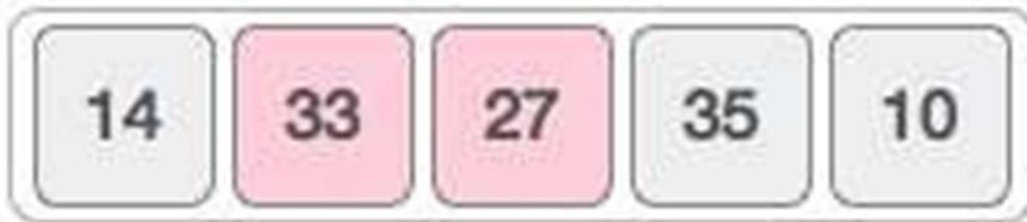
Bubble sort starts with very first two elements, comparing them to check which one is greater.



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 –



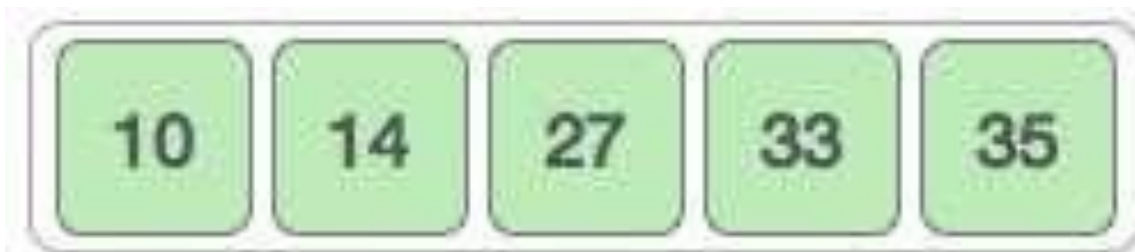
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –



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



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



Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: $O(n^2)$ - element to compare with all the other elements $n*n$
- Best Case Time Complexity [Big-omega]: $O(n)$ -
- Average Time Complexity [Big-theta]: $O(n^2)$

Space Complexity Analysis-

Bubble sort uses only a constant amount of extra space

Hence, the space complexity of bubble sort is $O(1)$.

It is an in-place sorting algorithm i.e. it modifies elements of the original array to sort the given array.

b. INSERTION SORT

Insertion sort is a sorting algorithm in which the elements are transferred one at a time to the **right position**. In other words, an insertion sort helps in building the final sorted list, one item at a time, with the movement of higher-ranked elements. 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 Insertion Sort Works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub- list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Step 1- Start

Step 2 – If it is the first element, it is already sorted.;

Step 3 – Pick next element

Step 4 – Compare with all elements in the sorted sub-list

Step 5 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 6 – Insert the value

Step 7 – Repeat until list is sorted

Step 8 – Stop

Time Complexity Analysis:

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n .

Wherein for an unsorted array, it takes for an **element to compare with all the other elements** which mean every n element compared with all other n elements. Thus, making it for $n \times n$, i.e., n^2 comparisons.

- **Worst Case Time Complexity [Big-O]:** $O(n^2)$
- **Best Case Time Complexity [Big-omega]:** $O(n)$
- **Average Time Complexity [Big-theta]:** $O(n^2)$

Space Complexity

Constant Complexity - $O(1)$ It takes a constant space, no matter the elements in that array

3. Selection Sorting

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two sub arrays in a given array.

- 1) The sub array which is already sorted.
- 2) Remaining sub array which is unsorted.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where n is the number of items.

EXAMPLE

Consider the following depicted array as an example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



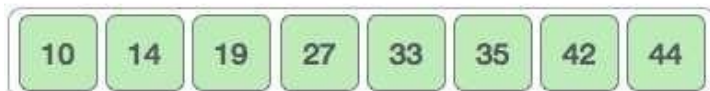
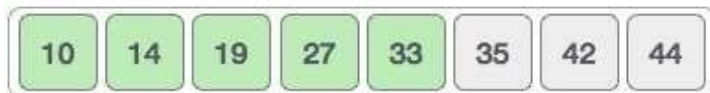
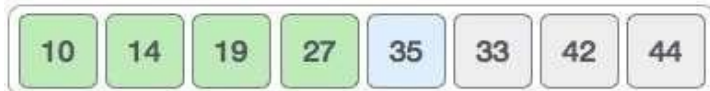
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array. Following is a pictorial depiction of the entire sorting process –



Algorithm

- Step 1** – Set MIN to location 0
- Step 2** – Search the minimum element in the list
- Step 3** – Swap with value at location MIN
- Step 4** – Increment MIN to point to next element
- Step 5** – Repeat until list is sorted

Time Complexity

- **Worst Case Time Complexity [Big-O]: $O(n^2)$** - The worst case is the case when the array is already sorted (with one swap) but the smallest element is the last element. The cost in this case is that at each step, a swap is done. This is because the smallest element will always be the last element and the swapped element which is kept at the end will be the second smallest element that is the smallest element of the new unsorted sub-array. Hence, the worst case has:
 - $N * (N+1) / 2$ comparisons
 - N swaps
 - Hence, the time complexity is $O(N^2)$.
- **Best Case Time Complexity [Big-omega]: $O(n)$** - The best case is the case when the array is already sorted. For example, if the sorted number as a_1, a_2, \dots, a_N , then: $a_1, a_2, a_3, \dots, a_N$ will be the best case for our particular implementation of Selection Sort.
- **Average Time Complexity [Big-theta]: $O(n^2)$**

Space Complexity of Selection Sort

The space complexity of Selection Sort is $O(1)$.

In terms of Space Complexity, Selection Sort is optimal as the memory requirements remain same for every input.

4. Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

Example

To understand merge sort, we take an unsorted array as the following –



We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.



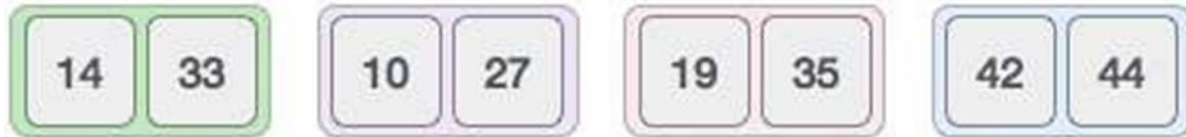
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



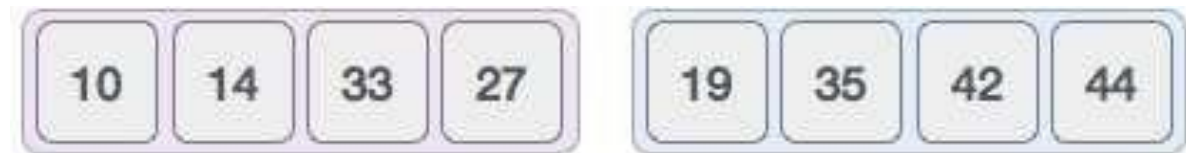
We further divide these arrays and we achieve atomic value which can no more be divided.



Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.



After the final merging, the list should look like this –



Algorithm

Step 1: Start

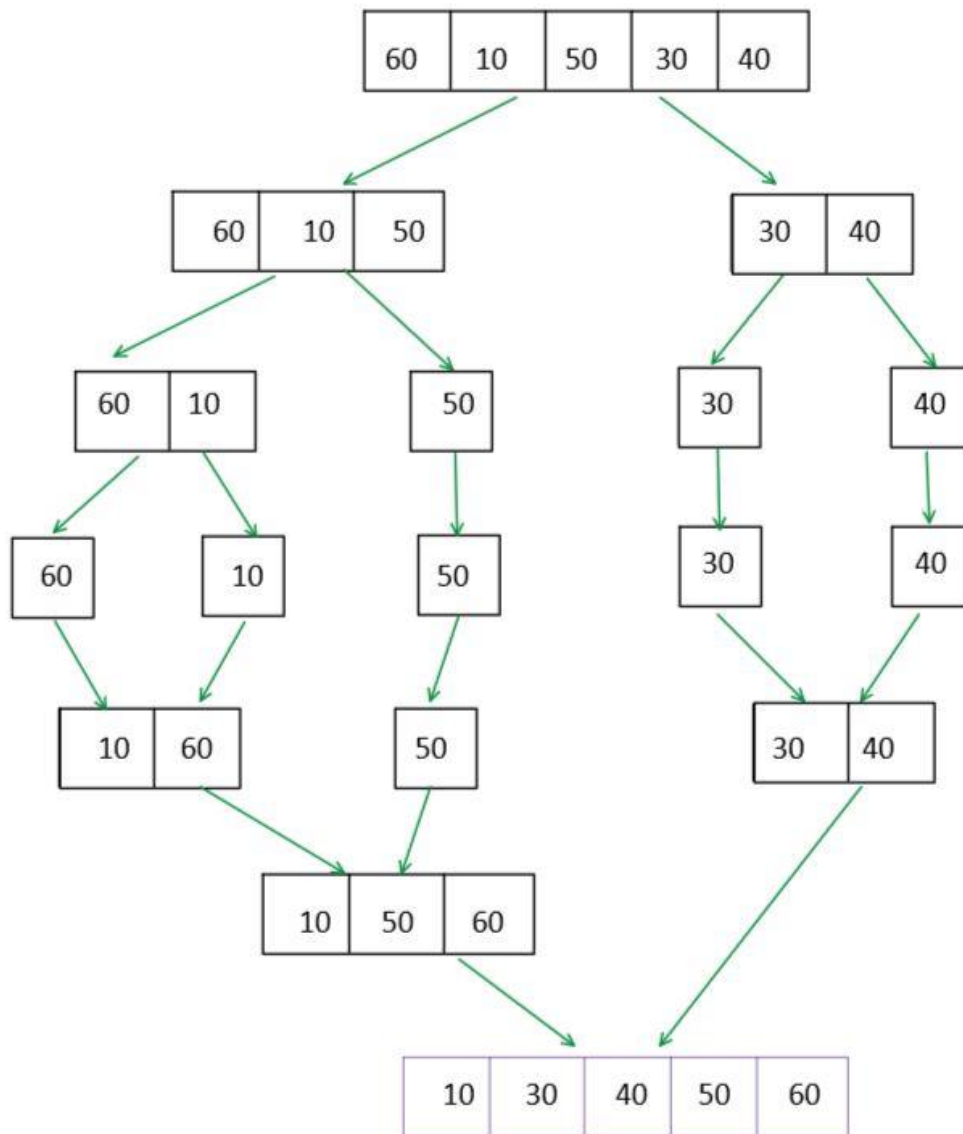
Step 2: – if it is only one element in the list it is already sorted, go to step 5.

Step 3 – divide the list recursively into two halves until it can no more be divided.

Step 4 – merge the smaller lists into new list in sorted order.

Step 5: stop

Example 2



Time Complexity

Time complexity of Merge Sort is $\theta(n \log n)$ in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

At each level merge sort has to perform N operation. The height of any tree is given by $\log N$.

Time complexity is given by operation at each level * the height of the tree

$$= n \log n$$

SPACE COMPLEXITY

It takes $O(N)$ space as we divide the array and store it into them where the total space consumed in making the entire array and merging back into one array is the total number of elements present in it.

5. QUICK SORT

Quicksort is another sorting algorithm which uses Divide and Conquer for its implementation.

Quicksort first chooses a pivot and then partition the array around this pivot. In the partitioning process, all the elements smaller than the pivot are put on one side of the pivot and all the elements larger than it on the other side.

Pivot element can be any element from the array; it can be the first element, the last element or any random element.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting sub arrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of $O(n \log n)$, where **n** is the number of items.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sub arrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

Quick Sort Pivot Algorithm

Step 1 – Choose the highest index value has pivot

Step 2 – Take two variables to point left and right of the list excluding pivot

Step 3 – left points to the low index

Step 4 – right points to the high

Step 5 – while value at left is less than pivot move right

Step 6 – while value at right is greater than pivot move left

Step 7 – if both step 5 and step 6 does not match swap left and right

Step 8 – if $\text{left} \geq \text{right}$, the point where they met is new pivot

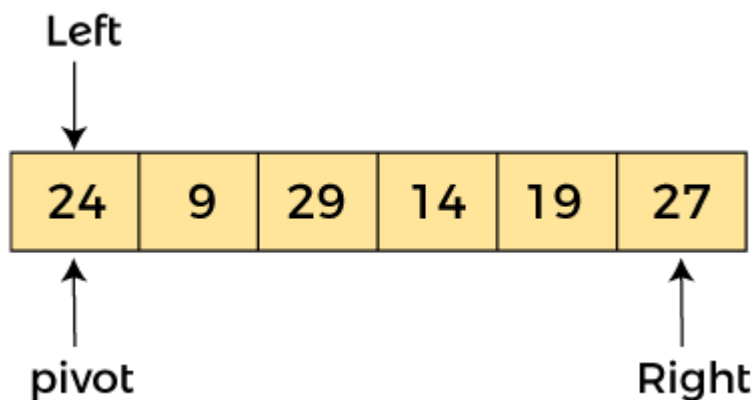
Example

Let the elements of array are:

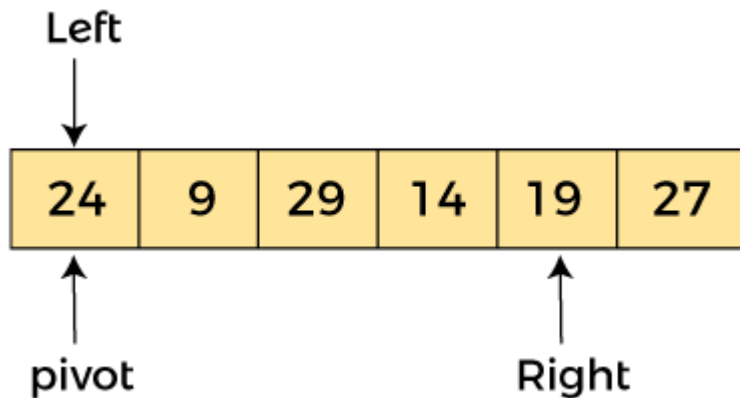
24	9	29	14	19	27
----	---	----	----	----	----

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

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

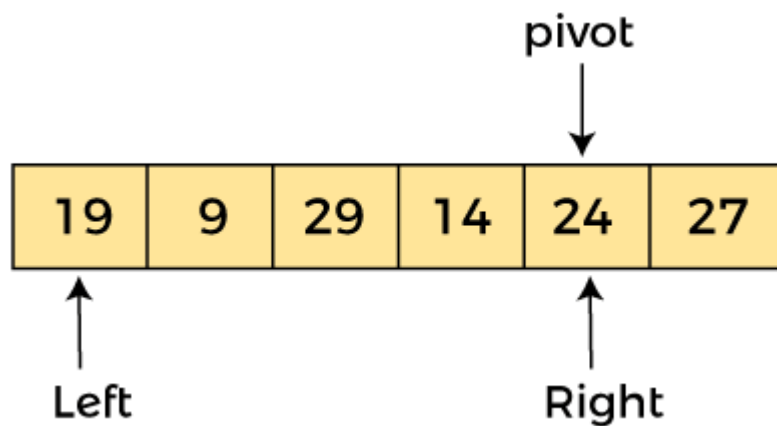


Now, $a[\text{pivot}] < a[\text{right}]$, so algorithm moves forward one position towards left, i.e. –



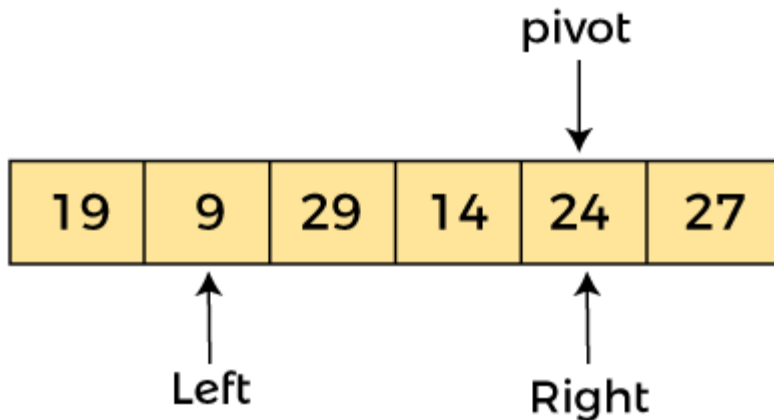
Now, $a[\text{left}] = 24$, $a[\text{right}] = 19$, and $a[\text{pivot}] = 24$.

Because, $a[\text{pivot}] > a[\text{right}]$, so, algorithm will swap $a[\text{pivot}]$ with $a[\text{right}]$, and pivot moves to right, as -

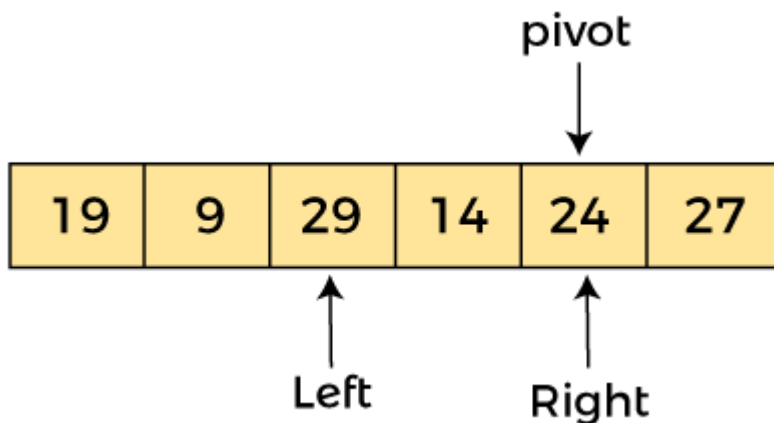


Now, $a[\text{left}] = 19$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. Since, pivot is at right, so algorithm starts from left and moves to right.

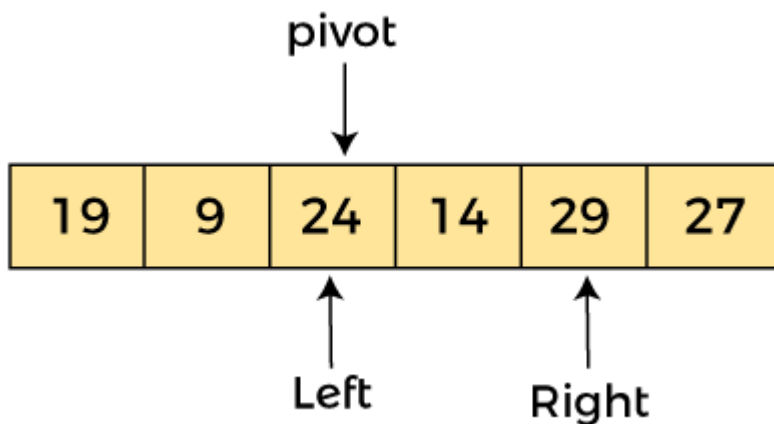
As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as –



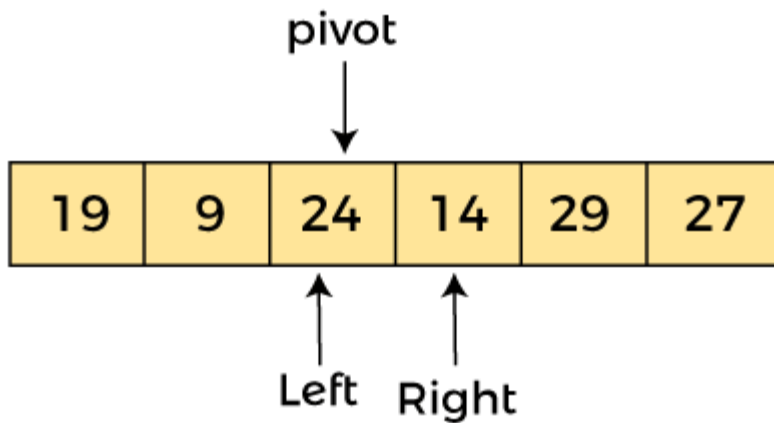
Now, $a[\text{left}] = 9$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] > a[\text{left}]$, so algorithm moves one position to right as –



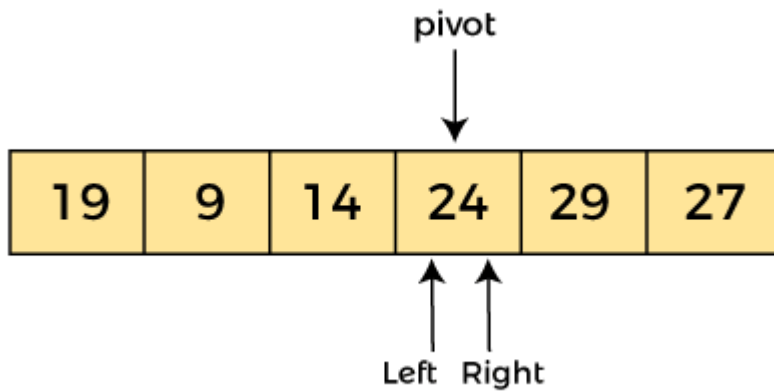
Now, $a[\text{left}] = 29$, $a[\text{right}] = 24$, and $a[\text{pivot}] = 24$. As $a[\text{pivot}] < a[\text{left}]$, so, swap $a[\text{pivot}]$ and $a[\text{left}]$, now pivot is at left, i.e. –



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



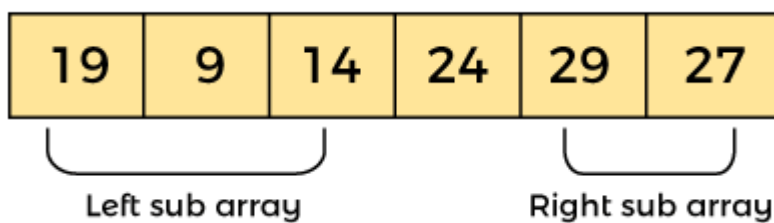
Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 14$, and $a[\text{right}] = 24$. Pivot is at right, so the algorithm starts from left and moves to right.



Now, $a[\text{pivot}] = 24$, $a[\text{left}] = 24$, and $a[\text{right}] = 24$. 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 –

9	14	19	24	27	29
---	----	----	----	----	----

Time Complexity

Case	Time Complexity
Best Case	$O(n \cdot \log n)$
Average Case	$O(n \cdot \log n)$
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 **$O(n \cdot \log n)$** .
- **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 **$O(n \cdot \log n)$** .
- **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 **$O(n^2)$** .

Space Complexity

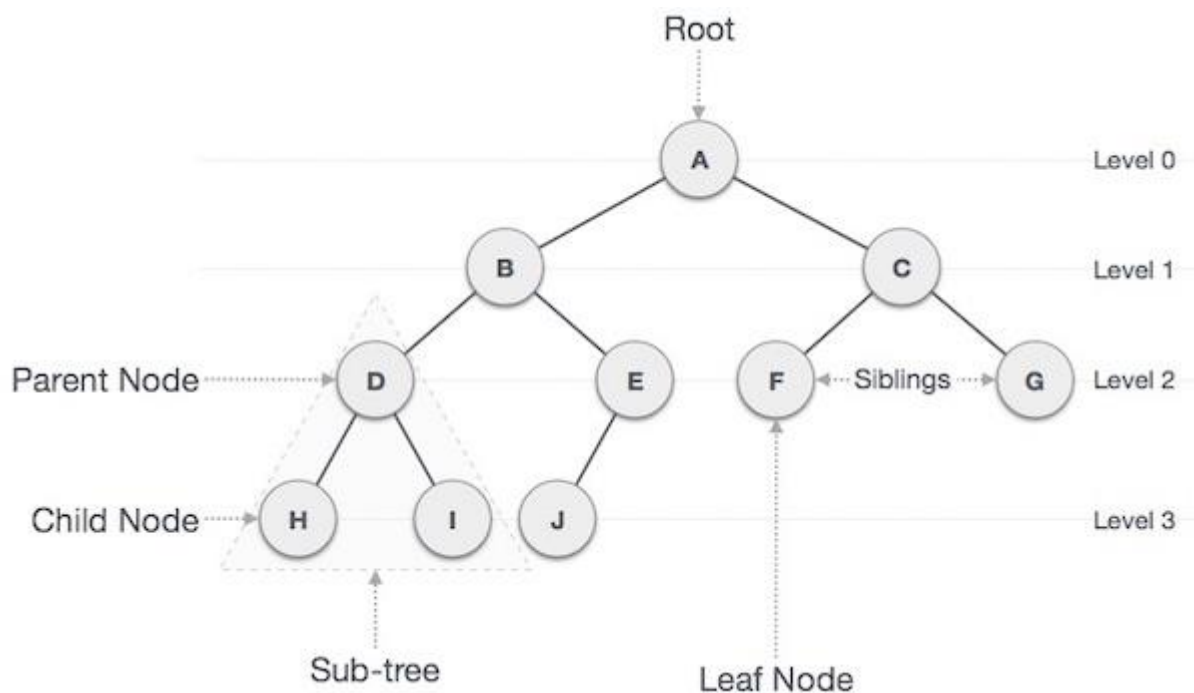
The space complexity of quicksort is $O(n \cdot \log n)$. - **Linearithmic Complexity**

TOPIC 5: TREES AND ALGORITHM

Tree represents the nodes connected by edges.

Binary tree or binary search tree specifically.

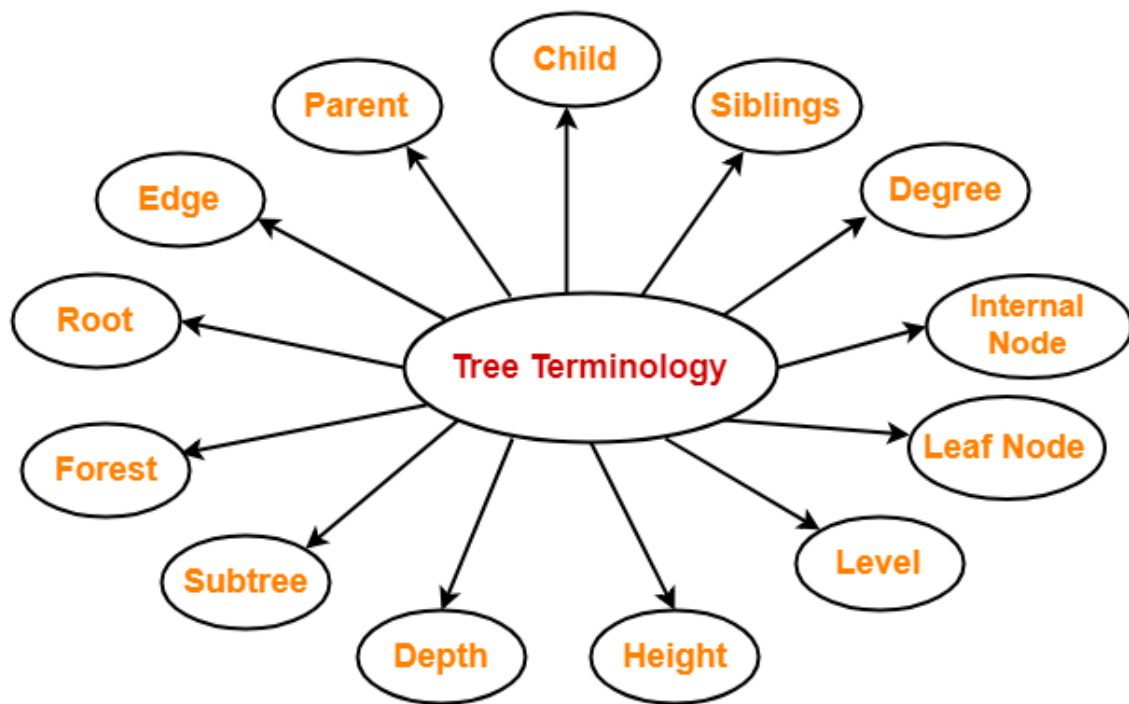
Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list



Important Terms

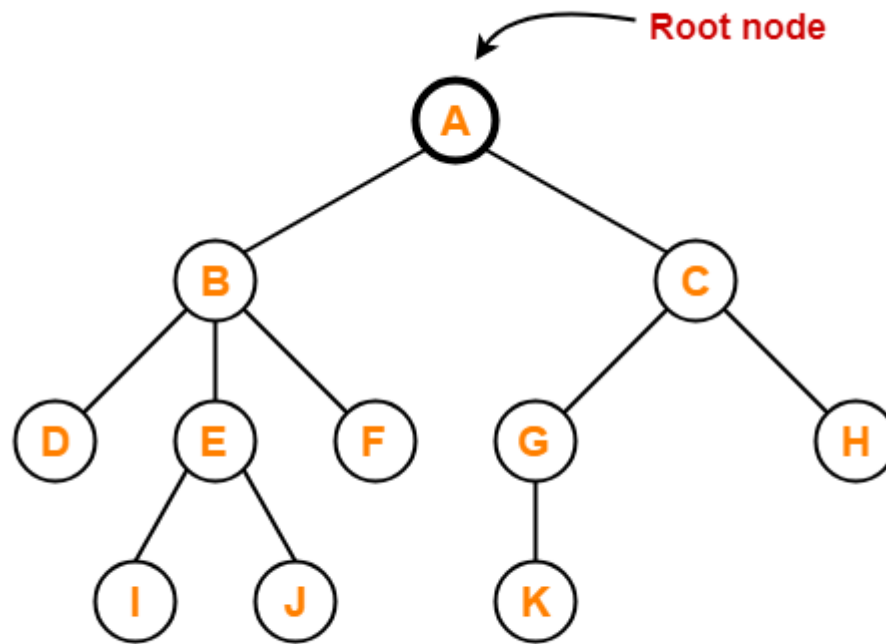
Tree Terminology-

The important terms related to tree data structure are-



1. Root-

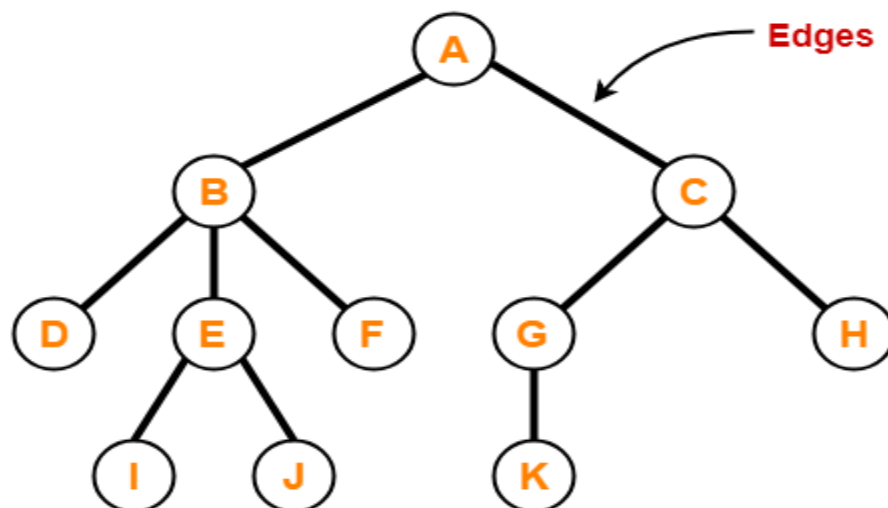
- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



Here, node A is the only root node.

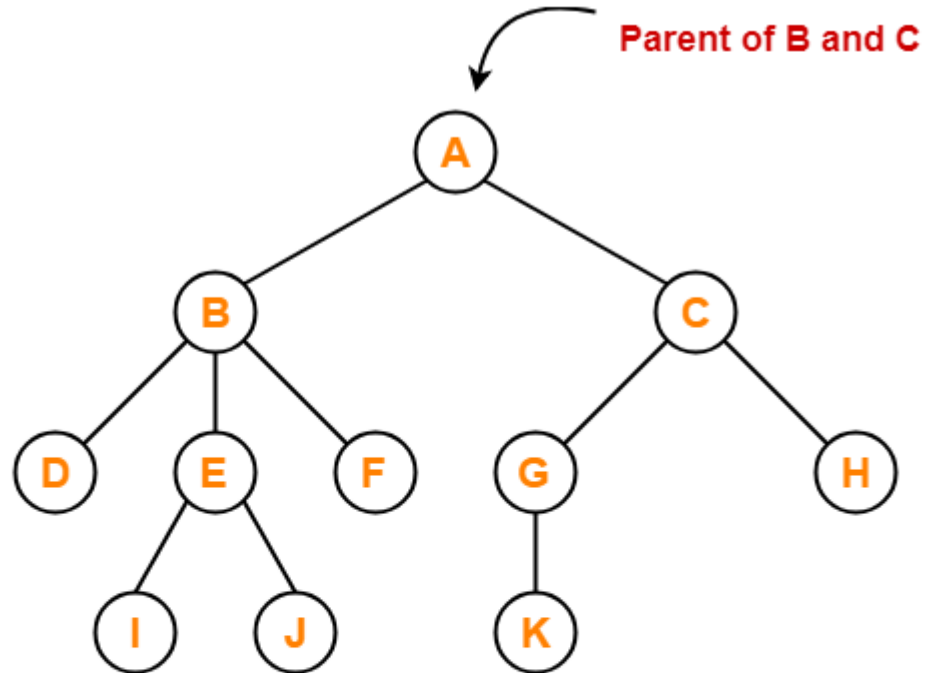
2. Edge-

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.



Parent-

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

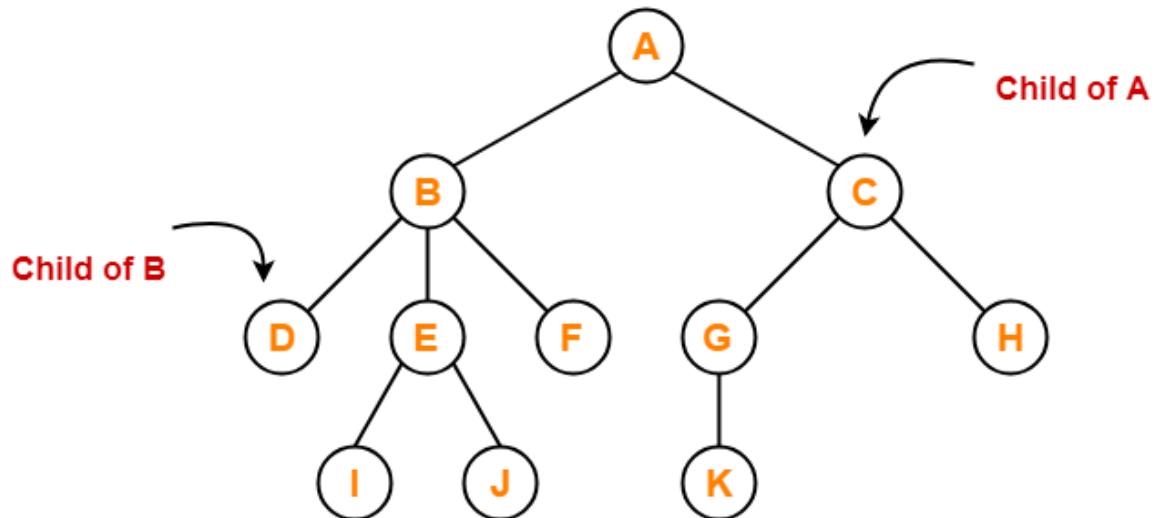


Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

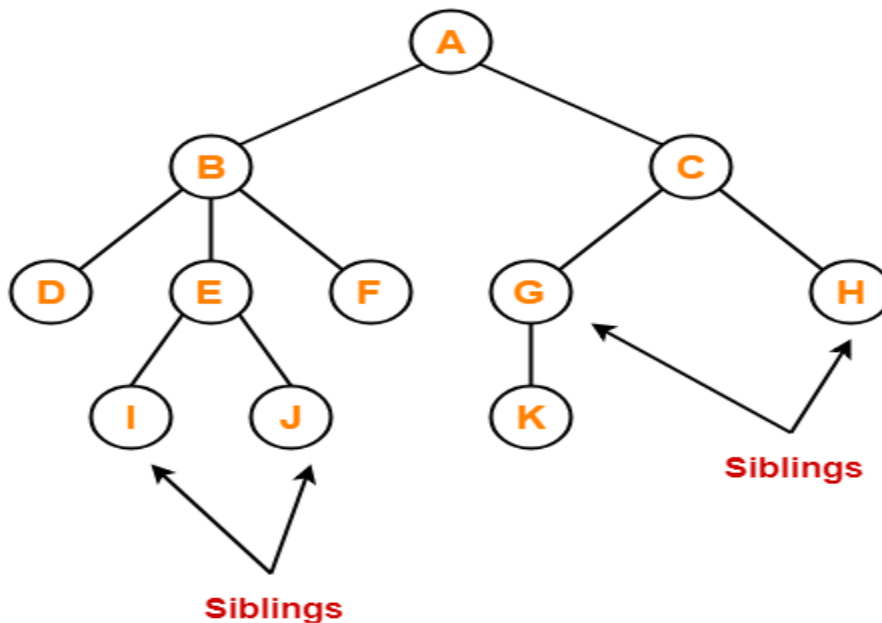
4. Child-

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.



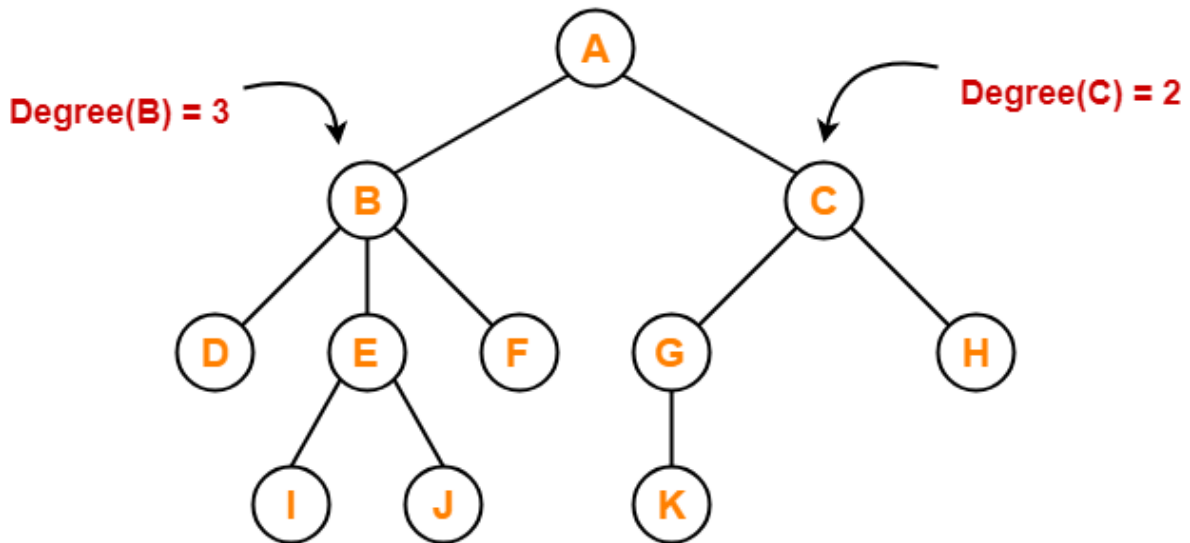
5. Siblings-

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.



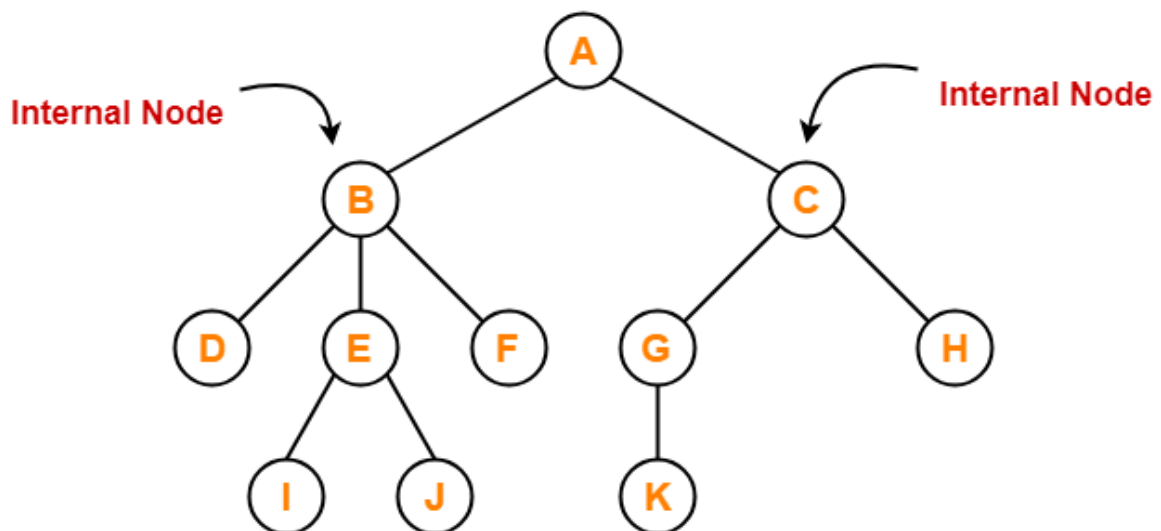
6. Degree-

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.



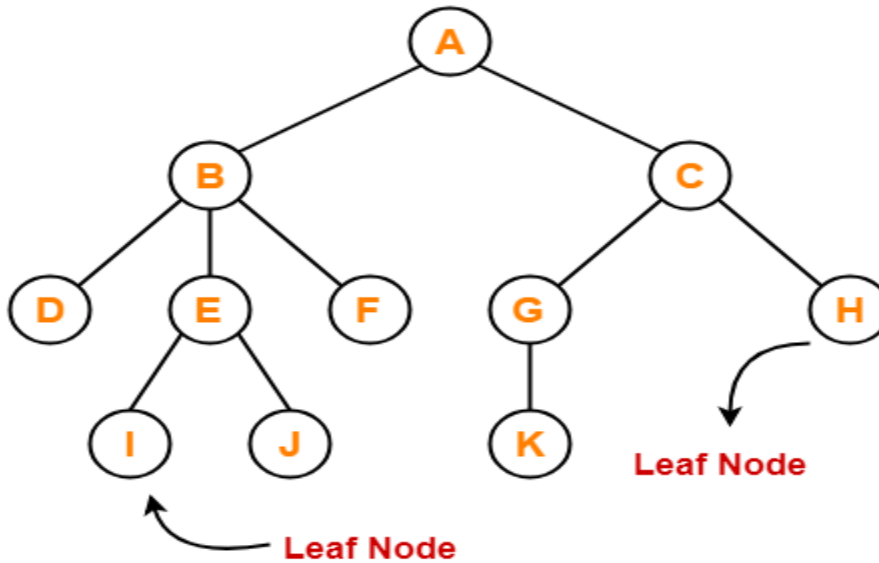
7. Internal Node-

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



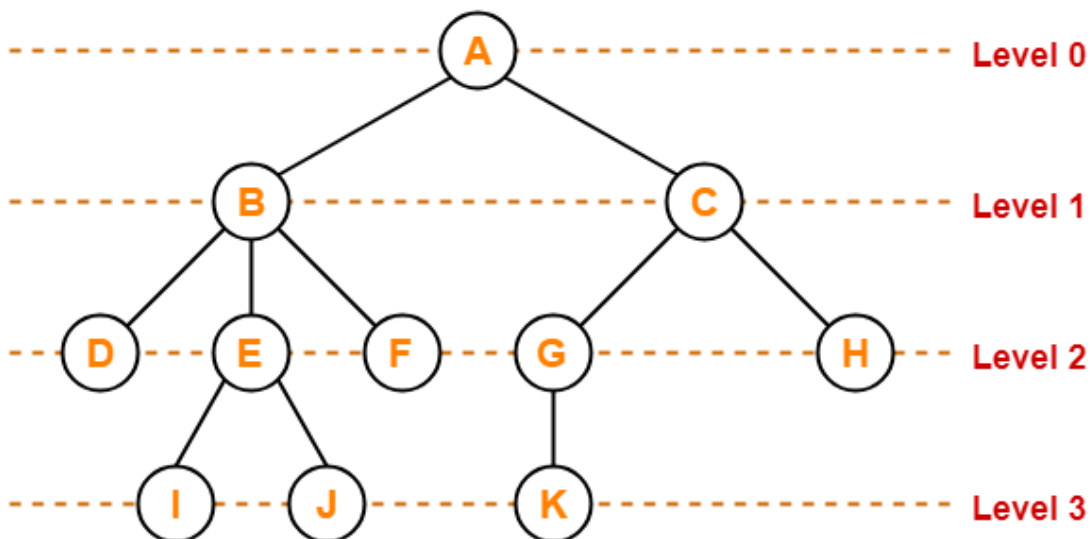
8. Leaf Node-

- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



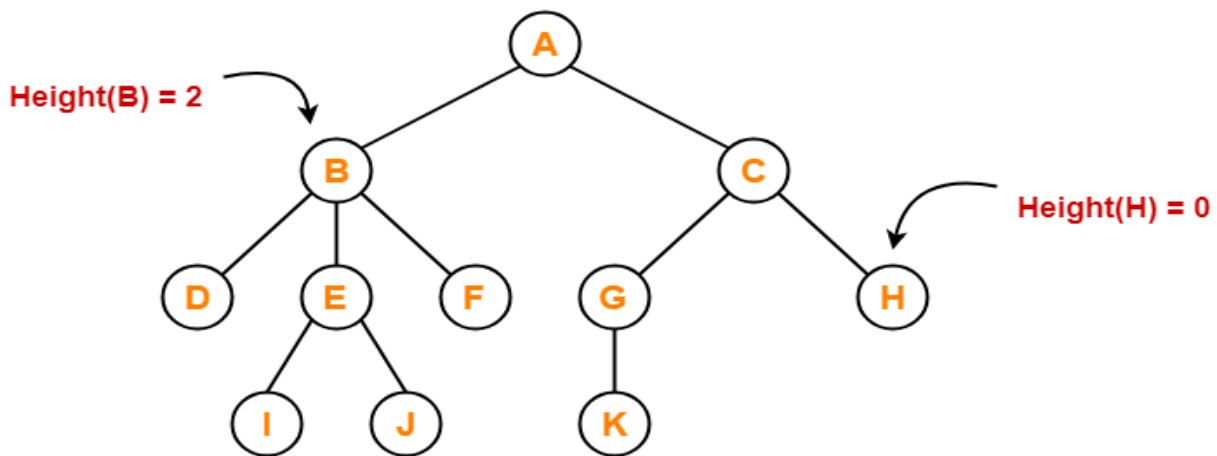
9. Level-

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



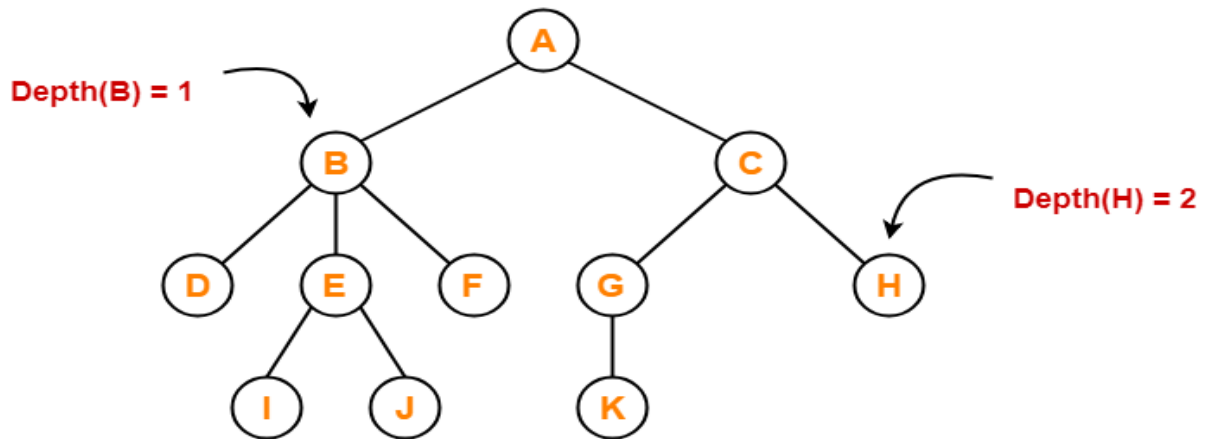
10. Height-

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0



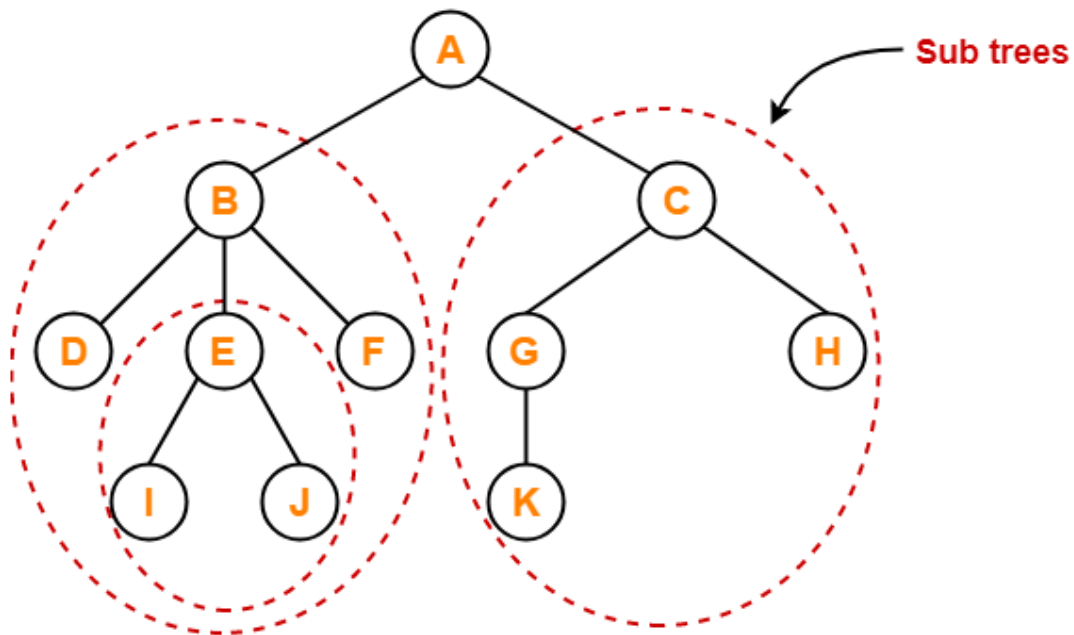
11. Depth-

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



12. Subtree-

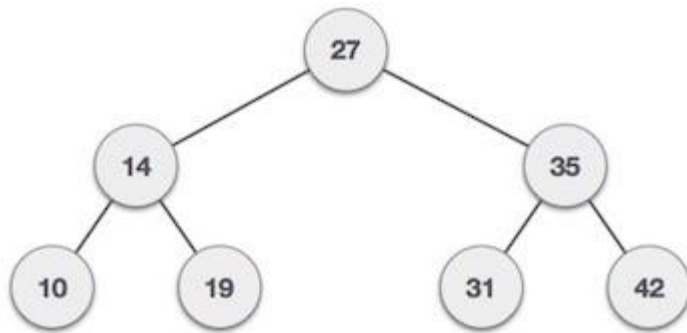
- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.



- **Huffman coding tree** or **Huffman tree** is a full binary **tree** in which each leaf of the **tree** corresponds to a letter in the given alphabet. Define the weighted path length of a leaf to be its weight times its depth.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following –

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Preorder Traversal** – Traverses a tree in a pre-order manner.
- **In order Traversal** – Traverses a tree in an in-order manner.
- **Post order Traversal** – Traverses a tree in a post-order manner.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left sub tree and insert the data. Otherwise, search for the empty location in the right sub tree and insert the data.

Algorithm

If root is NULL

then create root node

return

If root exists then

compare the data with node.data

while until insertion position is located

If data is greater than node.data

goto right subtree

else

```
        goto left subtree
    endwhile
    insert data
end If
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left sub tree. Otherwise, search for the element in the right sub tree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
    return root
else
    while data not found
        If data is greater than node.data
            goto right subtree
        else
            goto left subtree
        If data found
            return node
        endwhile
    return data not found
end if
```

Traversal

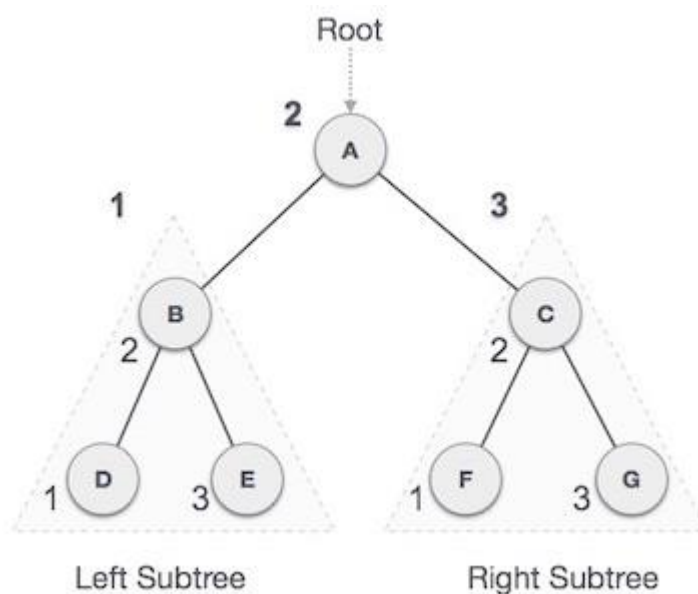
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

In-order Traversal

In this traversal method, the left sub tree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a sub tree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.



The output of inorder traversal of this tree will be –

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Algorithm

Until all nodes are traversed –

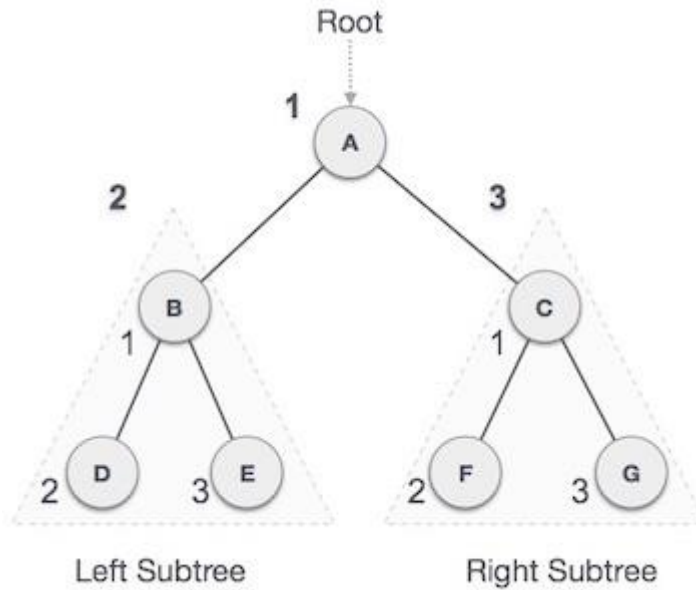
Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

Algorithm

Until all nodes are traversed –

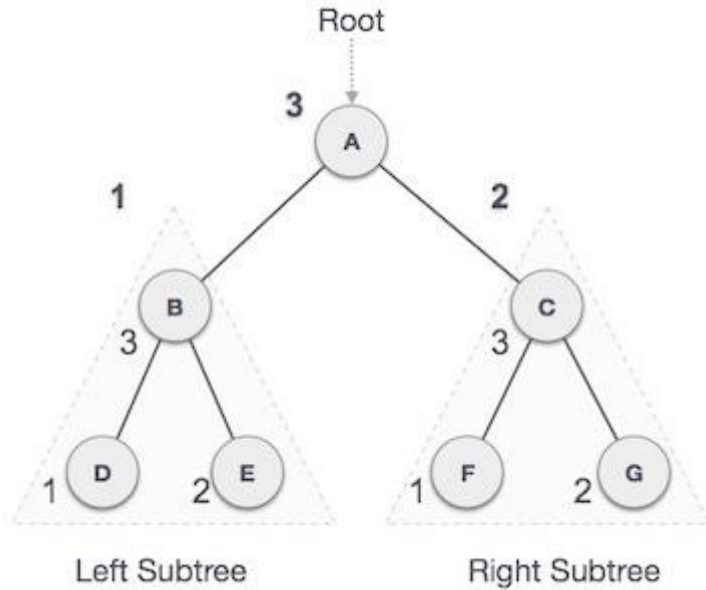
Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Algorithm

Until all nodes are traversed –

- Step 1** – *Recursively traverse left subtree.*
- Step 2** – *Recursively traverse right subtree.*
- Step 3** – *Visit root node.*

TOPIC 6: GRAPHS AND ALGORITHMS

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Graph Terminology

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path

A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0 = V_N$.

Simple Path

If all the nodes of the graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.

Cycle

A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Connected Graph

A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as $w(e)$ which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is associated with the similar end points can be called as Loop.

Adjacent Nodes

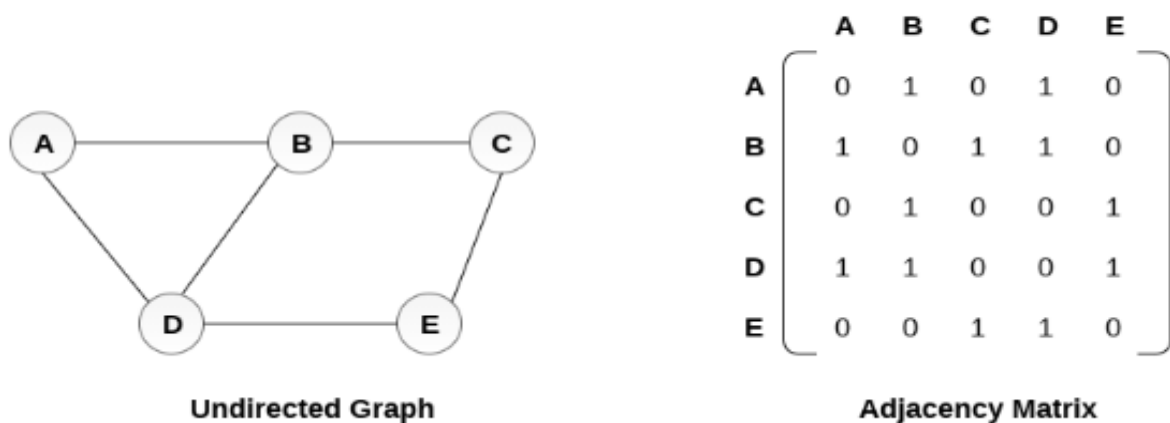
If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.

Degree of the Node

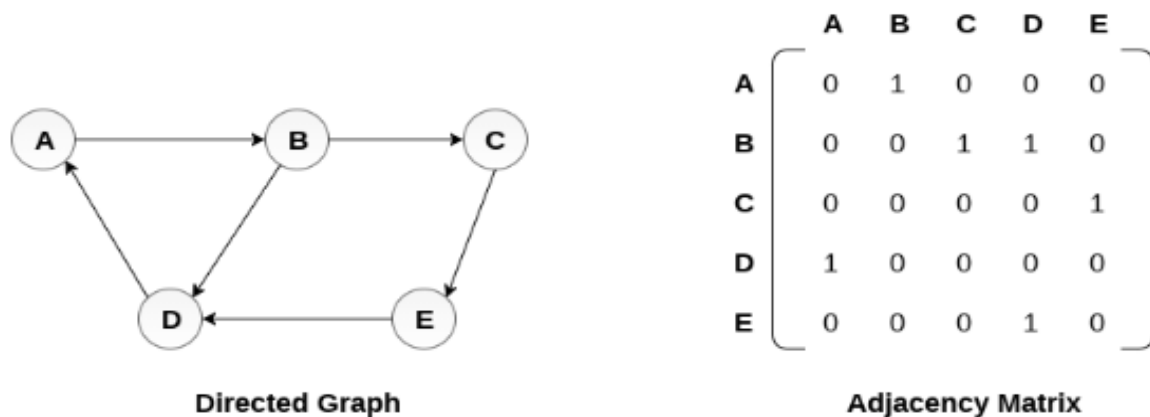
A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph representation

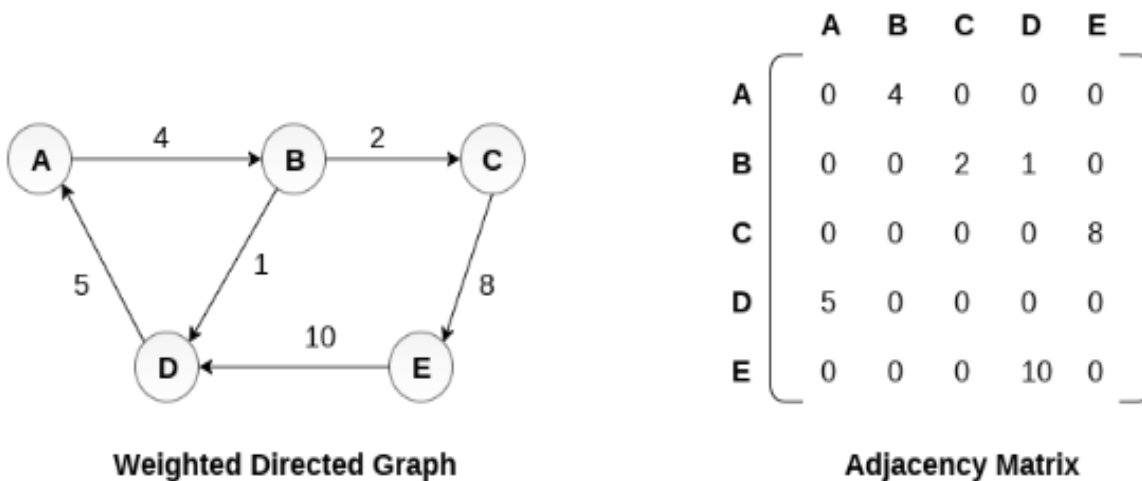
An undirected graph and its adjacency matrix representation is shown in the following figure.



A directed graph and its adjacency matrix representation is shown in the following figure.



The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



Graph traversal algorithms

Traversing the graph means examining all the nodes and vertices of the graph. Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them. There are two standard methods.

- I. Breadth First Search
- II. Depth First Search

Breadth First Search (BFS)

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

It employs the following rules.

- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

A standard BFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

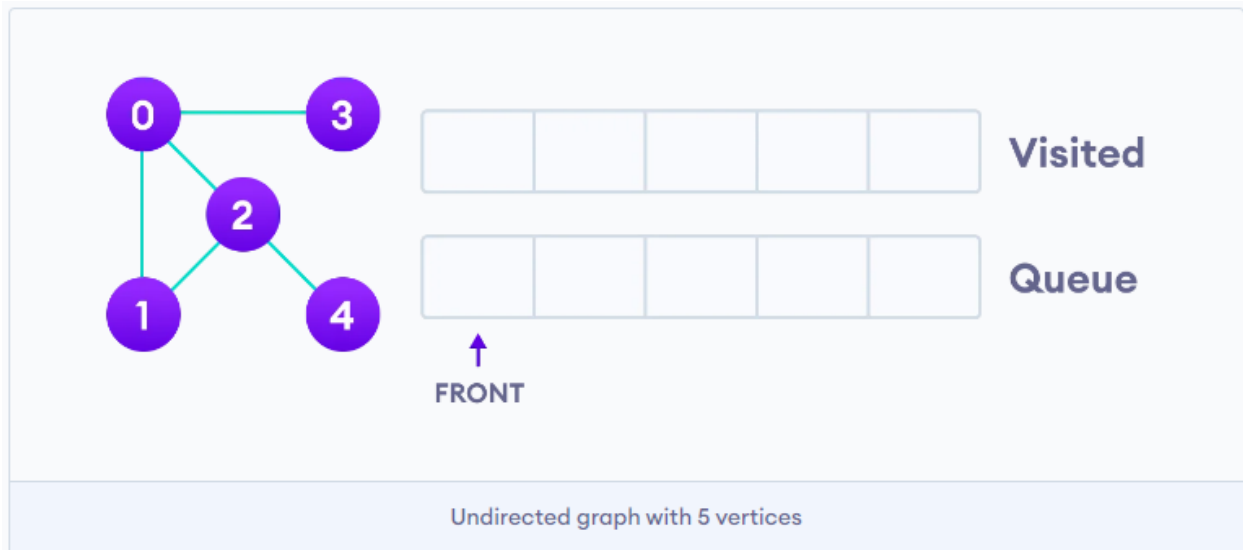
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The algorithm works as follows:

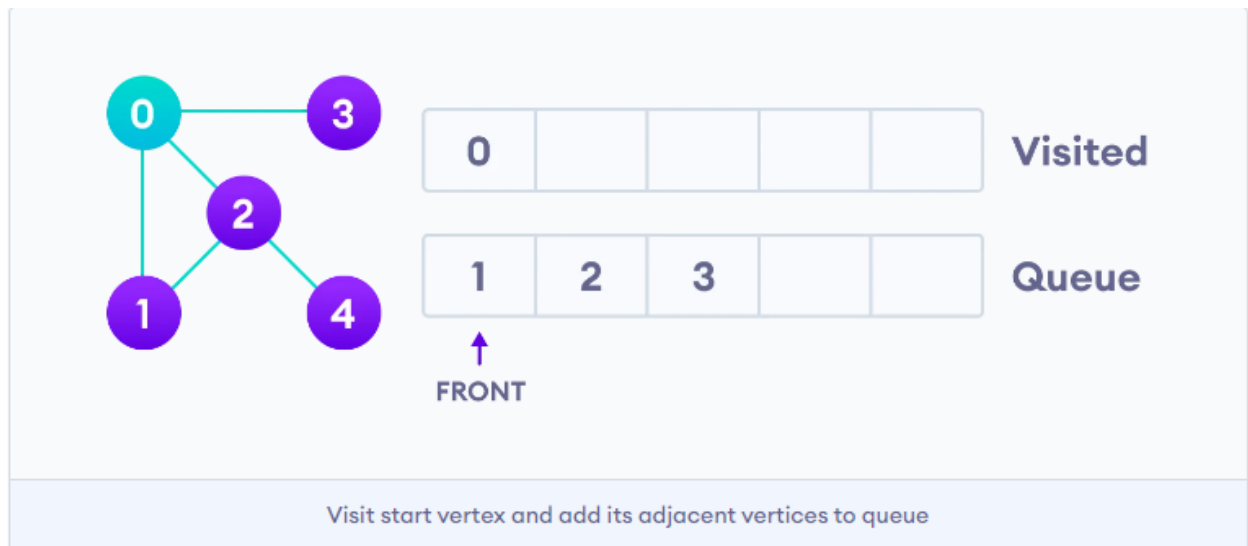
1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

Example

We use an undirected graph with 5 vertices.



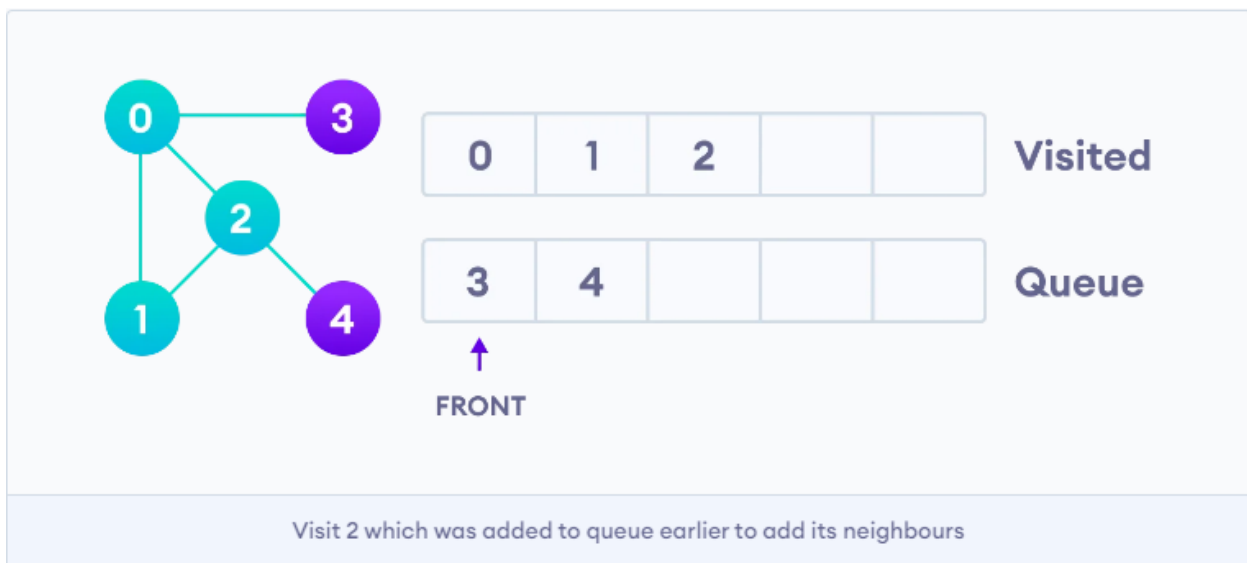
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

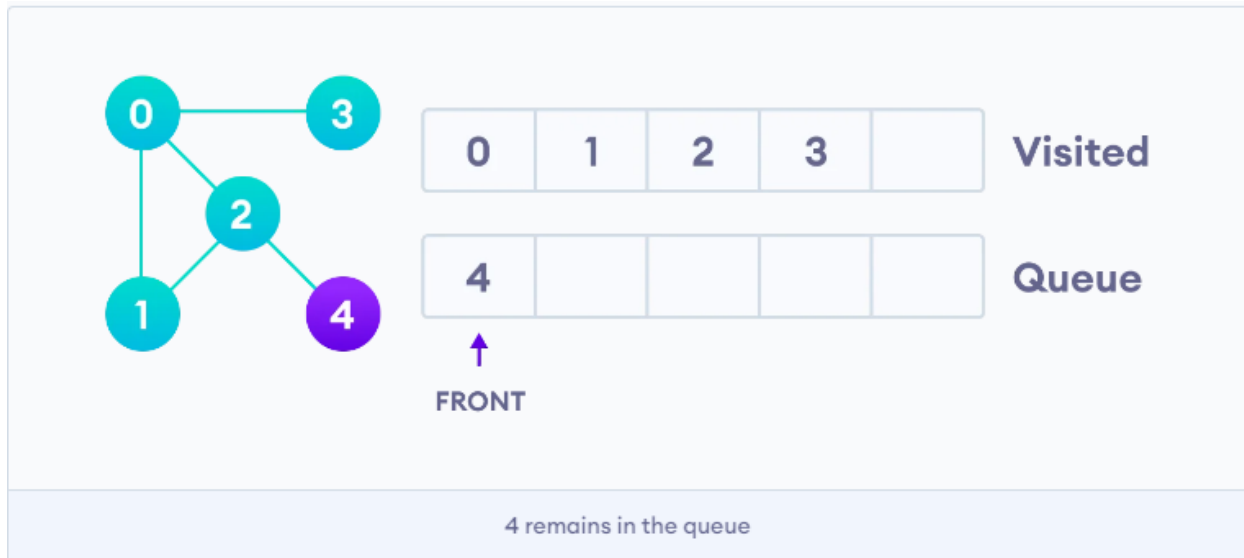


Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

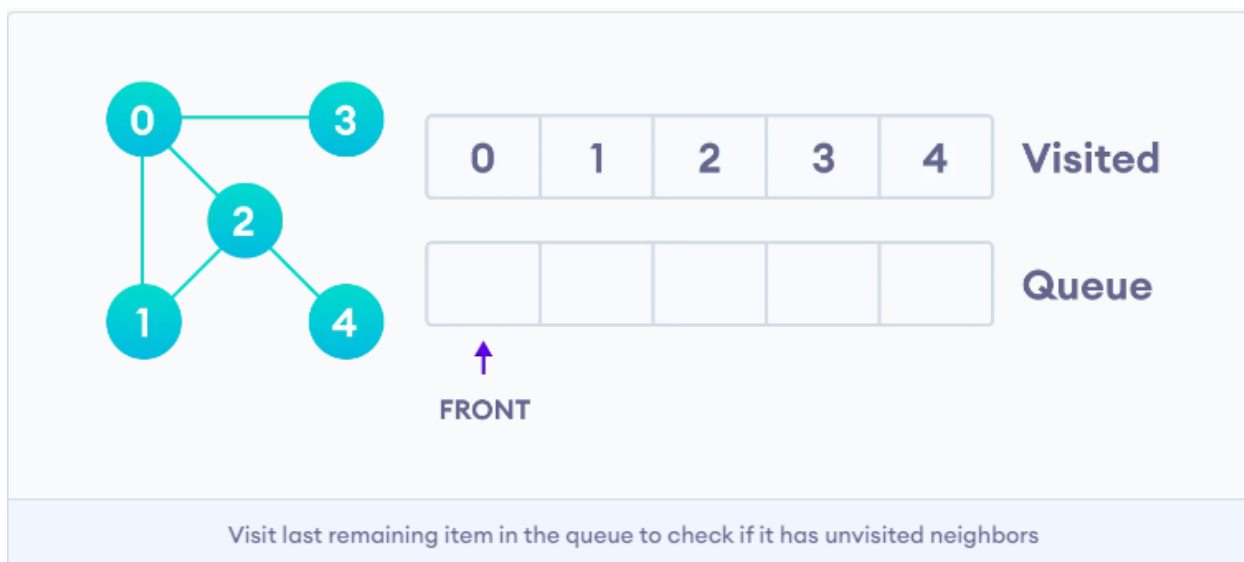


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.





Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



BFS Algorithm Complexity

The time complexity of the BFS algorithm is represented in the form of $O(V + E)$, where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is $O(V)$.

BFS Algorithm Applications

1. To build index by search index

2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph

Depth First Search (DFS)

The Depth First Search (DFS) is a graph traversal algorithm. In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner. It uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a [graph](#).

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

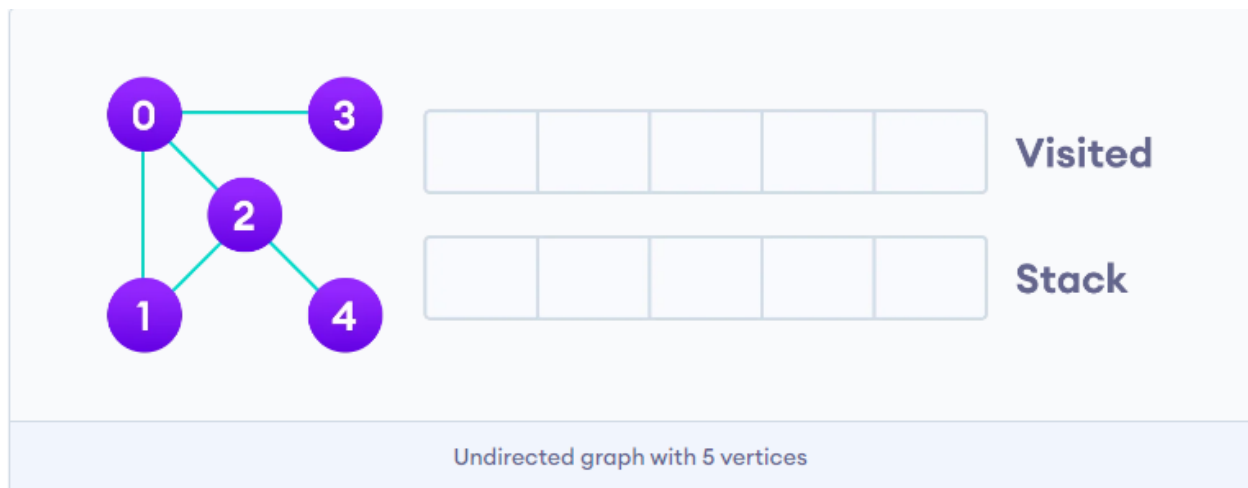
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

Depth First Search Example

We use an undirected graph with 5 vertices.



We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



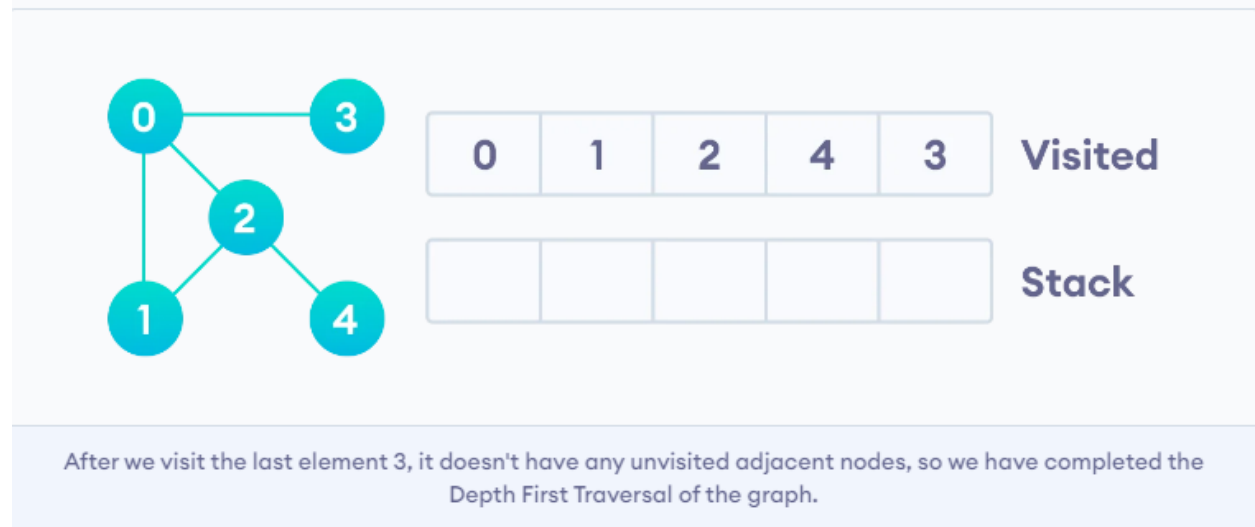
Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



Complexity of Depth First Search

The time complexity of the DFS algorithm is represented in the form of $O(V + E)$, where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is $O(V)$.

Application of DFS Algorithm

1. For finding the path
2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph
4. For detecting cycles in a graph

Short path algorithm

Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

A weighted graph is a graph in which each edge has a numerical value associated with it.

Floyd-Warshall algorithm is also called as Floyd's algorithm, Roy-Floyd algorithm, Roy-Warshall algorithm, or WFI algorithm.

Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

This algorithm follows the dynamic programming approach to find the shortest paths.

Floyd-Warshall Algorithm

For a graph with N vertices:

Step 1: Initialize the shortest paths between any 2 vertices with Infinity.

Step 2: Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex and so on. Until using all N vertices as intermediate nodes.

Step 3: Minimize the shortest paths between any 2 pairs in the previous operation.

Step 4: For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: $\min(\text{dist}[i][k] + \text{dist}[k][j], \text{dist}[i][j])$.

$\text{dist}[i][k]$ represents the shortest path that only uses the first K vertices, $\text{dist}[k][j]$ represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from i to k, then from k to j.

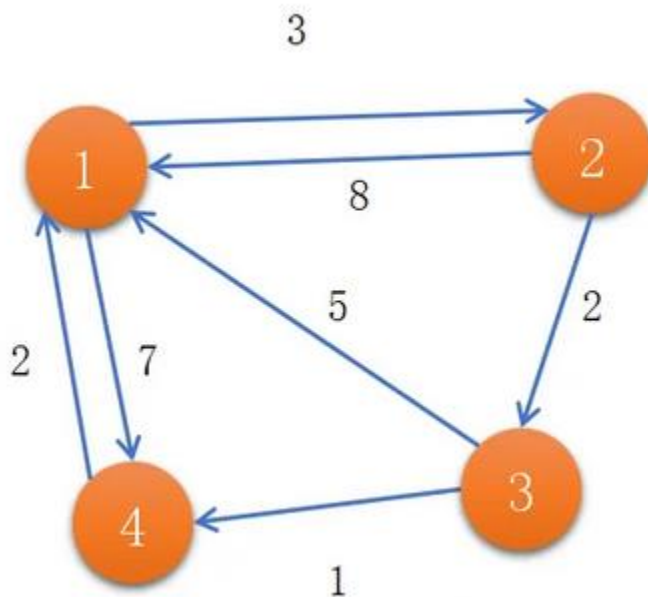
```

for(int k = 1; k <= n; k++){
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= n; j++){
            dist[i][j] = min( dist[i][j], dist[i][k] + dist[k][j] );
        }
    }
}

```

How Floyd-Warshall Algorithm Works?

Let the given graph be:



Follow the steps below to find the shortest path between all the pairs of vertices.

1. Create a matrix A^0 of dimension $n \times n$ where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

Each cell $A[i][j]$ is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.

Make a matrix A^0 which stores the information about the minimum distance of path between the direct paths for every pair of vertices. If there is no edge between two vertices

$$A^0 =$$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

Step-2

In this step, we use A^0 matrix and find the shortest path via 1 as an intermediate node. Here all the path that belongs to 1 remain unchanged in the matrix A^1 .

Now, create a matrix A^1 using matrix A^0 . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.

Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. $A[i][j]$ is filled with $(A[i][k] + A[k][j])$ if $(A[i][j] > A[i][k] + A[k][j])$.

That is, if the direct distance from the source to the destination is greater than the path through the vertex k , then the cell is filled with $A[i][k] + A[k][j]$.

$$A^1 =$$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	2	8	∞	0

Step-3

In this step, we use A^1 matrix and find the shortest path via 2 as an intermediate node. Here all the path that belongs to 2 remain unchanged in the matrix A^2 .

Similarly, A^2 is created using A^1 . The elements in the second column and the second row are left as they are.

In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in **step 2**.

$$A^2 =$$

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

Step-4

In this step, we use A^2 matrix and find the [shortest path](#) via 3 as an intermediate node. Here all the path that belongs to 3 remain unchanged in the matrix A^3

$$A^3 =$$

	1	2	3	4
1	0	3	5	6
2	7	0	2	3
3	5	8	0	1
4	2	5	7	0

Step-5

In this step, we use A^3 matrix and find the shortest path via 4 as an intermediate node. Here all the path that belongs to 4 remain unchanged in the matrix A^4 .

$$A^4 =$$

	1	2	3	4
1	0	3	5	6
2	5	0	2	3
3	3	6	0	1
4	2	5	7	0

A4 matrix is our final matrix which tells us the minimum distance between two vertices for all the pairs of vertices.

Floyd Warshall Algorithm Complexity

Time Complexity

There are three loops. Each loop has constant complexities. So, the time complexity of the Floyd-Warshall algorithm is $O(n^3)$.

Space Complexity

The space complexity of the Floyd-Warshall algorithm is $O(n^2)$.

Floyd Warshall Algorithm Applications

- To find the shortest path in a directed graph
- To find the transitive closure of directed graphs

- To find the Inversion of real matrices
- For testing whether an undirected graph is bipartite