UNIT-3

Brute Force & Exhaustive Search: Introduction to Brute Force approach, Selection Sort and Bubble Sort, Sequential search, Exhaustive Search- Travelling Salesman Problem and Knapsack Problem, Depth First Search, Breadth First Search

Introduction to Brute Force approach

There are various approaches to solve a given problem. In a straight forward approach and simple technique such as brute force, there is no emphasis on the efficiency of the algorithm. The concept used in this technique is "Just do it".

Definition:

The straight forward method of solving a given problem based on the problem's statement and definitions is called Brute Force technique. This method is often easier to implement than a more sophisticated one and, because of this simplicity. sometimes it can be more efficient.

For example, the algorithms to find GCD of 2 numbers (Consecutive integer checking method, Matrix multiplication, addition etc, Selection sort, Bubble sort, Linear search (sequential search), Brute-force string matching etc. can be solved using brute force technique.

The various advantages and disadvantages of this approach are shown below:

Advantages

- This method is applicable for wide variety of problems such as finding the sum of n numbers, Computing power, Computing GCD and so on
- Simple and easy algorithms can be written. For example, bubble sort, selection sort, matrix multiplication etc
- Can be used to judge more efficient alternative approaches to solve a problem

Disadvantages:

- Rarely yields efficient algorithms
- Some root force algorithms are unacceptably slow, for example bubble sort.
- Not as a constructive or creative as other design techniques such as divide and conquer.

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

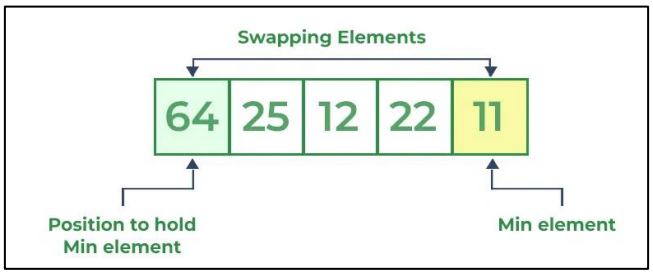
Procedure:

As the name indicates, we first find the smallest item in the list and we exchange it with the first item, obtain the second smallest in the list and exchange it with the second element, and so on. Finally, all the elements will be arranged in ascending order, since the next last item is selected and exchanged appropriately so that the elements are finally sorted. This technique is called ***selection sort***.

Let us see, How the elements {***64, 25, 12, 22,11***} can be sorted using Selection Sort.

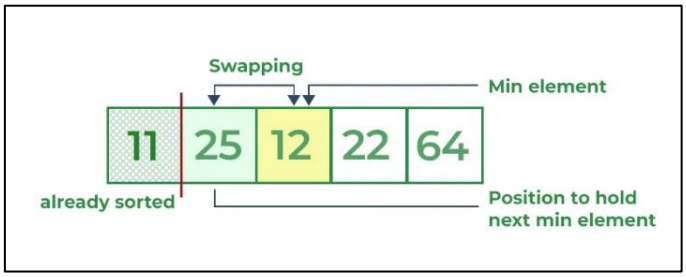
First pass:

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where 64 is stored presently, after traversing whole array it is clear that 11 is the lowest value.
- Thus, replace 64 with 11. After one iteration 11, which happens to be the least value in the array, tends to appear in the first position of the sorted list.



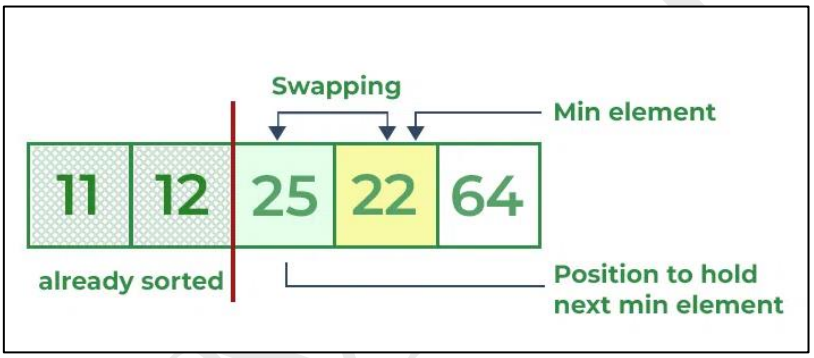
Second Pass:

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
- After traversing, we found that 12 is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.



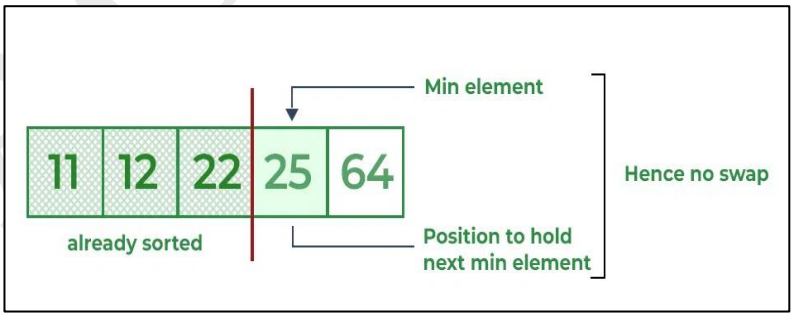
Third Pass:

- Now, for third place, where 25 is present again traverse the rest of the array and find the third least value present in the array.
- While traversing, 22 came out to be the third least value and it should appear at the third place in the array, thus swap 22 with element present at third position.



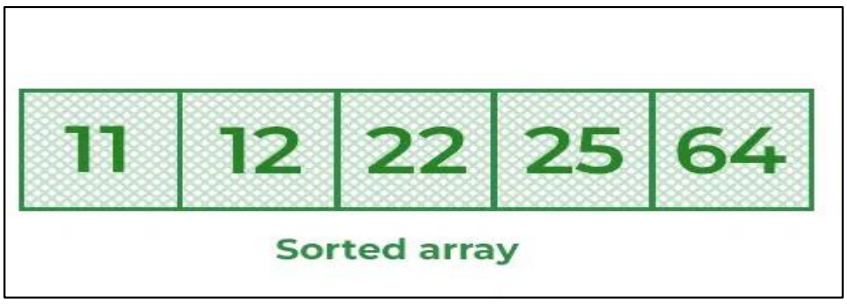
Fourth pass:

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- As 25 is the 4th lowest value hence, it will place at the fourth position.



Fifth Pass:

- At last the largest value present in the array automatically get placed at the last position in the array
- The resulted array is the sorted array.



Algorithm for Selection Sort:

```
Algorithm SelectionSort(arr, n)
//Purpose: Sort the given elements using selection sort
//Inputs:
//  - n: the number of items present in the array
//  - arr: the items to be sorted are present in the array
//Output:
//  - arr: contains the sorted list

for i ← 0 to n-2 do
    pos = i                    //Assumes ith element as smallest
    for j ← i+1 to n-1 do          // Find the position of the
smallest item
        if (arr[j] < arr[pos])
        pos ← j
    end for
    temp = arr[pos]
    arr[pos] = arr[i]
    arr[i] = temp
end for
```

Complexity Analysis of Selection Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic:

- If we are given n elements, then in the first pass, it will do n-1 comparisons;
- in the second pass, it will do n-2; in the third pass,
- it will do n-3 and so on. Thus, the total number of comparisons can be found by

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the selection sort algorithm encompasses a time complexity of ***O(n2)*** and a space complexity of ***O(1)*** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of ***O(n2)*** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is ***O(n2)***, 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(n2),*** which occurs when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is ***O(n2)*** in all three cases. This is because, in each step, we are required to find m

Bubble Sort

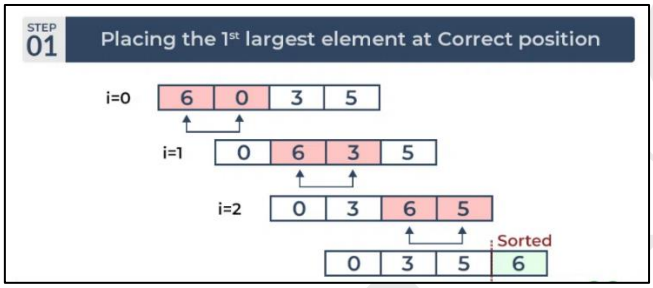
Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets.

Let us understand the working of bubble sort with the help of the following illustration:

Input: arr[] = {6, 3, 0, 5}

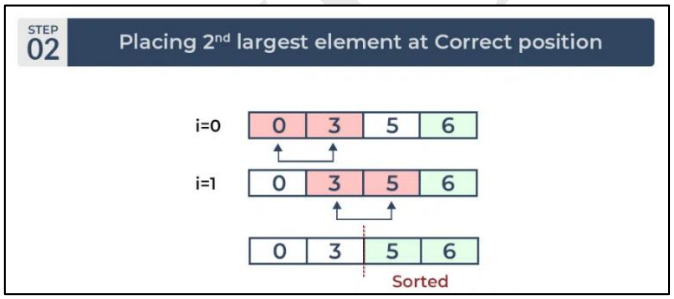
First Pass:

The largest element is placed in its correct position, i.e., the end of the array.



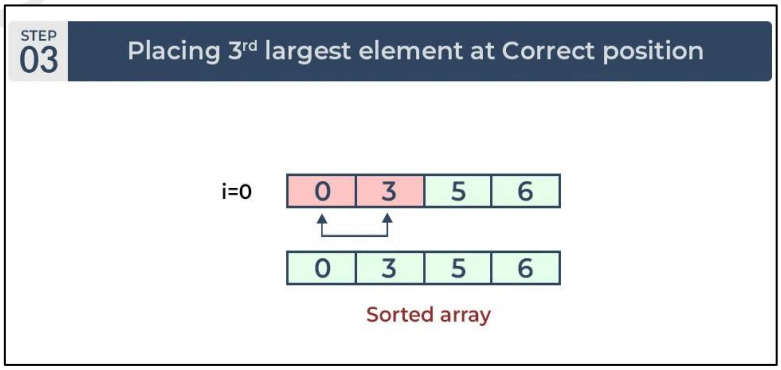
Second Pass:

Place the second largest element at correct position



Third Pass:

Place the remaining two elements at their correct positions.



Algorithm for Bubble Sort:

```
Algorithm BubbleSort(a[], n)
// Purpose: Arrange the numbers in ascending order
// Inputs:
//   n:the items to be sorted are present in the array
//   a: the number of items present in the array
// Output:
//   a: contains the sorted list

    for j ← 1 to n-1 do              // Perform n-1 passes
         for i ← 0 to n-j-1 do       // To compare items in each
pass
           if (a[i] > a[i+1]) then   // If out of order, exchange
adjacent
                temp = a[i]                elements
                a[i] = a[i+1]
                a[i+1] = temp
           end if
        end for
end for
```

Complexity Analysis of Bubble Sort

Input: Given a list of n elements.

Output: The number of steps required to sort the list.

Logic:

- During the first pass, Bubble Sort performs n-1 comparisons.
- In the second pass, it does n-2 comparisons,
- and in the third pass, it does n-3, continuing in this manner. Consequently, the total number of comparisons can be determined by:

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Time Complexity:

- **Best Case:** Bubble sort has a best-case time complexity of O(n) when the array is already sorted.
- **Average Case:** The average-case time complexity is O(n^2), occurring when two or more elements are in a jumbled order (neither ascending nor descending).
- **Worst Case:** The worst-case time complexity is O(n^2), happening when sorting a descending order array into ascending order.

Space Complexity: The space complexity is O(1) as it requires some extra memory space for a temporary variable used in swapping.

Sequential search:

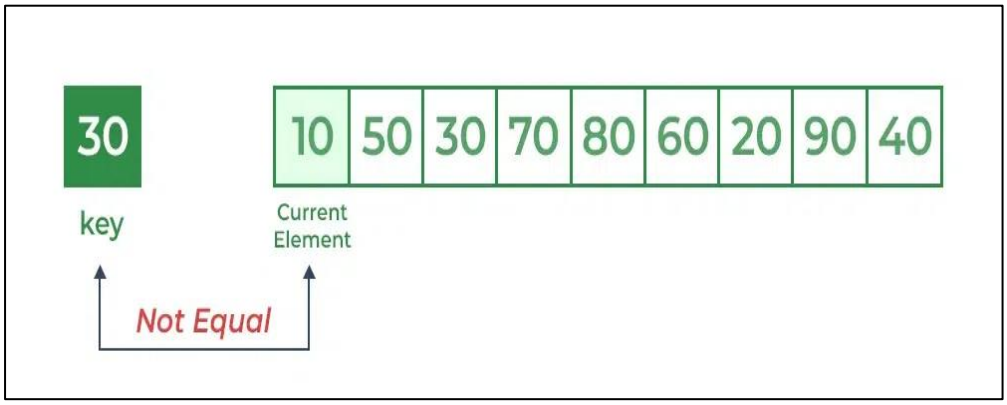
A linear search is a simple searching technique. In this technique we search for a given key item in the list of linear order, that is one after another. The item to be searched is often called key item. The *linear search* is also called as sequential search.

For example: Consider the list{10, 50, 30, 70, 80, 20, 90, 40} and key = 30 is present in the list,we say search is successful. If key is 100 and after searching we say that key is not present and hence search is unsuccessful.

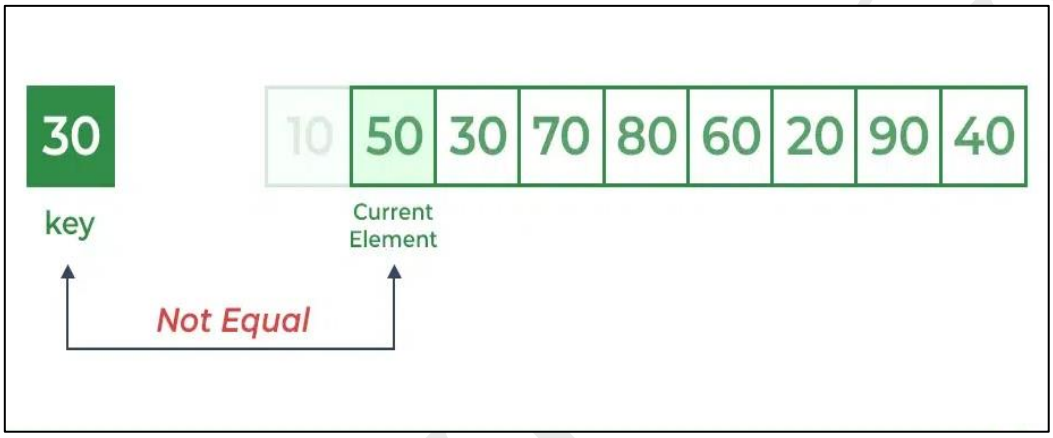
Working of Sequential Search:

Step 1: Start from the first element (index 0) and compare key with each element (arr[i]).

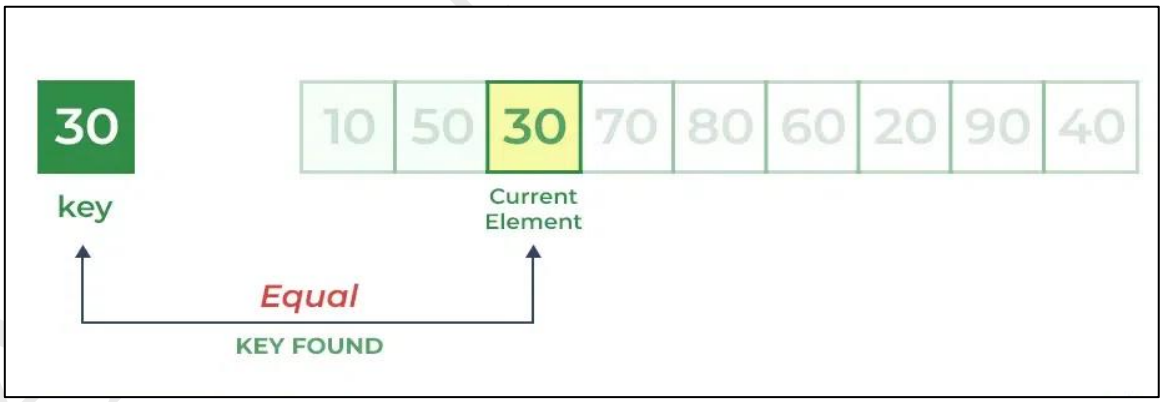
- Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.



- Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.



Step 2: Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).



Algorithm:Sequential Search

```
Algorithm LinearSearch(key, a[], n):
// Purpose: This algorithm searches for the key in the array 'a'
which has 'n' elements.

// Inputs:
// - n: the number of items present in the array
// - a: the items in the array where searching takes place
// - key: the item to be searched

// Output:
//  The function returns the position if the key is found;
Otherwise, the function returns -1 indicating the search is
unsuccessful.

    a[n] = key             // Insert the search key at the end of
the list
    i = 0                  // Start searching from the beginning of
the array

    while (a[i] != key) do
        i = i + 1
    end while

    if (i < n) then
        return i           // Key found at position i
    else
        return -1          // Key not found
    end if
```

Sequential Search Complexity Analysis

Input: A list of n elements and a target element to search.

Output: The number of steps required to find the target element in the list.

Logic: Sequential Search checks each element in the list one by one until the target element is found or the entire list is traversed.

Total comparisons = 1 + 2 + 3 + ... + (n-1) + n

Time Complexity:

- **Best Case:** O(1) when the target is at the beginning of the list.
- **Average Case:** O(n), occurs when the target is randomly located in the list.
- **Worst Case**: O(n), when the target is at the end of the list or not present.
- **Space Complexity:** O(1), as it requires only a constant amount of extra memory for control variables.

Exhaustive Search

Traveling Sales Man Problem:

Definition:

Given n cities, a salesperson starts at a specified city (often called source), visit all n-1 cities only once and return to the city from where he has started.

The ***objective*** of this problem is to find a route through the cities that minimizes the cost there by maximizing the profit.

This problem can be modelled by a directed weighted graph as shown below:

- The vertices of the graphs represent the various cities
- The weights associated with edges represent the distances between two cities or the cost involved while traveling from one city to other city.
- The cost involved from city i to city j is represented using a 2-dimensional array and C[i, j] gives the cost from city i to city j.

TSP Using Brute force:

Using brute force approach the TSP can be solved as shown below:

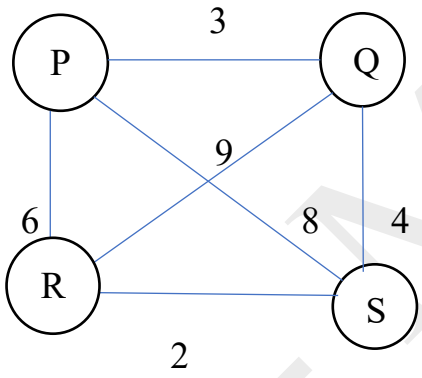
- Get all the routes from one city to another city by taking various permutations
- Compute the route length for each permutation and select the shortest among them.

But, in this technique, the execution time increases very rapidly as the size of the TSP problem increases. **For example,**

- If the number of cities = 16, the number of possible routes are 1,307,674,368,000
- If number of cities =20, the number of possible routes are 12,164,510,040,883,200.

Note: Thus, the easiest way to find an exact solution to the traveling salesman problem is to compute the cost of all tours and determine the tour with the minimum cost. The above method is called exhaustive sequential search algorithm and uses brute force approach.

Example:



- Where the cities are represented as P ,Q ,R & S and the distance between various cities are represented as number in each of the edge.
- The graph represented using cost adjacency matrix is also shown below

	P	Q	R	S
P	0	3	6	8
Q	3	0	9	4
R	6	9	0	2
S	8	4	2	0

Working of TSP Brute Force

1. Consider city P is the Starting and ending vertex

2. Generate all (n-1)! Permutations of cities
3. Calculate the cost of the every permutation and keep track on the minimum cost permutation.
4. Return the Permutation of the Minimum cost

Various Possible routs are:

p→q→r→s→p (3 + 9+ 2 + 8 = 22)

p→q→s→r→p (3 + 4 + 2 + 6 = 15)

p→r→q→s→p (6 + 9 + 4 + 8 = 27)

p→r→s→q→p (6 + 2 + 4 + 3 = 15)

p→s→q→r→p (8 + 4 + 9 + 6= 27)

p→s→r→q→p (8 + 2 + 9 + 3= 22)

- we have variety of routes with varying costs, we have to consider the route with minimum cost to get the maximum profit. So, the following routes can be selected by the salesperson:
  p→q→s→r→p (Cost = 15)
  p→r→s→q→p (Cost = 15)

Knapsack Problem Greedy Method

Definition:

Given a knapsack, bag or container of capacity m&n objectives of weights w_1,w_2,w_3…w_n with profit $p_1p_2p_32…p_n$, let $x_1x_2x_3$ …$x_n$be the fraction of the objects that are supposed to be added to the knapsack.

Objective:

The main objective is to place the objects into the knapsack so that the maximum profit is obtained and the weight of the object chosen should not exceed the capacity of knapsack.

Example:

Greedy Method

Fractional Knapsack problem

W = 15 n = 7

Objects	-	1	2	3	4	5	6	7
Profit (P)	-	5	10	15	7	8	9	4
Weight (w)	-	1	3	5	4	1	3	2
P/w	-	5	3.3	3	1.75	8	3	2

I Max Profit

Objects	Profit	Weight	Remaining Weight
3	15	5	15 - 5 = 10
2	10	3	10 - 3 = 7
6	9	3	7 - 3 = 4
5	8	1	4 - 1 = 3
4	7 x 3/4	3	3 - 3 = 0
	= 5.25		

= 47.25

Min weight

Objects	Profit	Weight	Remaining Weight
1	5	1	15 - 1 = 14
5	8	1	14 - 1 = 13
7	4	2	13 - 2 = 11
2	10	3	11 - 3 = 8
6	9	3	8 - 3 = 5
4	7	4	5 - 4 = 1
3	3 15x1/5	1	1 - 1 = 0
	46		

III Max P/W Ratio

Objects	Profit	Weight	Remaining Weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2
7	4	2	2 - 2 = 0
	51		

Algorithm for Knapsack Greedy method:

```
Algorithm GreedyKnapsack(m, n, w, x)
// Purpose: To find the solution vector that shows the fraction of
the object selected in the knapsack problem
// Input: m - the capacity of the knapsack, n - the number of
objects, w - an array of weights of all n objects
// Output: x - solution vector containing the fraction of each
object selected

// Initialize the solution vector x
for i from 1 to n do
    x[i] = 0
end for

// Initialize the remaining capacity of the knapsack
rc = m

// Greedy selection of objects
for i from 1 to n do
    // If the weight of the current object is greater than the
remaining capacity, break
    if w[i] > rc then
        break
    end if

    // Select the entire object
    x[i] = 1

    // Update the remaining capacity of the knapsack
    rc = rc - w[i]

    // If there is more space, select a fraction of the object
    if i <= n then
        x[i] = rc / w[i]
    end if
end for
```

Knapsack 0/1 Problem:

What is 0/1 knapsack problem

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Objective:

The main objective is to place the objects into the knapsack so that the maximum profit is obtained and the weight of the object chosen should not exceed the capacity of knapsack.

Definition : Given an knapsack. with following.

M - capacity of the knapsack.

n - number of objects.

W - an array consisting of weights $w_1,w_2,w_3 \ldots w_n$

P - Consisting of Profit $p_1p_2p_32 \ldots p_n$.

Example problem:

$$V[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \le j \end{cases}$$

Note: **V[n, M]** gives the optimal profit obtained. Now, let us solve one problem and see how to get the optimal solution using knapsack problem.

Example 5.8: Apply bottom-up dynamic programming technique to the following instance of the knapsack problem with capacity M = 5

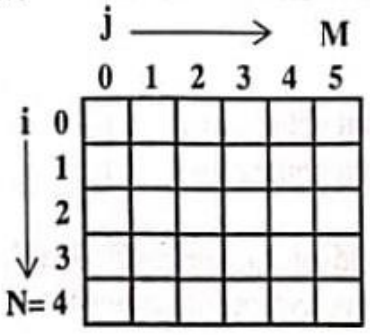
:

Item	Weight	Value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Solution: The following data is provided:

- Number of objects N = 4
- Capacity of the knapsack M = 5
- $w_1 = 2, w_2 = 1, w_3 = 3, w_4 = 2$
- $p_1 = 12, p_2 = 10, p_3 = 20, p_4 = 15$

Step 1: Since N = 4, M = 5 we have a solution matrix consisting of N+1 rows and M+1 columns as shown below:



Step 2: In this bottom up approach, we start from the initial solution with i = 0 or j = 0. Using equation (1) we have

$$V[i, j] = 0 \qquad \text{if } (i = 0 \text{ or } j = 0)$$

So, when i = 0 (i.e., when no objects are there), irrespective of the remaining capacity j = 0, 1, 2, 3, 4, 5 the profit V[i, j] = 0. Similarly, when j = 0 (i.e., when there is no knapsack) irrespective of number of objects i = 0, 1, 2, 3, 4 the profit V[i, j] = 0. These entries are made in above table. The resulting table is shown below:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
N=4	0					

Profit table

Step 3: When i = 1, W_i = 2 and P_i = 12, we have to compute the results for various values of j = 1, 2, 3, 4, 5 (See row 1 of above table. Those values should be computed). The results of row 1 can be computed as shown below:

$V[i, j] = V[i-1, j]$ if $W_i > j$ .....(1)

$V[i,j] = Max(V[i-1, j], V[i-1, j-W_i]+P_i)$ if $W_i \leq j$.....(2)

j=1

W_i	j	
2	1	V[1, 1] = V[0, 1] = 0
2	2	V[1, 2] = Max(V[0, 2], V[0, 0] + 12) = Max (0 , 0 + 12) = 12
2	3	V[1, 3] = Max(V[0, 3], V[0, 1] + 12) = Max(0, 0 + 12) = 12
2	4	V[1, 4] = Max(V[0, 4], V[0, 2] + 12) = Max(0, 0 + 12) = 12
2	5	V[1, 5] = Max(V[0, 5], V[0, 3] + 12) = Max)9, 0 + 12) = 12

Note: Only when Wi = 2, = 1, relation (1) is used. For the rest we use the relation (2)

Substituting these computed values in previous profit table, following table is obtained:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0					
3	0					
N=4	0					

Profit table

Step 4: When i = 2, W_i = 1 and P_i = 10, compute the results for various values of j as shown below:

$V[i, j] = V[i-1, j]$ if $W_i > j$ .....(1)

W_i j $V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i)$ if $W_i \le j$.....(2)

1	1	V[2, 1] = Max(V[1, 1], V[1, 0] + 10) = Max (0, 0 + 10) = 10
1	2	V[2, 2] = Max(V[1, 2], V[1, 1] + 10) = Max(12, 0 + 10) = 12
1	3	V[2, 3] = Max(V[1, 3], V[1, 2] + 10) = Max(12, 12+10) = 22
1	4	V[2, 4] = Max(V[1, 4], V[1, 3] + 10) = Max(12, 12 + 10) = 22
1	5	V[2, 5] = Max(V[1, 5], V[1, 4] + 10) = Max(12, 12 + 10) = 22

By storing all these values in the previous profit table, the following table is obtained.

i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0					
N= 4	0					

(j ⟶ M; i ↓)

Profit table

Step 5: When i = 3, $W_i = 3$ and $P_i = 20$, the values of row 3 (of above table) are computed as shown below:

$V[i, j] = V[i-1, j]$ if $W_i > j$ .....(1)

W_i j $V[i,j] = \text{Max}(V[i-1, j], V[i-1, j-W_i]+P_i)$ if $W_i \le j$.....(2)

3	1	V[3, 1] = V[2, 1] = 10
3	2	V[3, 2] = V[2, 2] = 12
3	3	V[3, 3] = Max(V[2, 3], V[2, 0] + 20) = Max(22, 0 + 20) = 22
3	4	V[3, 4] = Max(V[2, 4], V[2, 1] + 20) = Max(22, 10+20) = 30
3	5	V[3, 5] = Max(V[2, 5], V[2, 2] + 20) = Max(22, 12 + 20) = 32

By substituting these values in the previous profit table, we have the following table:

i \ j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
N= 4	0					

(j ⟶ M; i ↓)

Profit table

Step 6: When i = 4, W_i = 2 and P_i = 15, the values of row 4 (of above table) are computed as shown below:

$V[i, j] = V[i-1, j]$ if $W_i > j$ .....(1)

W_i j $V[i,j] = Max(V[i-1, j], V[i-1, j-W_i]+P_i)$ if $W_i \le j$.....(2)

2	1	V[4, 1] = V[3, 1] = 10
2	2	V[4, 2] = Max(V[3, 2], V[3, 0] + 15) = Max(12, 15) = 15
2	3	V[4, 3] = Max(V[3, 3], V[3, 1] + 15) = Max(22, 10+ 15) = 25
2	4	V[4, 4] = Max(V[3, 4], V[3, 2] + 15) = Max(30, 12 + 15) = 30
2	5	V[4, 5] = Max(V[3, 5], V[3, 3] + 15) = Max(32, 22+15) = 37

By substituting these values in the previous profit table, we have the table shown below:

j ⟶ M

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
N= 4	0	10	15	25	30	37

Profit table

Optimal solution

It is given that n = 4 and M = 5. Therefore, optimal solution **V[n, M] = V[4, 5] = 37**

Now, the complete algorithm to find the optimal solution for the knapsack can be obtained using the recurrence relation

$$V[i, j] = \begin{cases} 0 & \text{if } i = j = 0 \\ V[i-1, j] & \text{if } w_i > j \\ \max(V[i-1, j], V[i-1, j-w_i] + P_i) & \text{if } w_i \le j \end{cases}$$

```
Algorithm KNAPSACK(n,m,w,p,v)
// Purpose: To find the optimal solution for the knapsack problem
using dynamic // programming
// Input:
// - n: Number of objects to be selected
// - m: Capacity of the knapsack
// - w: Weights of all the objects
// - p: Profits of all the objects

// Output:
// - v: The optimal solution for the number of objects selected
with specified remaining capacity
for i ← 1 to n do
    for j ← 0 to m do
         if(i = 0 or j = 0)
              V[i,j] ← 0
         else if(w[i] > j) then
              V[i, j] ← V[i-1, j]
         else
              V[i, j] ← max(V[i-1, j], V[i-1, j - w[i]] + p[i])
         end if
    end for
end for
```

Depth First Search

The depth first search is a method of traversing the graph by visiting each node of the graph in a systematic order. As this name implies, ***depth- first- search*** means to search the deeper in the graph

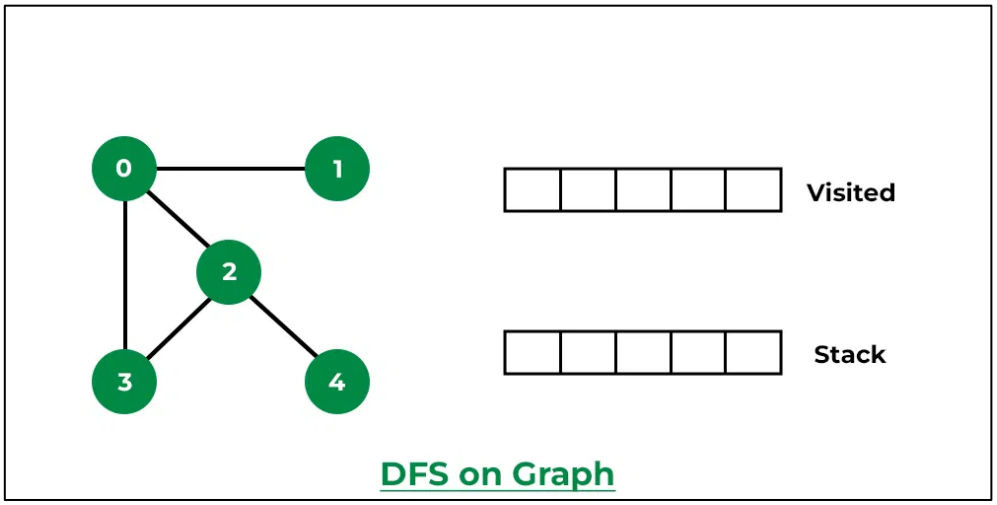
- In depth first search ***Stack*** is used.
- When a vertex is reached for the first time it pushed onto the stack and each vertex is numbered in the order in which it pushed on to the stack
- The order in which the vertices become dead ends when a vertices is dead end (i.e.. All adjacent vertices are explored). It is removed from the stack. Each node is numbered in order in which it is deleted from the stack.

The step by step process to implement the DFS traversal is given as follows -

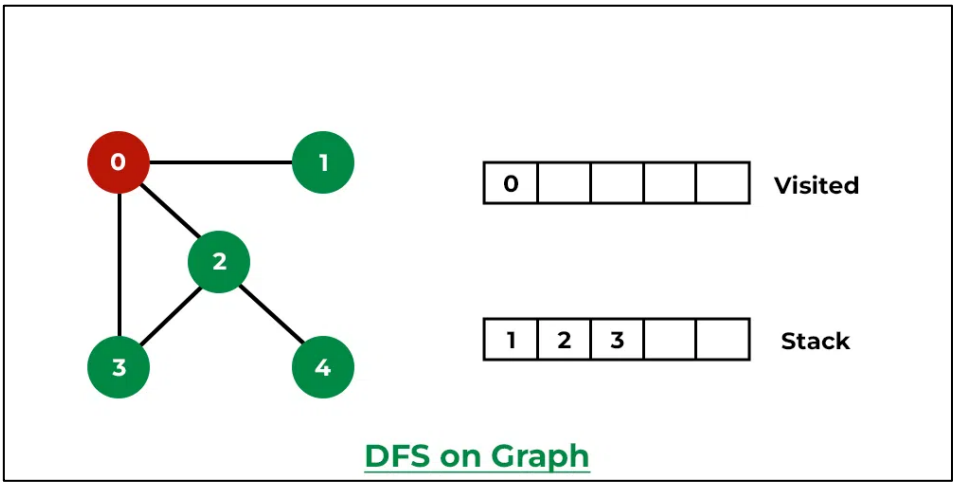
1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Example:

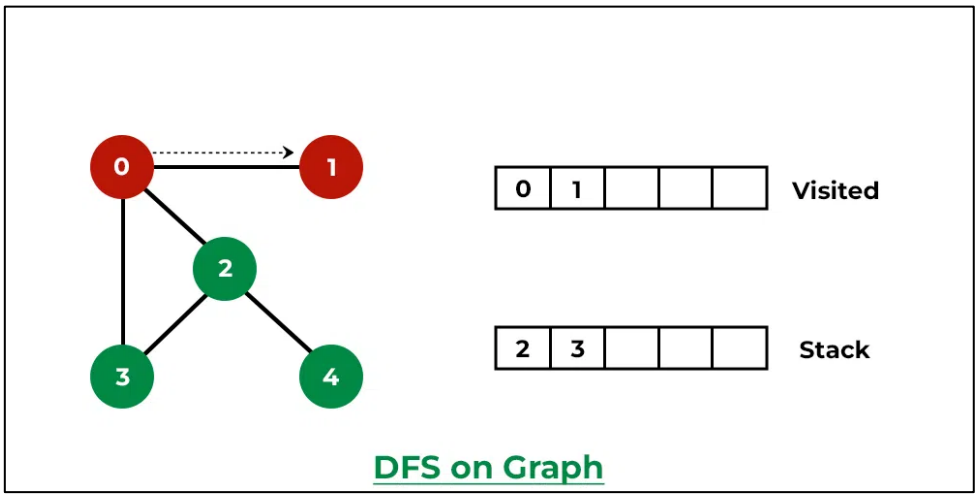
Step1: Initially stack and visited arrays are empty.



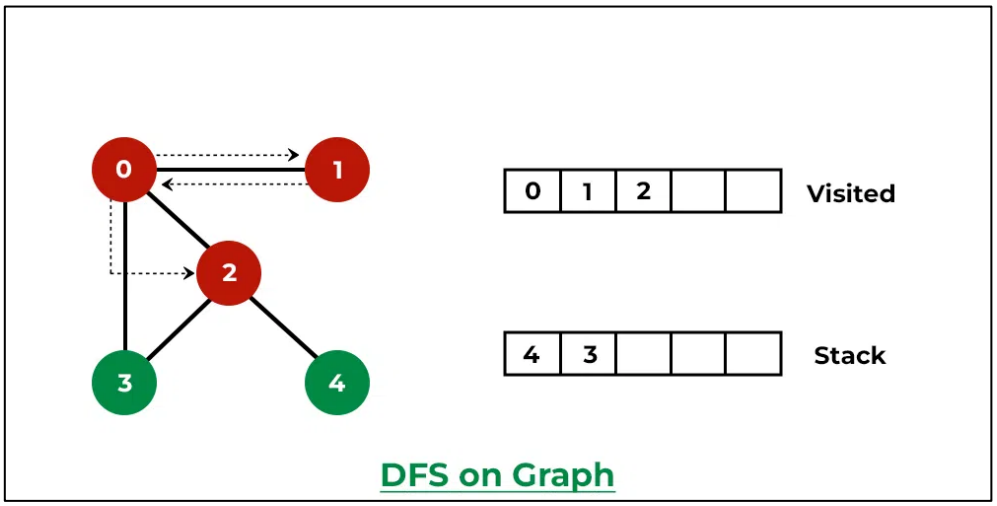
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.



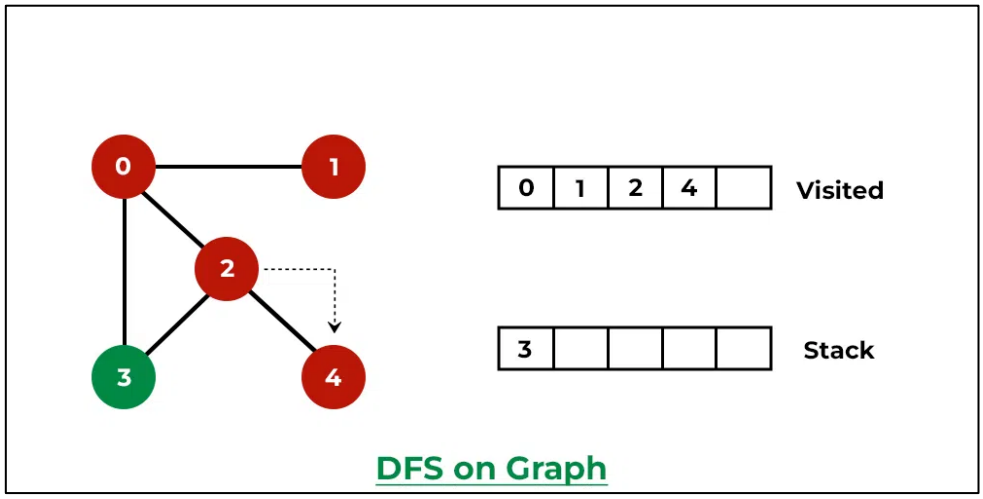
Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



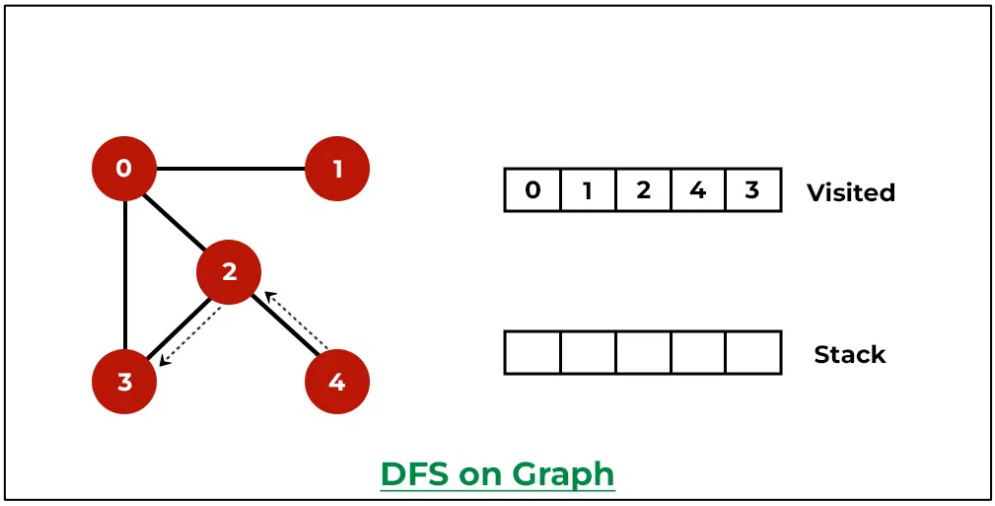
Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.



Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Algorithm to traverse the graph using DFS

```
Algorithm: DFS (a,n,u,s,t)

// Purpose: Traverse the graph from the given node source in DFS

// Input: a - adjacency matrix of the given graph

//        n - the number of nodes in the graph

//        u - the node from where the traversal is initiated

//        s - indicates the vertices that are visited and that are
not visited
```

```
// Output: (u, v) - the nodes 'v' reachable from 'u' are stored in
a vector 't'


s[u] = 1        //visit the node
for every v adjacent to vertex u do
    if v is not visited then
        t[k][0] ← u         //v is the node visited
        t[k][1] ← v         //Store the edge u-v
        k ← k + 1
        dfs(a, n, v, s, t)
    end if
end for
```

Breadth First Search

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node

The step by step process to implement the BFS traversal is given as follows –

1. Create a queue and mark all vertices as not visited.

2. Choose a starting vertex and enqueue it into the queue.

3. Mark the chosen vertex as visited.

4. Dequeue a vertex from the queue and visit it.

5. Enqueue all non-visited neighbors (adjacent vertices) of the dequeued vertex.

6. Repeat steps 4 and 5 until the queue is empty.

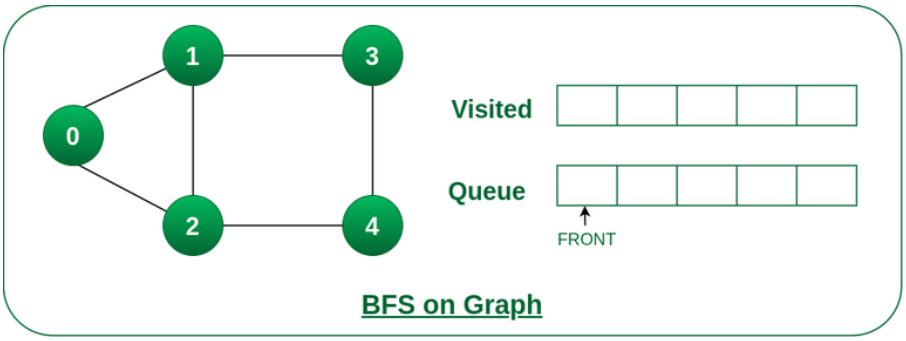
7. If there are still unvisited vertices, go back to step 2 and choose another starting vertex.

8. Repeat steps 2-7 until all vertices are visited.

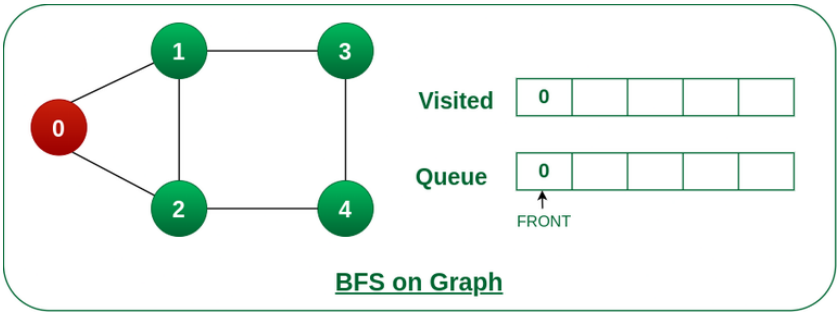
Illustration:

Let us understand the working of the algorithm with the help of the following example.

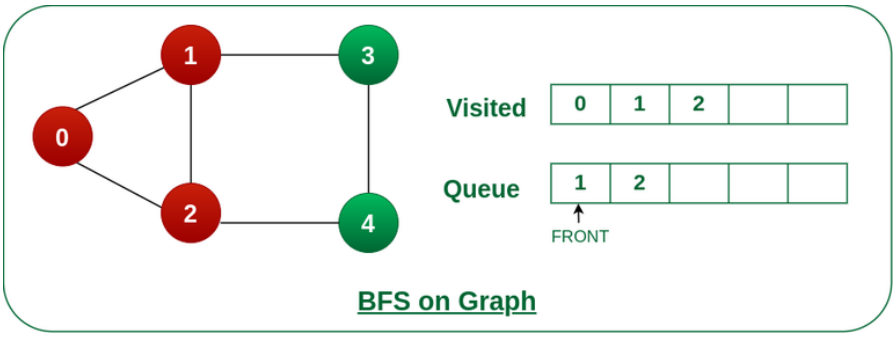
Step1: Initially queue and visited arrays are empty.



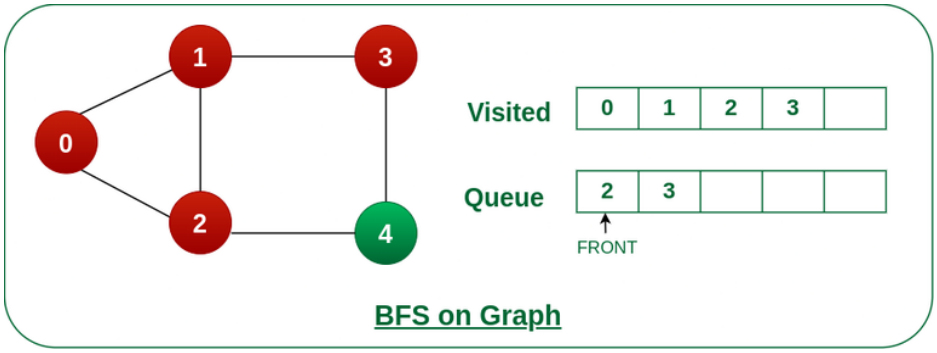
Step2: Push node 0 into queue and mark it visited.



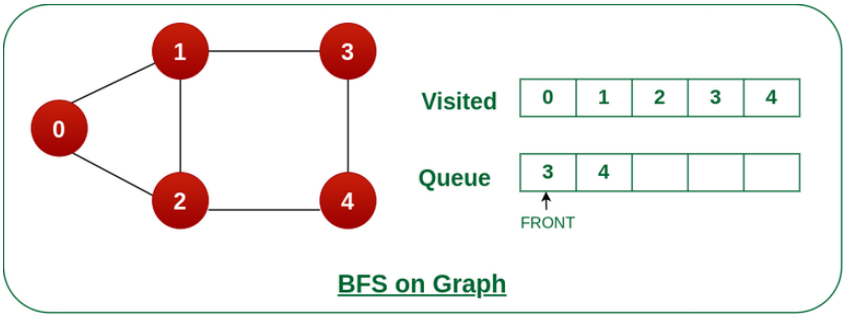
Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.

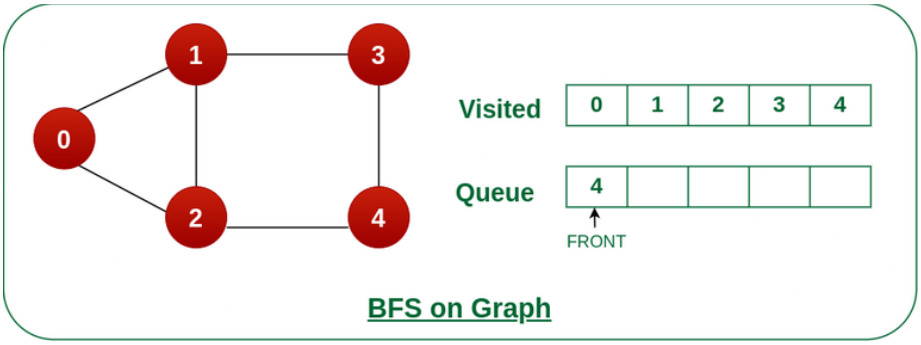


Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



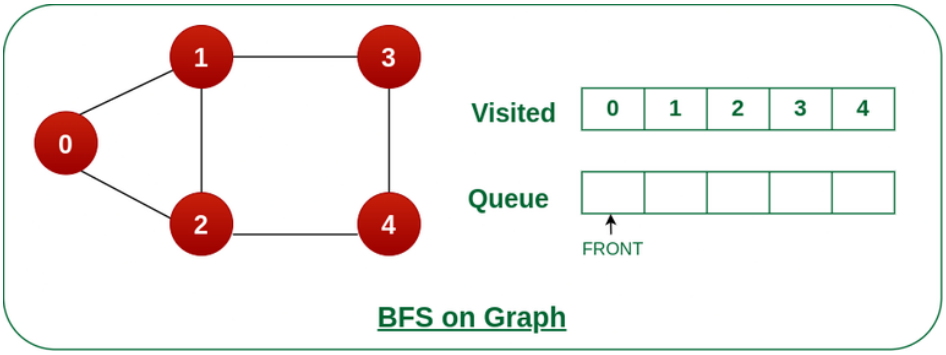
Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Steps 7: Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Now, Queue becomes empty, So, terminate these process of iteration.

Algorithm to traverse the graph using BFS

Algorithm BFS (a,n,source,T)

```
// Purpose: Traverse the graph from the given source node in BFS
// Input: a - adjacency matrix of the given graph
//        n - the number of nodes in the graph
//        source - the node from where the traversal is initiated
// Output:
//       (u, v) - the nodes v reachable from u are stored in a vector T
for i from 0 to n-1 do
    s[i] = 0
end for
// No node is visited
f ← 0
r ← 0
q[r] ← source
s[source] ← 1
k ← 0
while (f <= r) do
    u ← q[f]
    f ← f + 1

    for every v adjacent to u do
       if s[v] is not visited
            s[v] ← 1
            r ← r + 1
            q[r] ← v

            T[k, 1] ← u
            T[k, 2] ← v,  k ← k + 1
```

```
        end if
    end for
end while
```