



Silver Oak College of Engineering & Technology

GUJARAT TECHNOLOGICAL UNIVERSITY  
BACHELOR OF ENGINEERING

ANALYSIS AND DESIGN OF ALGORITHMS  
(3150703)

5<sup>th</sup> SEMESTER

COMPUTER ENGINEERING

# Laboratory Manual



## **ANALYSIS AND DESIGN OF ALGORITHMS PRACTICAL BOOK**

### **DEPARTMENT OF INFORMATION TECHNOLOGY**

#### **PREFACE**

It gives us immense pleasure to present the first edition of Analysis and Design of Algorithms Practical Book for the B.E. 3<sup>rd</sup> year students for **SILVER OAK GROUP OF INSTITUTES**.

The Analysis and Design of Algorithms theory and laboratory course at **SILVER OAK COLLEGE OF ENGINEERING & TECHNOLOGY, AHMEDABAD** is designed in such a way that students develop the basic understanding of the subject in the theory classes and then try their hands on the experiments to realize the various devices and circuits learnt during the theoretical sessions. The main objective of the ADA laboratory course is: Learning ADA through algorithm. The objective of this ADA Practical Book is to provide a comprehensive source for all the experiments included in the ADA laboratory course.

We acknowledge the authors and publishers of all the books which we have consulted while developing this Practical book. Hopefully this *Analysis and Design of Algorithms Practical Book* will serve the purpose for which it has been developed.



### **Instructions to Students**

1. Be prompt in arriving to the laboratory and always come well prepared for the experiment.
2. Students are instructed to remove your shoes and shocks outside the lab. Also don't keep your bags on the table.
3. Students need to maintain a proper decorum in the computer lab. Students must use the equipment with care. Any damage is caused is punishable.
4. Students are instructed to come to lab in formal dresses only.
5. Students are supposed to occupy the systems allotted to them and are not supposed to talk or make noise in the lab.
6. Students are required to carry their Lab Manual with completed practical while entering the lab.
7. Lab Manual need to be submitted every week.
8. Students are not supposed to use pen drives in the lab. The grades for the Analysis and Design of Algorithms Practical course work will be awarded based on your performance in the laboratory, regularity, recording of experiments in the Analysis and Design of Algorithms Practical Final Book, lab quiz, regular viva-voce and end - term examination.
9. Find the answers of all the questions mentioned under the section 'Solve the below mention questionnaires' at the end of each experiment in the Analysis and Design of Algorithms Practical Book.
10. At the end of the lab, shut-down the computers, switch off power supply and put chairs properly.



## CERTIFICATE

*This is to certify that Mr./Ms DARSHIL PATEL with enrollment. No 190770107135 from Semester....5<sup>TH</sup>.. div A has successfully completed his/her laboratory experiments in the **Analysis and Design Of Algorithms** (3150703) from the department of **COMPUTER ENGINEERING** during the academic year 2021-22*

Date of Submission:.....

Staff In charge:.....

Head of Department:.....

## TABLE OF CONTENT

Sr. No	Experiment Title	Page No		Date of Start	Date of Completion	Sign	Marks (out of 10)
		To	From				
1	Implementation and Time analysis of sorting algorithms. a) Bubble sort, b) Selection sort, c) Insertion sort.	1	9				
2	Implementation and Time analysis of sorting algorithms. a) Merge sort b) Quick sort.	10	20				
3	Implementation and Time analysis of linear and binary search algorithm.	21	27				
4	Implementation of max-heap sort algorithm.	28	31				
5	Implementation of a knapsack problem using dynamic programming.	32	36				
6	Implementation of chain matrix multiplication using dynamic programming.	37	42				
7	Implementation of making a change problem using dynamic programming.	43	45				

Sr. No	Experiment Title	Page No		Date of Start	Date of Completion	Sign	Marks (out of 10)
		To	From				
8	Implementation of a knapsack problem using greedy algorithm.	46	51				
9	Implementation of Graph and Searching (DFS)	52	57				
10	Implementation of Graph and Searching (BFS)	58	64				
11	Implement prim's algorithm.	65	69				
12	Implement kruskal's algorithm.	70	77				

## **PRACTICAL SET - 1**

**AIM: Implementation and Time analysis of different sorting algorithms.**

### **Bubble Sort:**

Bubble Sort is a simple algorithm which is used to sort a given set of  $n$  elements provided in form of an array with  $n$  number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on. If we have total  $n$  elements, then we need to repeat this process for  $n-1$  times.

It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

### **Implementing Bubble Sort Algorithm**

Following are the steps involved in bubble sort (for sorting a given array in ascending order):

1. Starting with the first element (index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1.**

### **Complexity Analysis of Bubble Sort**

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

- 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:  $O(1)$

**AIM 1.1: Write a program to implement Bubble sort.**

**Code:**

```
#include <stdio.h>

void bubblesort(int arr[], int size)
{
    int i, j;
    for (i = 0; i < size; i++)
    {
        for (j = 0; j < size - i; j++)
        {
            if (arr[j] > arr[j+1])
                swap(&arr[j], &arr[j+1]);
        }
    }
}

void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int array[100], i, size;
    printf("How many numbers you want to sort: ");
    scanf("%d", &size);
    printf("\nEnter %d numbers : ", size);
    for (i = 0; i < size; i++)
        scanf("%d", &array[i]);
    bubblesort(array, size);
    printf("\nSorted array is ");

    for (i = 0; i < size; i++)
        printf(" %d ", array[i]);
    return 0;
}
```



## Output:

D:\C\ A\_15\_190770107097\PRACTICAL\bin\Debug\PRACTICAL\ A\_15\_190770107097.exe

```
How many numbers you want to sort: 5
Enter 5 numbers : 89
75
15
23
14
Sorted array is 14 15 23 75 89
Process returned 0 (0x0)   execution time : 17.527 s
Press any key to continue.
```

## Selection Sort:

Selection sort is conceptually the simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

## Implementing Selection Sort Algorithm

Following are the steps involved in selection sort (for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.

4. This is repeated, until the array is completely sorted.

## **Complexity Analysis of Selection Sort**

The following will be the time and space complexity for selection sort algorithm:

- Worst Case Time Complexity [ Big-O ]:  **$O(n^2)$**
- Best Case Time Complexity [Big-omega]:  **$O(n^2)$**
- Average Time Complexity [Big-theta]:  **$O(n^2)$**
- Space Complexity:  **$O(1)$**

**AIM 1.2: Write a program to implement Selection sort.**

**Code:**

```
#include <stdio.h>
void selectionSort(int arr[], int size);
void swap(int *a, int *b);

void selectionSort(int arr[], int size)
{
    int i, j;
    for (i = 0 ; i < size; i++)
    {
        for (j = i ; j < size; j++)
        {
            if (arr[i] > arr[j])
                swap(&arr[i], &arr[j]);
        }
    }
}

void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int array[10], i, size;
    printf("How many numbers you want to sort: ");
    scanf("%d", &size);
    printf("\nEnter %d numbers\t", size);
    printf("\n");
    for (i = 0; i < size; i++)
        scanf("%d", &array[i]);
    selectionSort(array, size);
    printf("\nSorted array is ");
    for (i = 0; i < size; i++)
        printf(" %d ", array[i]);
    return 0;
}
```

## Output:

D:\C\A\_15\_190770107097\PRACTICAL\bin\Debug\PRACTICAL\A\_15\_190770107097.exe

```
How many numbers you want to sort: 5
Enter 5 numbers
18
5
63
100
1566
Sorted array is 5 18 63 100 1566
Process returned 0 (0x0)   execution time : 11.686 s
Press any key to continue.
```

## Insertion Sort:

The **insertion sort** works in a slightly different way. It always maintains a sorted sublist in the lower positions of the list. Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger. The shaded items represent the ordered sublists as the algorithm makes each pass.

## Implementing Selection Sort Algorithm

Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at index 1, the **key**. The **key** element here is the new number that we need to add to our existing sorted set of numbers
2. We compare the **key** element with the element(s) before it, in this case, element at index 0:
  - If the **key** element is less than the first element, we insert the **key** element before the first element.
  - If the **key** element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as **key** and will compare it with elements to its left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

## Complexity Analysis of Selection Sort

- 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: **O(1)**

### **AIM 1.3: Write a program to implement Insertion sort.**

#### **Code:**

```
#include <stdio.h>
int main()
{
    int n, i, j, temp;
    int arr[64];

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    for (i = 1 ; i <= n - 1; i++)
    {
        j = i;
        while ( j > 0 && arr[j-1] > arr[j])
        {
            temp  = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = temp;
            j--;
        }
    }
    printf("Sorted list in ascending order:\n");
    for (i = 0; i <= n - 1; i++)
    {
        printf("%d\n", arr[i]);
    }
    return 0;
}
```

## Output:

```

D:\C\ A_15_190770107097\ PRACTICAL\bin\Debug\ PRACTICAL\ A_15_190770107097.exe
Enter number of elements
5
Enter 5 integers
10
50
69
1
500
Sorted list in ascending order:
1
10
50
69
500
Process returned 0 (0x0)   execution time : 23.899 s
Press any key to continue.

```

## Solve the below mention questionnaires:

1) The way a card game player arranges his cards as he picks them one by one can be compared to

- a) Quick sort
- c) Insertion sort**
- b) Merge sort
- d). Bubble sort

2) The correct order of the efficiency of the following sorting algorithms according to their overall running time comparison is

- a) Insertion>selection>bubble
- b) Insertion>bubble>selection
- c) Selection>bubble>insertion.
- d) bubble>selection>insertion**

3) Analysis the different sorting algorithm of Bubble, Selection and Insertion Sort. Justify your answer which one is better sorting algorithm among them.

No of Elements	1000	5000	10000	50000	100000
Bubble Sort	$1000^2$	$5000^2$	$10000^2$	$50000^2$	$100000^2$
Selection Sort	$1000^2$	$5000^2$	$10000^2$	$50000^2$	$100000^2$
Insertion Sort	$1000^2$	$5000^2$	$10000^2$	$50000^2$	$100000^2$

Best case time complexity of bubble, selection and insertion sort is  $O(n)$ ,  $O(n)$ ,  $O(n)$  respectively. Worst case time complexity of these three are  $O(n^2)$ ,  $O(n^2)$ ,  $O(n^2)$ . In Bubble and insertion sort insertion sort do less comparison than bubble sort. So insertion sort is best among these three. Insertion sort outperform merge sort too if number of element is le

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$

## PRACTICAL SET - 2

**AIM: Implementation and Time analysis of Merge & Quick sort algorithms.**

### Merge Sort:

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

### Implementing Merge Sort Algorithm:

- Divide the unsorted list into N sublists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

### Algorithm:

MergeSort(arr[], l, r)

If  $r > l$

1. Find the middle point to divide the array into two halves:  
middle  $m = (l+r)/2$
2. Call mergeSort for first half:  
Call mergeSort(arr, l, m)
3. Call mergeSort for second half:  
Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:  
Call merge(arr, l, m, r)

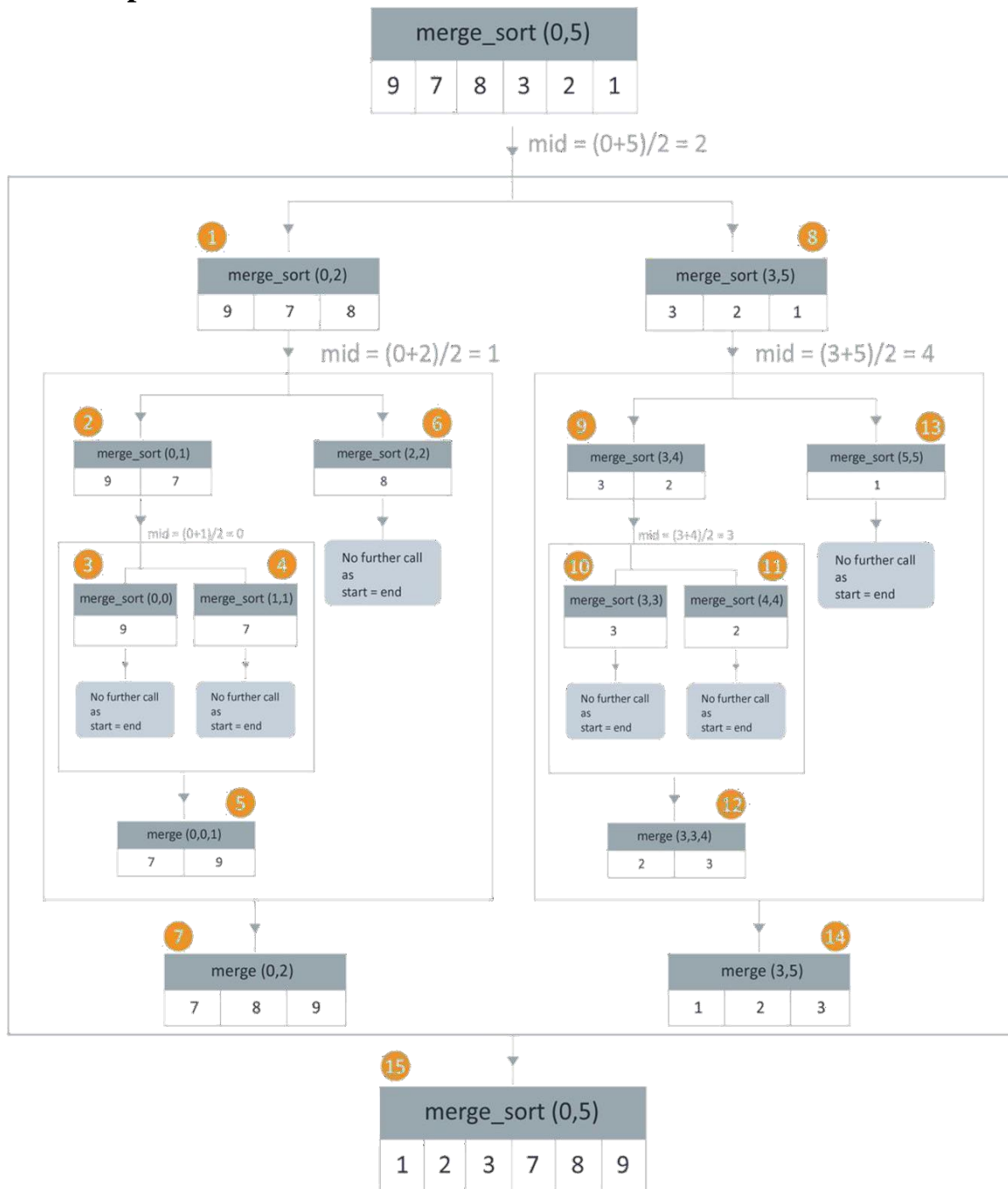
### Complexity Analysis of Merge Sort

The following will be the time and space complexity for merge sort algorithm:

- Worst Case Time Complexity [ Big-O ]:  **$O(n \log n)$**
- Best Case Time Complexity [Big-omega]:  **$O(n \log n)$**
- Average Time Complexity [Big-theta]:  **$O(n \log n)$**
- Space Complexity:  **$O(n)$**



**Example:**



## 2.1 : Write a program to implement Merge sort.

**Code:**

```
#include <stdio.h>

void mergeSort(int [], int, int, int);
void partition(int [],int, int);

int main()
{
    int list[50];
    int i, size;

    printf("Enter total number of elements:");
    scanf("%d", &size);
    printf("Enter the elements:\n");
    for(i = 0; i < size; i++)
    {
        scanf("%d", &list[i]);
    }
    partition(list, 0, size - 1);
    printf("After merge sort:\n");
    for(i = 0; i < size; i++)
    {
        printf("%d  ",list[i]);
    }

    return 0;
}

void partition(int list[],int low,int high)
{
    int mid;

    if(low < high)
    {
        mid = (low + high) / 2;
        partition(list, low, mid);
        partition(list, mid + 1, high);
    }
}
```

```

        mergeSort(list, low, mid, high);
    }
}

void mergeSort(int list[],int low,int mid,int high)
{
    int i, mi, k, lo, temp[50];

    lo = low;
    i = low;
    mi = mid + 1;
    while ((lo <= mid) && (mi <= high))
    {
        if (list[lo] <= list[mi])
        {
            temp[i] = list[lo];
            lo++;
        }
        else
        {
            temp[i] = list[mi];
            mi++;
        }
        i++;
    }
    if (lo > mid)
    {
        for (k = mi; k <= high; k++)
        {
            temp[i] = list[k];
            i++;
        }
    }
    else
    {
        for (k = lo; k <= mid; k++)
        {
            temp[i] = list[k];
            i++;
        }
    }

    for (k = low; k <= high; k++)

```

```
    {  
        list[k] = temp[k];  
    }  
}
```

## OUTPUT:



```
D:\C\ A_15_190770107097\PRACTICAL\bin\Debug\PRACTICAL\ A_15_190770107097.exe  
Enter total number of elements:5  
Enter the elements:  
17  
05  
3  
45  
477  
After merge sort:  
3 5 17 45 477  
Process returned 0 (0x0)   execution time : 11.678 s  
Press any key to continue.  
_
```

## Quick Sort:

Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot.

It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in an array results in an improved time complexity in most of the cases.

## Implementing Quick Sort Algorithm:

Select the first element of array as the pivot element First, we will see how the partition of the array takes place around the pivot.

- Choose the highest index value has pivot
- Take two variables to point left and right of the list excluding pivot
- left points to the low index
- right points to the high
- while value at left is less than pivot move right
- while value at right is greater than pivot move left
- if both step 5 and step 6 does not match swap left and right
- if  $\text{left} \geq \text{right}$ , the point where they met is newpivot

## Algorithm:

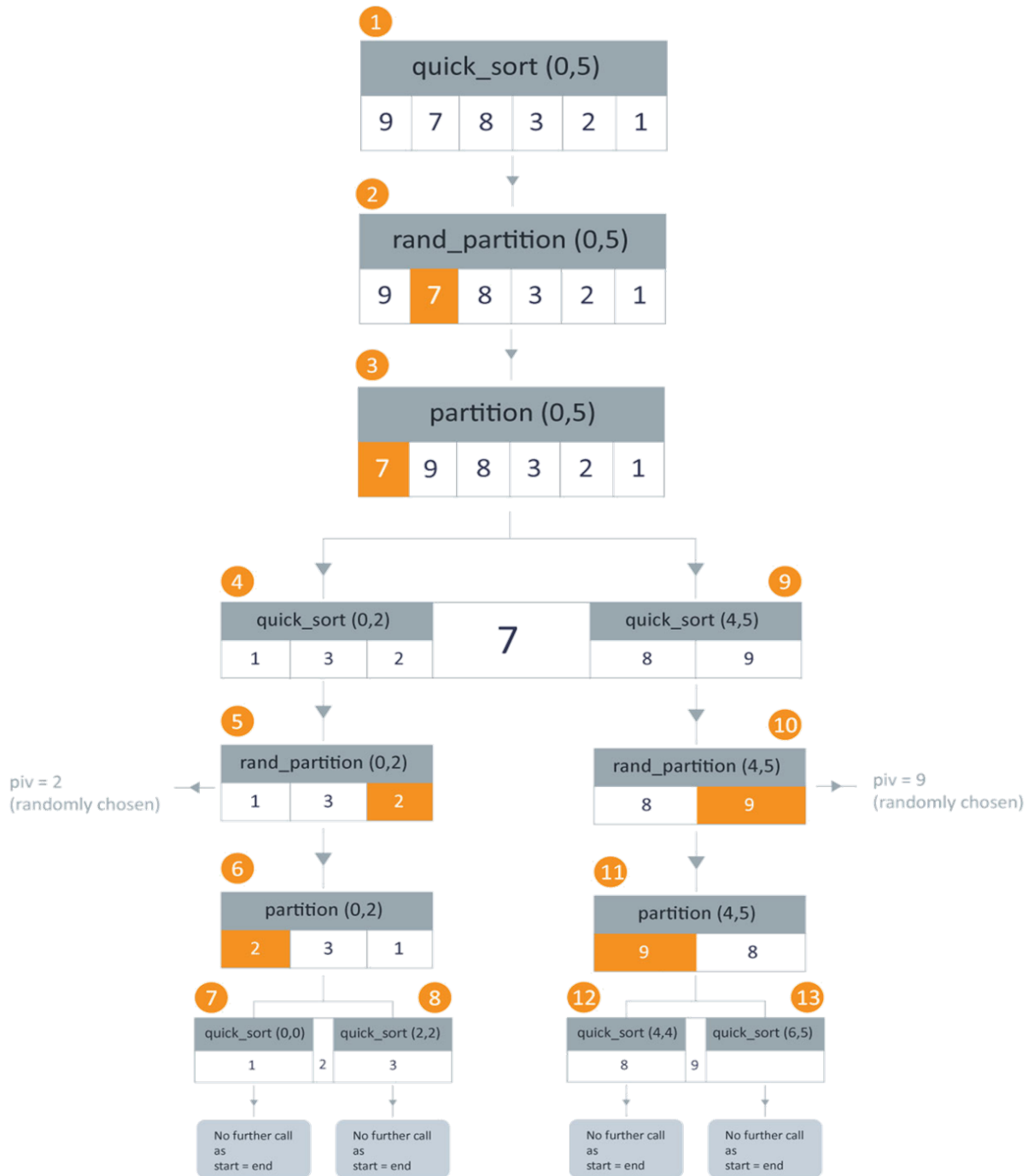
```
quickSort(left, right)
  if right-left <= 0
    return
  else
    pivot = A[right]
    partition = partitionFunc(left, right, pivot)
    quickSort(left,partition-1)
    quickSort(partition+1,right)
  end if
end procedure
```

## Complexity Analysis of Merge Sort:

The following will be the time and space complexity for merge sort algorithm:

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

### Example:



## 2.2 : Write a program to implement Quick sort.

### Code:

```
#include <stdio.h>

void quicksort (int [], int, int);

int main()
{
    int list[50];
    int size, i;

    printf("Enter the number of elements: ");
    scanf("%d", &size);
    printf("Enter the elements to be sorted:\n");
    for (i = 0; i < size; i++)
    {
        scanf("%d", &list[i]);
    }
    quicksort(list, 0, size - 1);
    printf("After applying quick sort\n");
    for (i = 0; i < size; i++)
    {
        printf("%d ", list[i]);
    }
    printf("\n");

    return 0;
}

void quicksort(int list[], int low, int high)
{
    int pivot, i, j, temp;
    if (low < high)
    {
        pivot = low;
        i = low;
        j = high;
        while (i < j)
        {
```

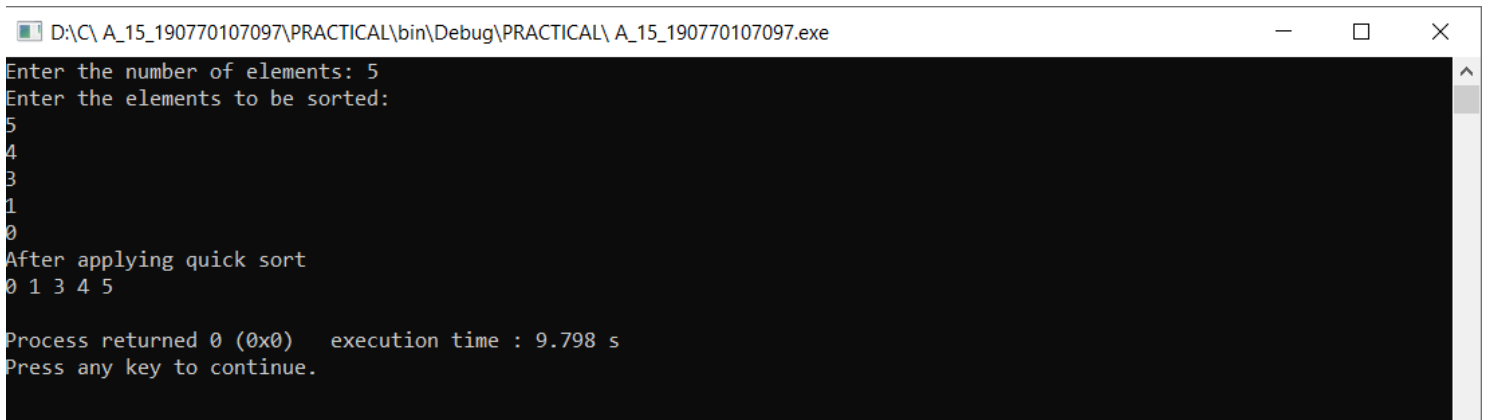
```

        while (list[i] <= list[pivot] && i <= high)
        {
            i++;
        }
        while (list[j] > list[pivot] && j >= low)
        {
            j--;
        }
        if (i < j)
        {
            temp = list[i];
            list[i] = list[j];
            list[j] = temp;
        }
    }

    temp = list[j];
    list[j] = list[pivot];
    list[pivot] = temp;
    quicksort(list, low, j - 1);
    quicksort(list, j + 1, high);
}

```

## OUTPUT:



```

D:\C\A_15_190770107097\PRACTICAL\bin\Debug\PRACTICAL\A_15_190770107097.exe
Enter the number of elements: 5
Enter the elements to be sorted:
5
4
3
1
0
After applying quick sort
0 1 3 4 5

Process returned 0 (0x0)   execution time : 9.798 s
Press any key to continue.

```



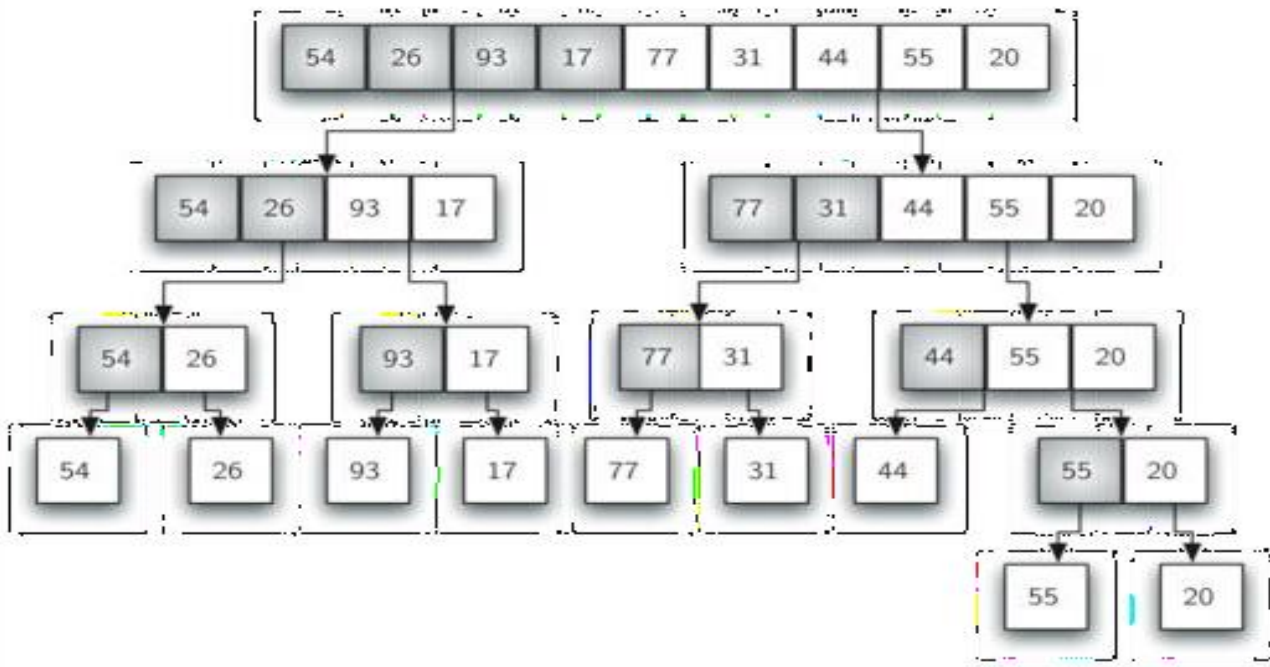
**Solve the below mention questionnaires:**

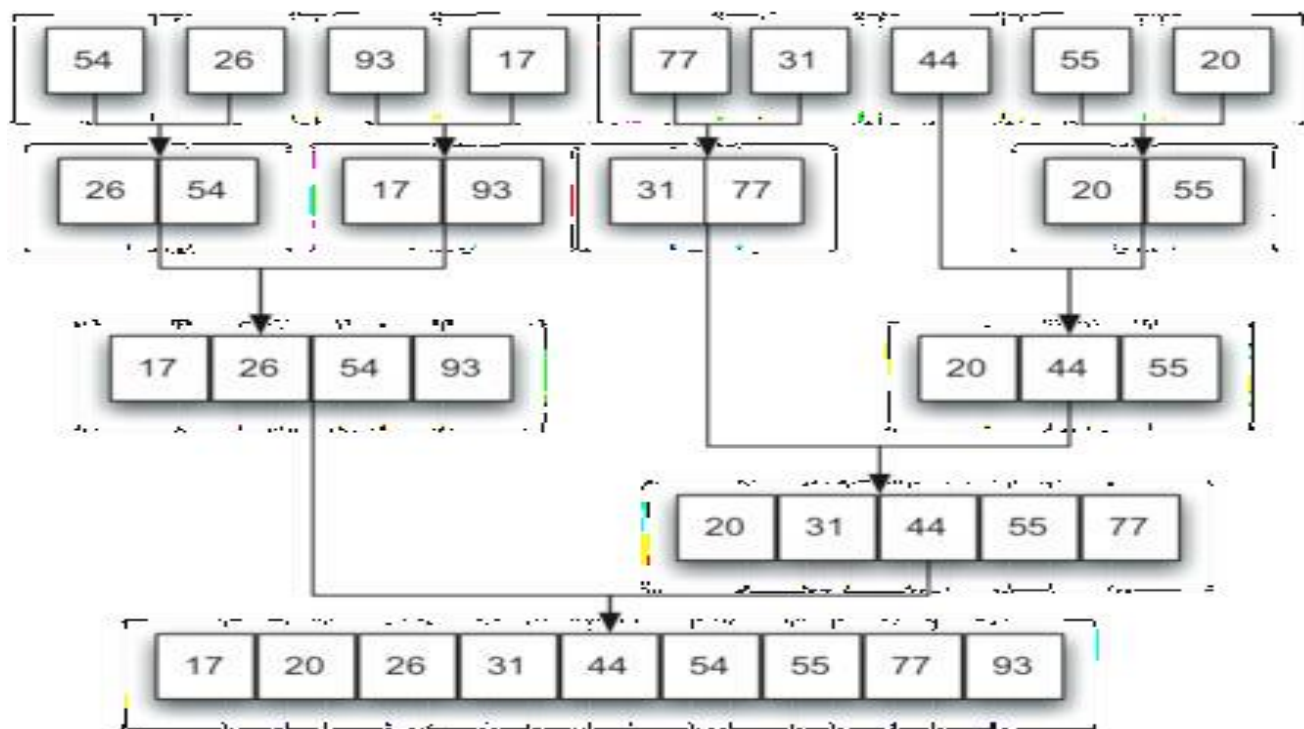
- 1) For the improvement of efficiency of quick sort the pivot can be
  - a) the first element
  - b) the mean element**
  - c) the last element
  - d) None of the above

**Answer (B)**
  
- 2) Which of the following is example of in-place algorithm?
  - a) **Merge Sort**
  - b) BubbleSort
  - c) Insertion Sort
  - d) Quick Sort

**Answer: ( A )**
  
- 3) Quick Sort can be categorized into which of the following?
  - a) Brute Force technique
  - b) Dynamic programming
  - c) Greedy algorithm
  - d) Divide and conquer**

**Answer: ( D )**
  
- 4) Perform the Merge Sort operation on the given list.  
 54, 26, 93, 17, 77, 31, 44, 55, 20





### **PRACTICAL SET - 3**

**AIM: Implementation and Time analysis of linear and binary search algorithm.**

#### **Linear Search Algorithm:**

Linear search algorithm finds given element in a list of elements with  $O(n)$  time complexity where  $n$  is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

#### **Implementing Linear Search Algorithm:**

- Read the search element from the user
- Compare, the search element with the first element in the list.
- If both are matching, then display "Given element found!!!" and terminate the function
- If both are not matching, then compare search element with the next element in the list.
- Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

#### **Algorithm:**

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

#### **Complexity Analysis of Linear Search:**

- Worst Case Time Complexity [ Big-O ]:  $O(n)$
- Best Case Time Complexity [Big-omega]:  $O(1)$
- Average Time Complexity [Big-theta]:  $O(n)$
- Space Complexity:  $O(1)$

### 3.1 : Write a program to implement Linear Search.

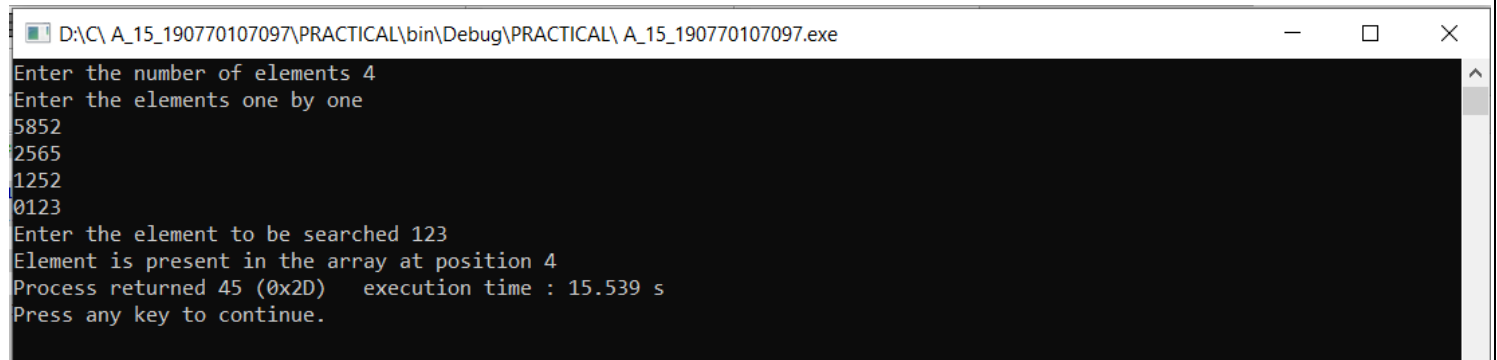
**Code:**

```
#include <stdio.h>

void main()
{
    int num;
    int i, keynum, found = 0;
    printf("Enter the number of elements ");
    scanf("%d", &num);
    int array[num];
    printf("Enter the elements one by one \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }

    printf("Enter the element to be searched ");
    scanf("%d", &keynum);
    for (i = 0; i < num ; i++)
    {
        if (keynum == array[i] )
        {
            found = 1;
            break;
        }
    }
    if (found == 1)
        printf("Element is present in the array at position %d",i+1);
    else
        printf("Element is not present in the array\n");
}
```

## Output:



```
D:\C\A_15_190770107097\PRACTICAL\bin\Debug\PRACTICAL\A_15_190770107097.exe
Enter the number of elements 4
Enter the elements one by one
5852
2565
1252
0123
Enter the element to be searched 123
Element is present in the array at position 4
Process returned 45 (0x2D) execution time : 15.539 s
Press any key to continue.
```

### Binary Search Algorithm:

The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in a order. The binary search cannot be used for list of element which are in random order.

This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element.

We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

### Implementing Binary Search Algorithm:

- Read the search element from the user
- Find the middle element in the sorted list
- Compare, the search element with the middle element in the sorted list.
- If both are matching, then display "Given element found!!!" and terminate the function
- If both are not matching, then check whether the search element is smaller or larger than middle element.
- If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- Repeat the same process until we find the search element in the list or until sublist contains only one element.
- If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

### Algorithm:

Procedure binary\_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exist.

```
set midPoint = (lowerBound + upperBound ) / 2

if A[midPoint] < x
    set lowerBound = midPoint + 1

if A[midPoint] > x
    set upperBound = midPoint - 1

if A[midPoint] == x
    EXIT: x found at location midPoint
end while
end procedure
```

### Complexity Analysis of Binary Search:

- Worst Case Time Complexity [ Big-O ]: **O(logn)**
- Best Case Time Complexity [Big-omega]:**O(1)**
- Average Time Complexity [Big-theta]: **O(logn)**
- Space Complexity: **O(1)**

### 3.2 : Write a program to implement Binary SearchAlgorithm.

#### Code:

```
#include <stdio.h>

void binary_search(int array[], int first, int last, int n)
{
    int i ,middle;
    middle = (first + last) / 2;

    while (first <= last)
    {
        if (array[middle] < n)
            first = middle + 1;
        else if (array[middle] == n)
        {
            printf("%d found at location %d.\n", n, middle+1);
            break;
        }
        else
            last = middle - 1;

        middle = (first + last) / 2;
    }
    if ( first > last )
        printf("Not found! %d is not present in the list.\n", n);
}
```

```

    }
    search(int arr[], int size, int data)
    {
        int p = (size - 1) / 2, low, high, a1 = 0, a2 = 1, i = 1;
        low = p + a1;
        high = p + a2;
        while(i)
        {
            if(data >= arr[low] && data <= arr[high])
            {
                binary_search(arr, low, high, data);
                break;
            }
            else if(data < arr[low])
            {
                binary_search(arr, 0, low, data);
                break;
            }
            else
            {
                a2 = a2 * 2;
                low = high;
                high = p + a2;
            }
        }
    }
}
int main()
{
    int a[200], i, j, n, size;
    printf("Enter the size of the list:");
    scanf("%d", &size);
    printf("Enter %d Integers in ascending order\n", size);
    for (i = 0; i < size; i++)
        scanf("%d", &a[i]);
    printf("Enter value to find\n");
    scanf("%d", &n);
    search(a, size, n );
    return 0;
}

```

## OUTPUT :

```

D:\C\A_15_190770107097\PRACTICAL\bin\Debug\PRACTICAL\A_15_190770107097.exe
Enter the size of the list:5
Enter 5 Integers in ascending order
45
46
47
48
49
Enter value to find
47
47 found at location 3.

Process returned 0 (0x0)   execution time : 17.326 s
Press any key to continue.

```



**Solve the below mention questionnaires:**

1) The Number of comparisons done by sequential search is?

- a).  $(N/2)+1$
- b).  $(N+1)/2$
- c).  $(N-1)/2$
- d).  $(N-2)/2$

**Answer: (b)**

2) What are the applications of binary search?

- a). To find the lower/upper bound in an ordered sequence
- b). Union of intervals
- c). Debugging
- d). All of thementioned

**Answer: (d)**

3) Binary search algorithm cannot be applied to ...

- a). sorted linked list
- b). sorted binary trees
- c). sorted linear array
- d). pointer array

**Answer: (a)**

**Conclusion / Outcome:**

So, we learned about the performance effects of linear search and binary search on ordered arrays. Linear search is slower due to checking the desired value to each data point in the array one by one. Contrast this to binary search, which cuts the search time by getting the middle value and going higher or lower depending on the desired value.

## PRACTICAL SET – 4

### AIM: Implementation of max-heap sort algorithm.

**Explanation:** Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining elements.

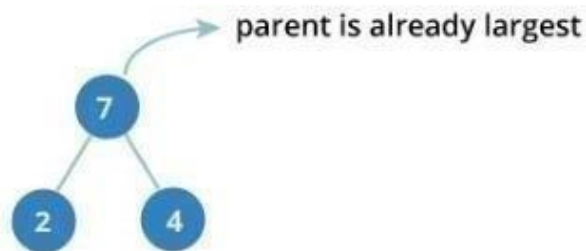
Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

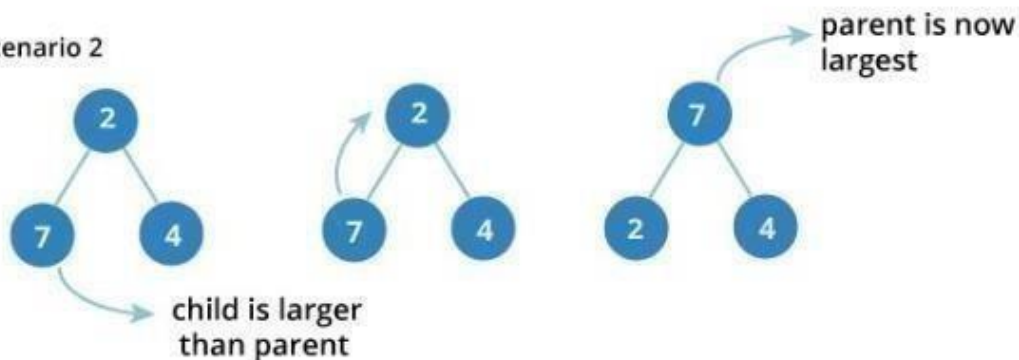
How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Scenario 1



Scenario 2



**Code:**

```
#include<stdio.h>
int temp;

void heapify(int arr[], int size, int i)
{
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < size && arr[left] > arr[largest])
        largest = left;

    if (right < size && arr[right] > arr[largest])
        largest = right;

    if (largest != i)
    {
        temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr, size, largest);
    }
}

void heapSort(int arr[], int size)
{
    int i;
    for (i = size / 2 - 1; i >= 0; i--)
        heapify(arr, size, i);
    for (i = size - 1; i >= 0; i--)
    {
        temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        heapify(arr, i, 0);
    }
}

void main()
{
    int arr[] = {1, 10, 2, 3, 4, 1};
```

```
        int i;  
        int size = sizeof(arr)/sizeof(arr[0]);  
  
        heapSort(arr, size);  
  
        printf("printing sorted elements\n");  
        for (i=0; i<size; ++i)  
            printf("%d\n",arr[i]);  
    }
```

### Output:

"D:\C\A\_15\_190770107097 PRACTICAL\bin\Debug\A\_15\_190770107097 PRACTICAL.exe"

printing sorted elements

1  
1  
2  
3  
4  
10

Process returned 6 (0x6) execution time : 0.024 s

Press any key to continue.

**Solve the below mention questionnaires:**

1. In a max-heap, element with the greatest key is always in which node?
- a) Leaf node
  - b) First node of left sub tree
  - c) **Root node**
  - d) First node of right sub tree

**Answer : ( C )**

2. Heap can be used as...
- a) **Priority queue**
  - b) Stack
  - c) A decreasing order array
  - d) None of the mentioned

**Answer : ( A )**

3. An array consists of  $n$  elements. We want to create a heap using the elements. The time complexity of building a heap will be in order of
- a)  $O(n*n*\log n)$
  - b)  **$O(n*\log n)$**
  - c)  $O(n*n)$
  - d)  $O(n * \log n * \log n)$

**Answer : ( B )**

4. The max heap constructed from the list of numbers 30, 10, 80, 60, 15, 55 is
- a) **80, 55, 60, 15, 10, 30**
  - b) 60, 80, 55, 30, 10, 15
  - c) 80, 60, 55, 30, 10, 15
  - d) None

**Answer : ( A )**

5. Always heap is a
- a) Binary search tree
  - b) Full binary tree
  - c) None
  - d) **Complete binary tree**

**Answer : ( D )**

### **Conclusion / Outcome:**

The primary advantage of heap tree is its efficiency. The execution time efficiency of the heap sort is  $O(n\log n)$ . The memory efficiency of the heap sort, unlike the other  $n \log n$  sorts is constant,  $O(1)$  because the heap sort algorithm is not recursive

## **DYNAMIC PROGRAMMING**

Dynamic programming is a method for solving a complex problem by breaking it down into simpler sub-problems, solving each of those sub-problems just once, and storing their solutions – in an array usually.

Now, every time the same sub-problem occurs, instead of recomputing its solution, the previously calculated solutions are used, thereby saving computation time at the expense of storage space.

Dynamic programming can be implemented in two ways –

**Memoization** – Memoization uses the top-down technique to solve the problem i.e. it begins with the original problem then breaks it into sub-problems and solves these sub-problems in the same way.

In this approach, you assume that you have already computed all sub-problems. You typically perform a recursive call (or some iterative equivalent) from the main problem. You ensure that the recursive call never recomputes a sub-problem because you cache the results, and thus duplicate sub-problems are not recomputed.

**Tabulation** – Tabulation is the typical Dynamic Programming approach. Tabulation uses the bottom up approach to solve the problem, i.e., by solving all related sub-problems first, typically by storing the results in an array. Based on the results stored in the array, the solution to the “top” / original problem is then computed.

Memoization and tabulation are both storage techniques applied to avoid recomputation of a sub-problem

The idea behind dynamic programming, in general, is to solve a given problem, by solving different parts of the problem (sub-problems), then using the cached solutions of the sub-problems to reach an overall solution.

**PRACTICAL – 5**

**AIM : Implementation of a knapsack problem using dynamic programming.**

**Explanation :** The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a profit value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total profit value is as large as possible. This is a 0-1 knapsack problem hence we can either take an entire item or reject it completely. We cannot break an item and fill the knapsack.

- In this problem we have a Knapsack that has a weight limit  $W$ , which represents knapsack capacity.
- Given two integer arrays  $val[0..n-1]$  and  $wt[0..n-1]$  which represent values and weights associated with  $n$  items respectively.
- Find out the maximum value subset of  $val[]$  such that sum of the weights of this subset is smaller than or equal to  $W$ .

**Code:**

```
#include <stdio.h>
```

```
int max(int a, int b) { return (a > b)? a : b; }
```

```
int knapsack(int W, int wt[], int val[], int n)
```

```
{
```

```
    int i, w;
```

```
    int K[n+1][W+1];
```

```
    for (i = 0; i <= n; i++)
```

```
    {
```

```
        for (w = 0; w <= W; w++)
```

```
        {
```

```
            if (i==0 || w==0)
```

```
                K[i][w] = 0;
```

```

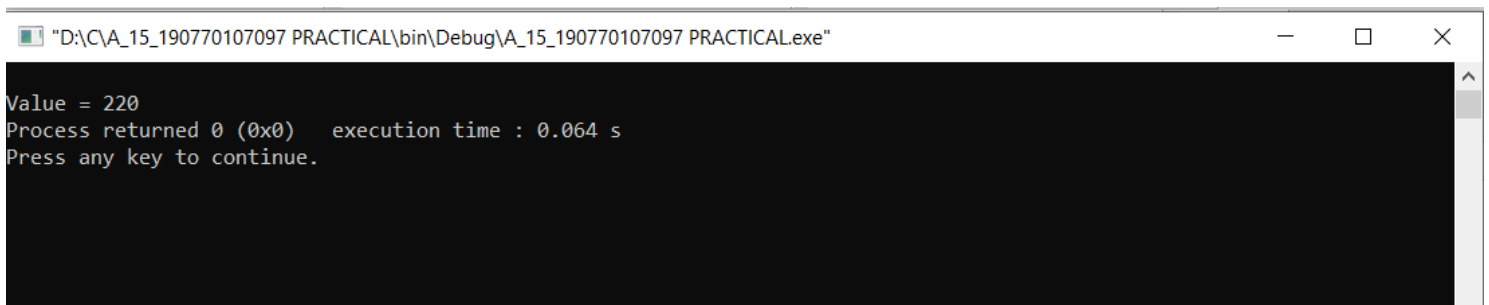
        else if (wt[i-1] <= w)
            K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);
        else
            K[i][w] = K[i-1][w];
    }
}

return K[n][W];
}

int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("\nValue = %d", knapsack(W, wt, val, n));
    return 0;
}

```

**Output:**



The screenshot shows a Windows command prompt window with the title bar "D:\C\A\_15\_190770107097 PRACTICAL\bin\Debug\A\_15\_190770107097 PRACTICAL.exe". The window contains the following text:

```

Value = 220
Process returned 0 (0x0)   execution time : 0.064 s
Press any key to continue.

```



**Solve the below mention questionnaires:**

1. You are given a knapsack that can carry a maximum weight of 60. There are 4 items with weights {20, 30, 40, 70} and values {70, 80, 90, 200}. What is the maximum value of the items you can carry using the knapsack?

a) 160  
b) 200  
c) 170  
d) 90

**Answer : (a)**

2. What is the time complexity of the above dynamic programming implementation of the Knapsack problem with n items and a maximum weight of W?

a)  $O(n)$   
b)  $O(n + w)$   
c)  $O(nW)$   
d)  $O(n^2)$

**Answer : (c)**

3. What is the space complexity of the above dynamic programming implementation of the Knapsack problem with n items and a maximum weight of W?

a)  $O(n)$   
b)  $O(nW)$   
c)  $O(n + W)$   
d)  $O(n^2)$

**Answer : (b)**

4. 0/1 knapsack is based on

a) Greedy method  
b) Branch and bound method  
c) Dynamic programming  
d) Divide and conquer

**Answer : ( c )**

5. Bin-packing problem is the application of

a) Knapsack  
b) Branch and bound  
c) Back tracking  
d) Dynamic programming

**Answer : (d)**

**Conclusion / Outcome:.**

**We conclude that in 0/1 knapsack problem . In this problem, a whole item can be selected (1) or a whole item can't be selected (0) or a fraction of item can also be selected (between 0 or 1)**

**PRACTICAL – 6**

**AIM : Implementation of chain matrix multiplication using dynamic programming.**

**Explanation:** Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

$$(ABC) D = (AB) (CD) = A (BCD) = \dots$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a  $10 \times 30$  matrix, B is a  $30 \times 5$  matrix, and C is a  $5 \times 60$  matrix. Then,

$$(AB) C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operations}$$

$$A (BC) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operations.}$$

Clearly the first parenthesization requires less number of operations.

Given an array  $p[]$  which represents the chain of matrices such that the  $i$ th matrix  $A_i$  is of dimension  $p[i-1] \times p[i]$ . We need to write a function `MatrixChainOrder()` that should return the minimum number of multiplications needed to multiply the chain.

This problem can be solved using Dynamic Programming. First we will compute results for sub-chains of the original chain and store these results in a 2-D array. Then we will use these results to compute results for larger chains.

**Code:**

```
#include<conio.h>
#include <stdio.h>
long int m[20][20];
int s[20][20];
int p[20],i,j,n;

void print_optimal(int i,int j)
{
    if (i == j)
    {
        printf(" A%d ",i);
    }
    else
    {
        printf("( ");
        print_optimal(i, s[i][j]);
        print_optimal(s[i][j] + 1, j);
        printf(")");
    }
}

void matmultiply(void)
{
    long int q;
    int k;
    for(i=n;i>0;i--)
    {
        for(j=i;j<=n;j++)
        {
            if(i==j)
                m[i][j]=0;
            else
            {
                for(k=i;k<j;k++)
                {
                    q=m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j];
                    if(q<m[i][j])
                    {
                        m[i][j]=q;
                        s[i][j]=k;
                    }
                }
            }
        }
    }
}

int MatrixChainOrder(int p[], int i, int j)
{

```

```

        if(i == j)
            return 0;
        int k;
        int min = INT_MAX;
        int count;

        for (k = i; k < j; k++)
        {
            count = MatrixChainOrder(p, i, k) +
                    MatrixChainOrder(p, k+1, j) +
                    p[i-1]*p[k]*p[j];

            if (count < min)
                min = count;
        }
        return min;
    }

void main()
{
    int k;
    printf("Enter the no. of elements: ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    for(j=i+1;j<=n;j++)
    {
        m[i][i]=0;
        m[i][j]=INFY;
        s[i][j]=0;
    }
    printf("\nEnter the dimensions: \n");
    for(k=0;k<=n;k++)
    {
        printf("P%d: ",k);
        scanf("%d",&p[k]);
    }
    matmultiply();
    printf("\nCost Matrix M:\n");
    for(i=1;i<=n;i++)
    for(j=i;j<=n;j++)
    printf("m[%d][%d]: %ld\n",i,j,m[i][j]);
    i=1,j=n;
    printf("\nMultiplication Sequence : ");
    print_optimal(i,j);
    printf("\nMinimum number of multiplications is : %d ",
           MatrixChainOrder(p, 1, n));
}

```

## OUTPUT

"D:\C\A\_15\_190770107097 PRACTICAL\bin\Debug\A\_15\_190770107097 PRACTICAL.exe"

Enter the no. of elements: 4

Enter the dimensions:

P0: 1

P1: 2

P2: 3

P3: 4

P4: 5

Cost Matrix M:

m[1][1]: 0

m[1][2]: 6

m[1][3]: 18

m[1][4]: 38

m[2][2]: 0

m[2][3]: 24

m[2][4]: 64

m[3][3]: 0

m[3][4]: 60

m[4][4]: 0

Multiplication Sequence : ( ( ( A1 A2 ) A3 ) A4 )

Minimum number of multiplications is : 38

Process returned 43 (0x2B) execution time : 8.830 s

Press any key to continue.

**Solve the below mention questionnaires :**

1. Consider the two matrices P and Q which are 10 x 20 and 20 x 30 matrices respectively. What is the number of multiplications required to multiply the two matrices?

a)  $10 * 20$

c)  $20 * 30$

b)  $10 * 30$

d)  $10 * 20 * 30$

Answer : (d)

2. What is the time complexity of the above dynamic programming implementation of the matrix chain problem?

a)  $O(1)$

c)  $O(n^2)$

b)  $O(n)$

d)  $O(n^3)$

Answer : (d)

3. Which of the following methods can be used to solve the matrix chain multiplication problem?

a) Dynamic programming

c) Recursion

b) Brute force

d) All of the mentioned

Answer : (d)

4. Consider the matrices P, Q and R which are 10 x 20, 20 x 30 and 30 x 40 matrices respectively. What is the minimum number of multiplications required to multiply the three matrices?

a) 18000

c) 24000

b) 12000

d) 32000

Answer : (a)



## Education to Innovation



- d) 12000

**Answer : (c)**

**Conclusion / Outcome:**

**From the above discussion we can say that the proposed matrix chain multiplication algorithm using Dynamic Programming in the best case and average case takes  $O(n^2)$  time complexity which is less when it is compared with existing matrix chain multiplication which takes  $O(n^3)$ . Hence the time complexity is reduced with the space requirement of  $O(n^2)$ .**

### Practical Set-7

**AIM: Implementation of making a change problem using dynamic programming.**

**Explanation:**

Given a value N, if we want to make change for N cents, and we have infinite supply of each of  $S = \{ S_1, S_2, \dots, S_m \}$  valued coins, how many ways can we make the change? The order of coins doesn't matter.

For example, for  $N = 4$  and  $S = \{1, 2, 3\}$ , there are four solutions:  $\{1, 1, 1, 1\}, \{1, 1, 2\}, \{2, 2\}, \{1, 3\}$ . So output should be 4. For  $N = 10$  and  $S = \{2, 5, 3, 6\}$ , there are five solutions:  $\{2, 2, 2, 2, 2\}, \{2, 2, 3, 3\}, \{2, 2, 6\}, \{2, 3, 5\}$  and  $\{5, 5\}$ . So the output should be 5

**Code:**

```
#include <stdio.h>
int main ()
{
    int num_denominations, coin_list[100], use_these[100], i, owed;
    printf("Enter number of denominations : ");
    scanf("%d", &num_denominations);
    printf("\nEnter the denominations in descending order: ");
    for(i=0; i< num_denominations; i++)
    {
        scanf("%d", &coin_list[i]);
    }

    printf("\nEnter the amount owed : ");
    scanf("%d", &owed);

    for(i=0; i < num_denominations; i++)
    {
        use_these[i] = owed / coin_list[i];
        owed %= coin_list[i];
    }

    printf("\nSolution: \n");
    for(i=0; i < num_denominations; i++)
    {
        printf("%dx%d ", coin_list[i], use_these[i]);
    }
}
```



## Output:

```
"D:\C\A_15_190770107097 PRACTICAL\bin\Debug\A_15_190770107097 PRACTICAL.exe"
Enter number of denominations : 3
Enter the denominations in descending order: 60
15
10
Enter the amount owed : 40
Solution:
60x0 15x2 10x1
Process returned 0 (0x0)   execution time : 12.721 s
Press any key to continue.
```

**Give the Answer to the below Short Questions:**

1. **What is time complexity of making change using dynamic programming?**

**Time Complexity:  $O(n^m)n$**

2. If  $S = 5$  and  $N = \{1,2,3\}$ , means we have a sum of 5 and want to make change for this sum using coins of denomination 1,2 and 3. There is infinite supply of these coins. We have to find in how many ways we can make this change.

In this we will take 3 one time and 2 one time to make a sum of 5

**Conclusion / Outcome:**

The change-making problem addresses the question of finding the minimum number of coins (of certain denominations) that add up to a given amount of money. It is a special case of the integer knapsack problem, and has applications wider than just currency.

### Practical Set- 8

#### AIM: Implementation of a knapsack problem using greedy algorithm.

##### Description:

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a subproblem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

##### Problem Scenario

A thief is robbing a store and can carry a maximal weight of  $W$  into his knapsack. There are  $n$  items available in the store and weight of  $i^{th}$  item is  $w_i$  and its profit is  $p_i$ . What items should the thief take?

In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.

Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

##### Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.

According to the problem statement,

- There are  $n$  items in the store
- Weight of  $i^{th}$  item  $w_i > 0$
- Profit for  $i^{th}$  item  $p_i > 0$  and
- Capacity of the Knapsack is  $W$

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction  $x_i$  of  $i^{th}$  item.

$$0 \leq x_i \leq 1$$

The  $i^{\text{th}}$  item contributes the weight  $x_i \cdot w_i$  to the total weight in the knapsack and profit  $x_i \cdot p_i$  to the total profit.

Hence, the objective of this algorithm is to

$$\text{maximize } \sum_{i=1}^n (x_i \cdot p_i) \quad \text{maximize } \sum_{i=1}^n (x_i \cdot p_i)$$

subject to constraint,

$$\sum_{i=1}^n (x_i \cdot w_i) \leq W$$

It is clear that an optimal solution must fill the knapsack exactly, otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{i=1}^n (x_i \cdot w_i) = W$$

In this context, first we need to sort those items according to the value of  $p_i/w_i$ , so that  $p_{i+1}/w_{i+1} \leq p_i/w_i$ . Here,  $x$  is an array to store the fraction of items.

**Algorithm:** Greedy-Fractional-Knapsack ( $w[1..n]$ ,  $p[1..n]$ ,  $W$ )

```

for i = 1 to n
  do  $x[i] = 0$ 
weight = 0
for i = 1 to n
  if  $\text{weight} + w[i] \leq W$  then
     $x[i] = 1$ 
     $\text{weight} = \text{weight} + w[i]$ 
  else
     $x[i] = (W - \text{weight}) / w[i]$ 
     $\text{weight} = W$ 
    break
return x

```

## Analysis

If the provided items are already sorted into a decreasing order of  $p_i/w_i$ , then the whileloop takes a time in  $O(n)$ ; Therefore, the total time including the sort is in  $O(n \log n)$ .

### Example

Let us consider that the capacity of the knapsack  $W = 60$  and the list of provided items are shown in the following table –

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio $(p_i)/(w_i)$	7	10	6	5

As the provided items are not sorted based on  $p_i/w_i$ . After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio $(p_i/w_i)$	10	7	6	5

### Solution

After sorting all the items according to  $p_i/w_i$ . First all of **B** is chosen as weight of **B** is less than the capacity of the knapsack. Next, item **A** is chosen, as the available capacity of the knapsack is greater than the weight of **A**. Now, **C** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of **C**.

Hence, fraction of **C** (i.e.  $(60 - 50)/20$ ) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more item can be selected.

The total weight of the selected items is  $10 + 40 + 20 * (10/20) = 60$

And the total profit is  $100 + 280 + 120 * (10/20) = 380 + 60 = 440$

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

**Code:**

```
# include<stdio.h>

void knapsack(int n, float weight[], float profit[], float capacity) {
    float x[20], tp = 0;
    int i, j, u;
    u = capacity;

    for (i = 0; i < n; i++)
        x[i] = 0.0;

    for (i = 0; i < n; i++)
    {
        if (weight[i] > u)
            break;
        else
        {
            x[i] = 1.0;
            tp = tp + profit[i]; u
            = u - weight[i];
        }
    }

    if (i < n)
        x[i] = u / weight[i];

    tp = tp + (x[i] * profit[i]);

    printf("\nThe result vector is:- ");
    for (i = 0; i < n; i++)
        printf("%f\t", x[i]); printf("\nMaximum

profit is:- %f", tp);

}

int main() {
    float weight[20], profit[20], capacity;
    int num, i, j;
    float ratio[20], temp;

    printf("\nEnter the no. of objects:- ");
    scanf("%d", &num);

    printf("\nEnter the wts and profits of each object:- "); for
    (i = 0; i < num; i++) {
        scanf("%f %f", &weight[i], &profit[i]);
    }

    printf("\nEnter the capacity of knapsack:- ");
    scanf("%f", &capacity);
```

```

for (i = 0; i < num; i++)
{
    ratio[i] = profit[i] / weight[i];
}

for (i = 0; i < num; i++)
{
    for (j = i + 1; j < num; j++)
    {
        if (ratio[i] < ratio[j])
        {
            temp = ratio[j];
            ratio[j] = ratio[i];
            ratio[i] = temp;

            temp = weight[j];
            weight[j] = weight[i];
            weight[i] = temp;

            temp = profit[j];
            profit[j] = profit[i];
            profit[i] = temp;
        }
    }
}

knapsack(num, weight, profit, capacity);
return(0);
}

```

## Output:

"D:\C\A\_15\_190770107097 PRACTICAL\bin\Debug\A\_15\_190770107097 PRACTICAL.exe"

Enter the no. of objects:- 5

Enter the wts and profits of each object:- 10 20

20 30

30 66

40 40

50 60

Enter the capacity of knapsack:- 100

The result vector is:- 1.000000 1.000000 1.000000 0.800000 0.000000

Maximum profit is:- 164.000000

Process returned 0 (0x0) execution time : 43.037 s

Press any key to continue.

**Give the Answer to the below Short Questions:**

1. The Knapsack problem is an example of  
a) Greedy algorithm  
b) **2D dynamic programming**  
c) 1D dynamic programming  
d) Divide and conquer  
**Answer: (b)**
2. Which of the following methods can be used to solve the Knapsack problem?  
a) Brute force algorithm  
b) Recursion  
c) Dynamic programming  
d) **All of the mentioned**  
**Answer: (d)**
3. You are given a knapsack that can carry a maximum weight of 60. There are 4 items with weights {20, 30, 40, 70} and values {70, 80, 90, 200}. What is the maximum value of the items you can carry using the knapsack?  
a) **160**  
b) 200  
c) 170  
d) 90  
**Answer: (a)**
4. Which of the following problems is equivalent to the 0-1 Knapsack problem?  
a) You are given a bag that can carry a maximum weight of  $W$ . You are given  $N$  items which have a weight of  $\{w_1, w_2, w_3, \dots, w_n\}$  and a value of  $\{v_1, v_2, v_3, \dots, v_n\}$ . You can break the items into smaller pieces. Choose the items in such a way that you get the maximum value.  
b) You are studying for an exam and you have to study  $N$  questions. The questions take  $\{t_1, t_2, t_3, \dots, t_n\}$  time (in hours) and carry  $\{m_1, m_2, m_3, \dots, m_n\}$  marks. You can study for a maximum of  $T$  hours. You can either study a question or leave it. Choose the questions in such a way that your score is maximized.  
c) You are given infinite coins of denominations  $\{v_1, v_2, v_3, \dots, v_n\}$  and a sum  $S$ . You have to find the minimum number of coins required to get the sum  $S$ .  
d) None of the mentioned  
**Answer: (b)**
5. The 0-1 Knapsack problem can be solved using Greedy algorithm.  
a) True  
b) **False**  
**Answer: (b)**



**Practical Set-9**

**AIM: Implementation of Graph and Searching (DFS)**

**Explanation:**

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

1. Visited
2. Not Visited

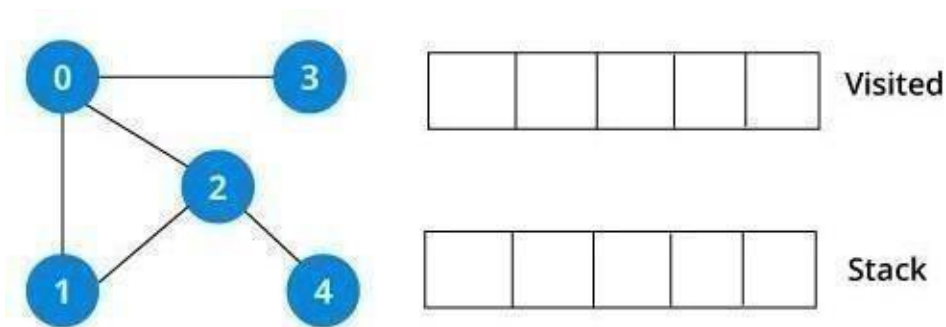
The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

**The DFS algorithm works as follows:**

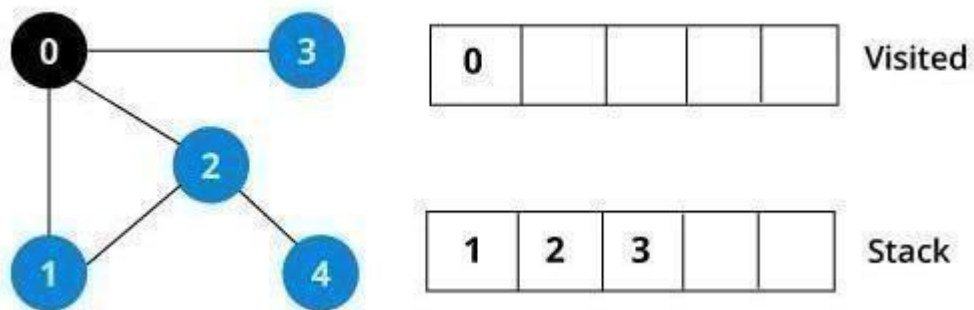
1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
4. Keep repeating steps 2 and 3 until the stack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

**DFS Example**

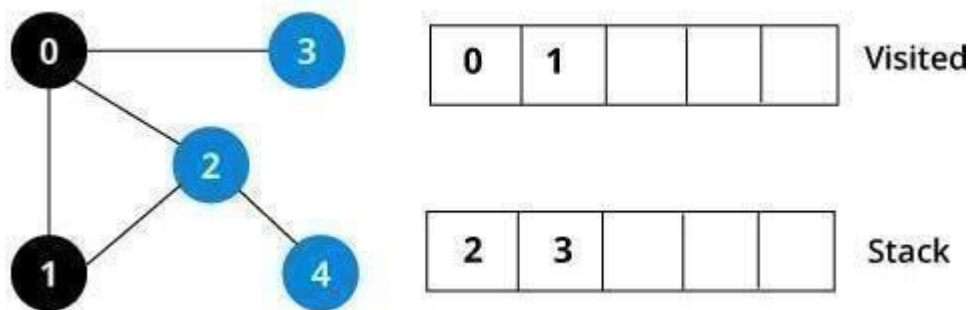
Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



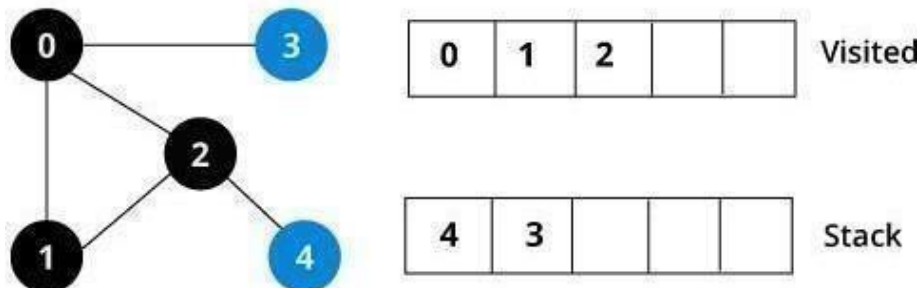
We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

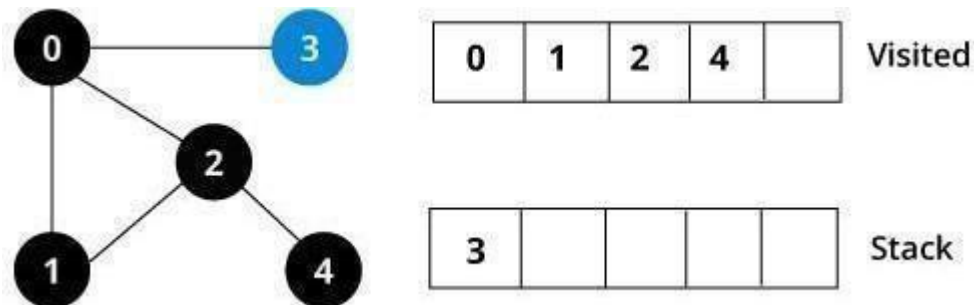


Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

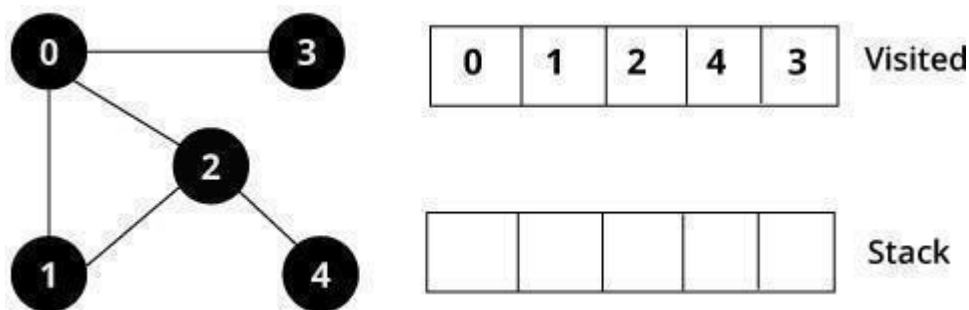


Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.





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.



### Code:

```
#include <stdio.h>
#include <stdlib.h>
int source,V,E,time,visited[20],G[20][20];
void DFS(int i)
{
    int j;
    visited[i]=1;
    printf(" %d->",i+1);
    for(j=0;j<V;j++)
    {
        if(G[i][j]==1&&visited[j]==0)
            DFS(j);
    }
}
int main()
{
    int i,j,v1,v2;
    printf("\t\t\tGraphs\n");
```

```

    printf("Enter the no of edges:");
    scanf("%d",&E);
    printf("Enter the no of vertices:");
    scanf("%d",&V);
    for(i=0;i<V;i++)
    {
        for(j=0;j<V;j++)
            G[i][j]=0;
    }

    for(i=0;i<E;i++)
    {
        printf("Enter the edges (format: V1 V2) : ");
        scanf("%d%d",&v1,&v2);
        G[v1-1][v2-1]=1;
    }

    for(i=0;i<V;i++)
    {
        for(j=0;j<V;j++)
            printf(" %d ",G[i][j]);
        printf("\n");
    }
    printf("Enter the source: ");
    scanf("%d",&source);
    DFS(source-1);
    return 0;
}

```

## Output;

"D:\C\A\_15\_190770107097 PRACTICAL\bin\Debug\A\_15\_190770107097 PRACTICAL.exe"

```
Graphs
Enter the no of edges:11
Enter the no of vertices:10
Enter the edges (format: V1 V2) : 1 2
Enter the edges (format: V1 V2) : 1 3
Enter the edges (format: V1 V2) : 2 4
Enter the edges (format: V1 V2) : 2 5
Enter the edges (format: V1 V2) : 3 6
Enter the edges (format: V1 V2) : 3 7
Enter the edges (format: V1 V2) : 4 8
Enter the edges (format: V1 V2) : 5 9
Enter the edges (format: V1 V2) : 6 10
Enter the edges (format: V1 V2) : 8 9
Enter the edges (format: V1 V2) : 9 10
0 1 1 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0
Enter the source: 1
1-> 2-> 4-> 8-> 9-> 10-> 5-> 3-> 6-> 7->
Process returned 0 (0x0)   execution time : 37.131 s
Press any key to continue.
```

### Give the Answer to Questions:

1. Depth First Search is equivalent to which of the traversal in the Binary Trees?  
a) Pre-order Traversal  
b) Post-order Traversal  
c) Level-order Traversal  
d) In-order Traversal  
**Answer: (a)**
2. Time Complexity of DFS is? ( $V$  – number of vertices,  $E$  – number of edges)  
a)  $O(V + E)$   
b)  $O(V)$   
c)  $O(E)$   
d) None  
**Answer: (a)**
3. The Depth First Search traversal of a graph will result into?  
a) Linked List  
b) Tree  
c) Graph with back edges  
d) None  
**Answer: (b)**
4. A person wants to visit some places. He starts from a vertex and then wants to visit every vertex till it finishes from one vertex, backtracks and then explore other vertex from same vertex. What algorithm he should use?  
a) Depth First Search  
b) Breadth First Search  
c) Trim's algorithm  
d) None  
**Answer: (a)**

**Conclusion/ Outcome:**

We have seen that The purpose of the algorithm is to mark each vertex as visited while avoiding cycles. Searching or tracking is one method to solve general problems, especially AI (Artificial Intelligence) problems. Searching is a process of looking for a solution of a problem through a set of possible state space (state space). Depth-First Search (DFS) algorithm is a method that included of blind search. That is, the search is done by ensuring all vertices have been visited, but without the exact solution or a suboptimal solution

❖  
**Practical Set-10**

## **AIM: Implementation of Graph and Searching (BFS)**

### **Explanation:**

Traversal means visiting all the nodes of a graph. Breadth first traversal or Breadth first Search is a recursive algorithm for searching all the vertices of a graph or tree data structure. In this article, you will learn with the help of examples the BFS algorithm, BFS pseudocode and the code of the breadth first search algorithm with implementation in C++, C, Java and Python programs.

### **BFS 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.

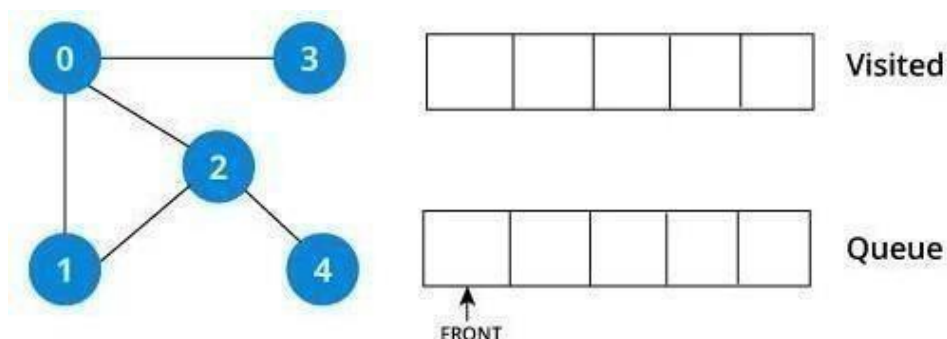
The algorithm works as follows:

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.

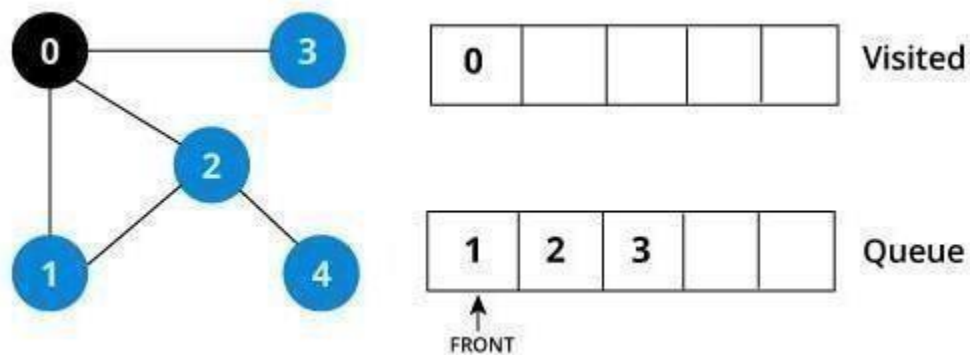
The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node

### **BFS example**

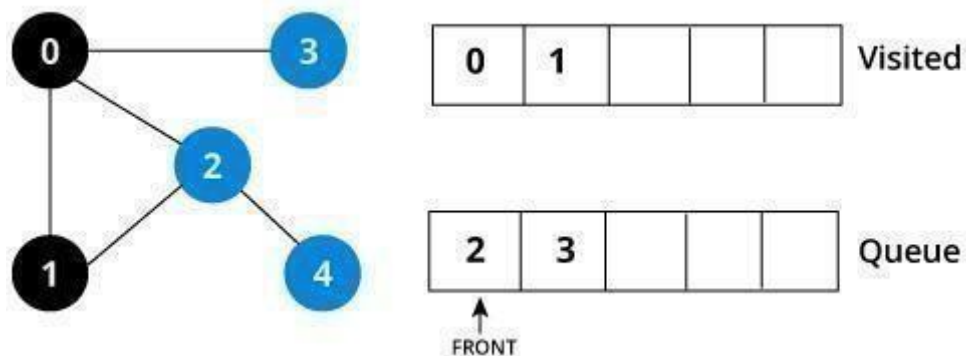
Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



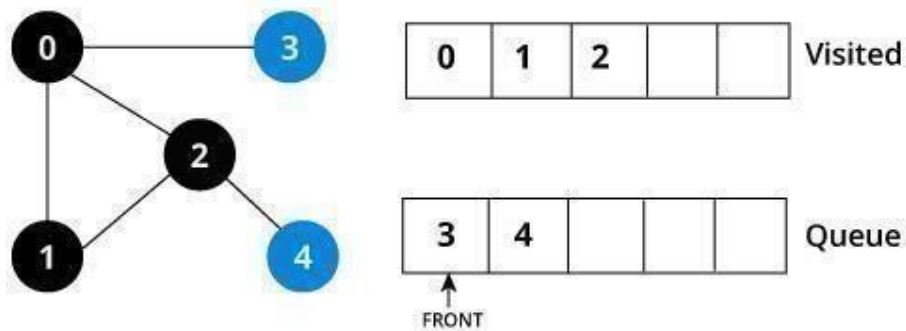
We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.



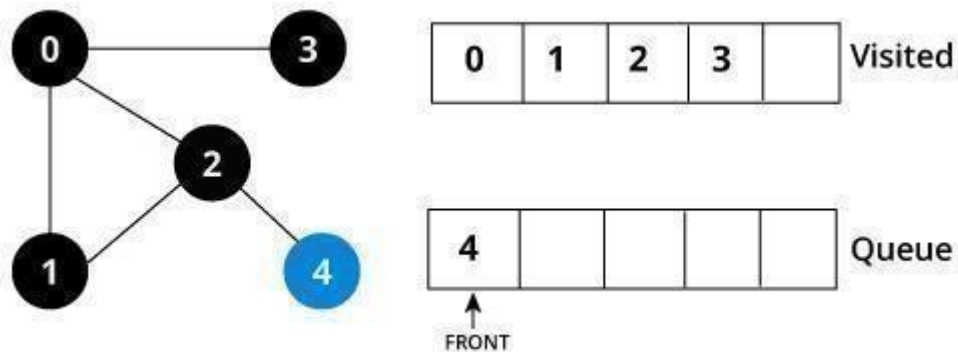
Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



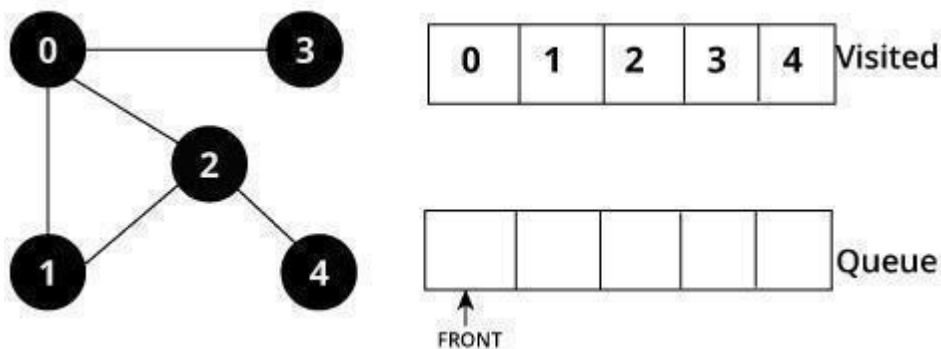
Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.







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



Since the queue is empty, we have completed the Depth First Traversal of the graph.

### Code:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue
{
    int items[SIZE];
    int front;
    int rear;
};
struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);

int isEmpty(struct queue* q);
void printQueue(struct queue* q);
```

```

struct node
{
    int vertex;
    struct node* next;
};
struct node* createNode(int);
struct Graph
{
    int numVertices;
    struct node** adjLists;
    int* visited;
};
void bfs(struct Graph* graph, int startVertex)
{
    struct queue* q = createQueue();
    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);
    while (!isEmpty(q))
    {
        printQueue(q);
        int currentVertex = dequeue(q);
        printf("Visited %d\n", currentVertex);
        struct node* temp = graph->adjLists[currentVertex];
        while (temp)
        {
            int adjVertex = temp->vertex;
            if (graph->visited[adjVertex] == 0)
            {
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
            }
            temp = temp->next;
        }
    }
}
struct node* createNode(int v)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int vertices)
{
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));
    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
}

```

```

        return graph;
    }
void addEdge(struct Graph* graph, int src, int dest)
{
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
struct queue* createQueue()
{
    struct queue* q = malloc(sizeof(struct queue));
    q->front = -1;
    q->rear = -1;
    return q;
}
int isEmpty(struct queue* q)
{
    if (q->rear == -1)
        return 1;
    else
        return 0;
}

void enqueue(struct queue* q, int value)
{
    if (q->rear == SIZE - 1)
        printf("\nQueue is Full!!");
    else
    {
        if (q->front == -1)
            q->front = 0;
        q->rear++;

        q->items[q->rear] = value;
    }
}
int dequeue(struct queue* q)
{
    int item;
    if (isEmpty(q))
    {
        printf("Queue is empty");
        item = -1;
    }
    else
    {
        item = q->items[q->front];
        q->front++;
        if (q->front > q->rear)
        {
            printf("Resetting queue ");
            q->front = q->rear = -1;
        }
    }
}

```

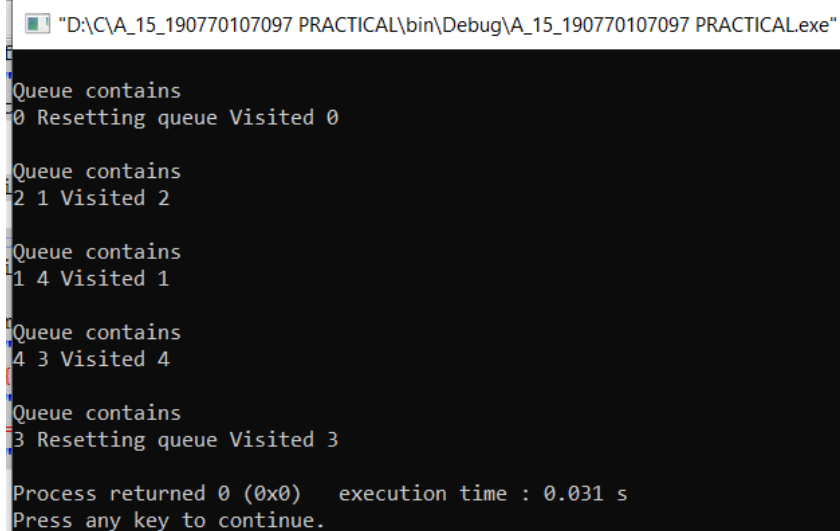
```

    }
    }
    return item;
}
void printQueue(struct queue* q)
{
    int i = q->front;
    if (isEmpty(q))
    {
        printf( "Queue is empty");
    }
    else
    {
        printf("\nQueue contains \n");
        for (i = q->front; i < q->rear + 1; i++)
        {
            printf("%d ", q->items[i]);
        }
    }
}

int main() {
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    bfs(graph, 0);
    return 0;
}

```

## Output:



```

"D:\C\A_15_190770107097 PRACTICAL\bin\Debug\A_15_190770107097 PRACTICAL.exe"
Queue contains
0 Resetting queue Visited 0

Queue contains
2 1 Visited 2

Queue contains
1 4 Visited 1

Queue contains
4 3 Visited 4

Queue contains
3 Resetting queue Visited 3

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.

```

### Give the Answer to the below Questions:

1. Breadth First Search is equivalent to which of the traversal in the Binary Trees?  
a) Pre-order Traversal                      b) Post-order Traversal  
c) Level-order Traversal                      d) In-order Traversal                      **Answer( c )**
2. Time Complexity of Breadth First Search is? (V – number of vertices, E – number of edges)  
a)  **$O(V + E)$**                       b)  $O(V)$   
c)  $O(E)$                       d) None                      **Answer: (A)**
3. The Data structure used in standard implementation of Breadth First Search is?  
a) Stack                      b) **Queue**  
c) Linked List                      d) None                      **Answer: (B)**
4. What can be the applications of Breadth FirstSearch?  
a) Finding shortest path between two nodes  
b) Finding bipartitions of a graph  
c) GPS navigation system  
d) All of the mentioned                      **Answer: (C )**
5. When the Breadth First Search of a graph is unique?  
a) When the graph is a Binary Tree  
b) When the graph is a Linked List  
c) When the graph is a n-ary Tree  
d) None of the mentioned                      **Answer: (A )**

### Conclusion/ Outcome:

Here we learn about best first search how to implement and use.

## Practical Set-11

### AIM: Implement prim's algorithm.

#### Explanation:

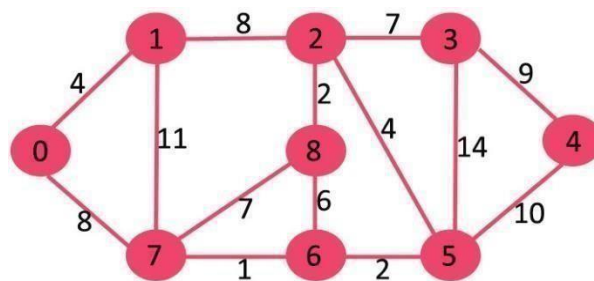
The idea behind Prim's algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a *Spanning Tree*. And they must be connected with the minimum weight edge to make it a *Minimum Spanning Tree*.

#### Algorithm

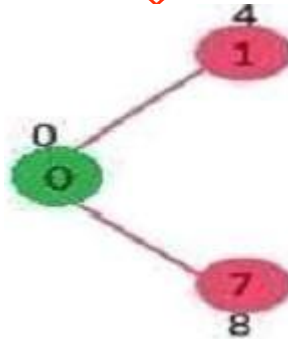
- 1) Create a set *mstSet* that keeps track of vertices already included in MST.
- 2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE.  
Assign key value as 0 for the first vertex so that it is picked first.
- 3) While *mstSet* doesn't include all vertices
  - a) Pick a vertex *u* which is not there in *mstSet* and has minimum keyvalue.
  - b) Include *u* to *mstSet*.
  - c) Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*.

The idea of using key values is to pick the minimum weight edge from cut. The key values are used only for vertices which are not yet included in MST, the key value for these vertices indicate the minimum weight edges connecting them to the set of vertices included in MST.

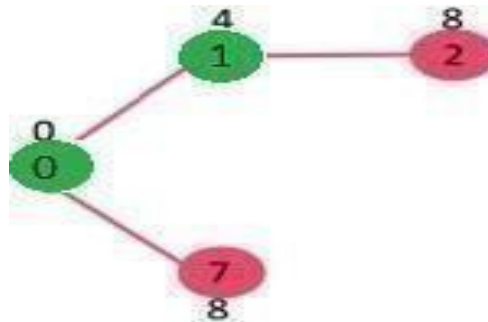
#### Example:



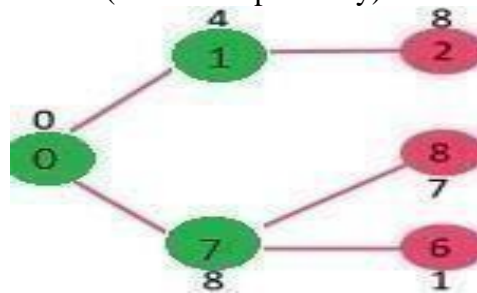
The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum key value. The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}. After including to *mstSet*, update key values of adjacent vertices. Adjacent vertices of 0 are 1 and 7. The key values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their key values, only the vertices with finite key values are shown. The vertices included in MST are shown in greencolor.



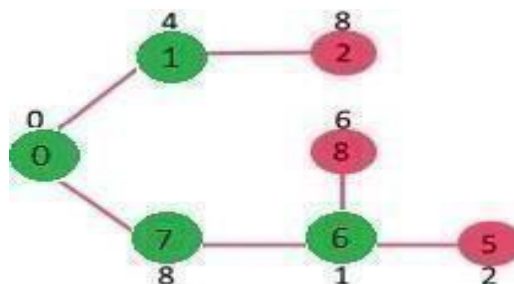
Pick the vertex with minimum key value and not already included in MST (not in mstSET). The vertex 1 is picked and added to mstSet. So mstSet now becomes {0, 1}. Update the key values of adjacent vertices of 1. The key value of vertex 2 becomes 8.



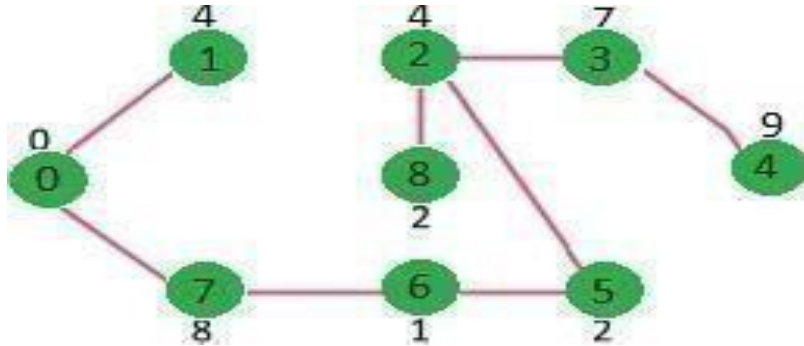
Pick the vertex with minimum key value and not already included in MST (not in mstSET). We can either pick vertex 7 or vertex 2, let vertex 7 is picked. So mstSet now becomes {0, 1, 7}. Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (7 and 1 respectively).



Pick the vertex with minimum key value and not already included in MST (not in mstSET). Vertex 6 is picked. So mstSet now becomes {0, 1, 7, 6}. Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



We repeat the above steps until *mstSet* includes all vertices of given graph. Finally, we get the following graph.



### Code:

```
#include<stdio.h>
#include<stdbool.h>
> #define INF
9999999
// number of vertices in
graph #define V 5
// create a 2d array of size 5x5
//for adjacency matrix to represent
graph int G[V][V] = {
{0, 9, 75, 0, 0},
{9, 0, 95, 19, 42},
{75, 95, 0, 51, 66},
{0, 19, 51, 0, 31},
{0, 42, 66, 31, 0}};
int main() {
int no_edge; // number of edge
// create a array to track selected vertex
// selected will become true otherwise
false int selected[V];
// set selected false initially
memset(selected, false,
sizeof(selected));
// set number of edge to
0 no_edge = 0;
// the number of egde in minimum spanning tree will be
// always less than (V -1), where V is number of vertices in
```



```

//graph
// choose 0th vertex and make it
true selected[0] = true;
int x; // row
number int y; // col
number
// print for edge and
weight printf("Edge :
Weight\n"); while
(no_edge < V - 1) {
//For every vertex in the set S, find the all adjacent vertices
// , calculate the distance from the vertex selected at step 1.
// if the vertex is already in the set S, discard it otherwise
//choose another vertex nearest to selected vertex at
step 1. int min = INF;
x = 0;
y = 0;
for (int i = 0; i < V;
i++) { if (selected[i]) {
for (int j = 0; j < V; j++) {
if (!selected[j] && G[i][j]) { // not in selected and there is an
edge if (min > G[i][j]) {
min = G[i][j];
x = i;
y = j;
}
}
}
}
}
printf("%d - %d : %d\n", x, y,
G[x][y]); selected[y] = true;
no_edge++;
}
return 0;

```

## Output:

```

D:\C\A_15_190770107097\bin\Debug\A_15_190770107097.exe
Edge : Weight
0 - 1 : 9
1 - 3 : 19
3 - 4 : 31
3 - 2 : 51

Process returned 0 (0x0)   execution time : 0.063 s
Press any key to continue.

```

## Give the Answer to the below Questions:

1. What is the time complexity to extract a vertex from the priority queue in Prim's algorithm?  
a)  $\log(V)$  c)  $V.V$   
b)  $E.E$  d)  $\log(E)$  **Answer: (a)**

2. What algorithm technique is used in the implementation of Prim's solution for the MST?  
a) Greedy Technique  
b) **Divide-and-Conquer Technique**  
c) Dynamic Programming Technique  
d) The algorithm combines more than one of the above techniques **Answer: (b)**

3. The output of Prim's algorithm is KRUSKAL ALGORITHM.
4. Define Prim's algorithm.

In computer science, **Prim's** (also known as Jarník's) **algorithm** is a greedy **algorithm** that finds a minimum spanning tree **for** a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

5. For minimum Spanning Tree construction, Prim's algorithm selects an edge  
a) That adds a new vertex to partially constructed tree with minimal increment in cost of MST  
b) With maximum number of vertices connected to it so that MST has least diameter  
c) **With minimum weight so that cost of MST is always minimum.**  
d) That does not introduce a cycle.

**Answer: (C)**

## Conclusion/ Outcome:

Here we learn about how Implement prim's algorithm

## Practical Set-12

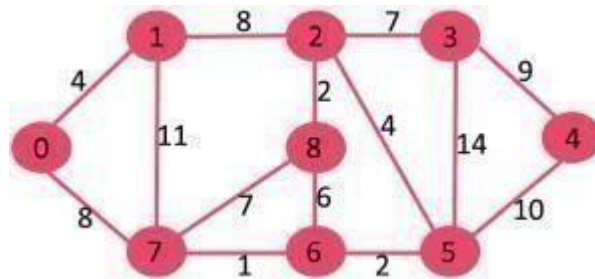
**Aim: Implement Krushal's algorithm.**

### Explanation:

Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are  $(V-1)$  edges in the spanning tree.

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having  $(9 - 1) = 8$  edges.

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

Now pick all edges one by one from sorted list of edges

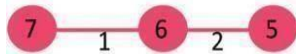
1. Pick edge 7-6: No cycle is formed, include it.



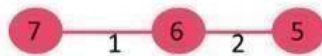
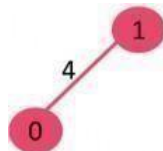
2. Pick edge 8-2: No cycle is formed, include it.



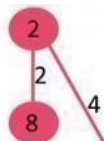
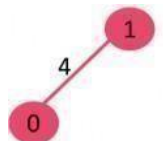
3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.

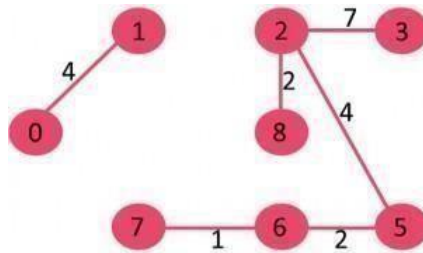


5. Pick edge 2-5: No cycle is formed, include it.



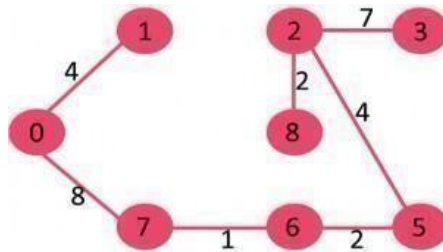
6. Pick edge 8-6: Since including this edge results in cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.



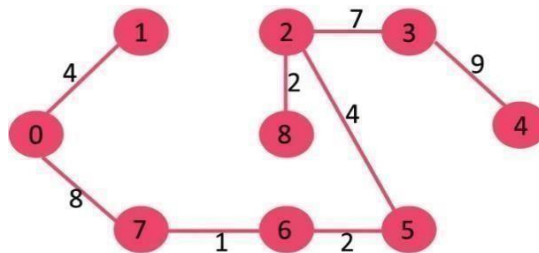
8. Pick edge 7-8: Since including this edge results in cycle, discard it.

9. Pick edge 0-7: No cycle is formed, include it.



10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.



Since the number of edges included equals  $(V - 1)$ , the algorithm stops here.

**Code:**

```
#include <stdio.h>

#define MAX 30

typedef struct edge {
    int u, v, w;
} edge;

typedef struct edge_list {
    edge data[MAX];
    int n;
} edge_list;

edge_list elist;

int Graph[MAX][MAX], n;
edge_list spanlist;

void kruskalAlgo();
int find(int belongs[], int vertexno);
void applyUnion(int belongs[], int c1, int c2);
void sort();
void print();

// Applying Krushkal Algo
void kruskalAlgo() {
    int belongs[MAX], i, j, cno1, cno2;
    elist.n = 0;

    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++) {
            if (Graph[i][j] != 0) {
                elist.data[elist.n].u = i;
                elist.data[elist.n].v = j;
                elist.data[elist.n].w = Graph[i][j];
                elist.n++;
            }
        }

    sort();

    for (i = 0; i < n; i++)
        belongs[i] = i;
```

```

spanlist.n = 0;

for (i = 0; i < elist.n; i++) {
    cno1 = find(belongs, elist.data[i].u);
    cno2 = find(belongs, elist.data[i].v);

    if (cno1 != cno2) {
        spanlist.data[spanlist.n] = elist.data[i];
        spanlist.n = spanlist.n + 1;
        applyUnion(belongs, cno1, cno2);
    }
}

int find(int belongs[], int vertexno) {
    return (belongs[vertexno]);
}

void applyUnion(int belongs[], int c1, int c2) {
    int i;

    for (i = 0; i < n; i++)
        if (belongs[i] == c2)
            belongs[i] = c1;
}

// Sorting algo
void sort() {
    int i, j;
    edge temp;

    for (i = 1; i < elist.n; i++)
        for (j = 0; j < elist.n - 1; j++)
            if (elist.data[j].w > elist.data[j + 1].w) {
                temp = elist.data[j];
                elist.data[j] = elist.data[j + 1];
                elist.data[j + 1] = temp;
            }
}

// Printing the result
void print() {
    int i, cost = 0;

    for (i = 0; i < spanlist.n; i++) {
        printf("\n%d - %d : %d", spanlist.data[i].u, spanlist.data[i].v, spanlist.data[i].w);
        cost = cost + spanlist.data[i].w;
    }
}

```

```

    printf("\nSpanning tree cost: %d", cost);
}

int main() {
    int i, j, total_cost;

    n = 6;

    Graph[0][0] = 0;
    Graph[0][1] = 4;
    Graph[0][2] = 4;
    Graph[0][3] = 0;
    Graph[0][4] = 0;
    Graph[0][5] = 0;
    Graph[0][6] = 0;

    Graph[1][0] = 4;
    Graph[1][1] = 0;
    Graph[1][2] = 2;
    Graph[1][3] = 0;
    Graph[1][4] = 0;
    Graph[1][5] = 0;
    Graph[1][6] = 0;

    Graph[2][0] = 4;
    Graph[2][1] = 2;
    Graph[2][2] = 0;
    Graph[2][3] = 3;
    Graph[2][4] = 4;
    Graph[2][5] = 0;
    Graph[2][6] = 0;

    Graph[3][0] = 0;
    Graph[3][1] = 0;
    Graph[3][2] = 3;
    Graph[3][3] = 0;
    Graph[3][4] = 3;
    Graph[3][5] = 0;
    Graph[3][6] = 0;

    Graph[4][0] = 0;
    Graph[4][1] = 0;
    Graph[4][2] = 4;
    Graph[4][3] = 3;
    Graph[4][4] = 0;
    Graph[4][5] = 0;
    Graph[4][6] = 0;

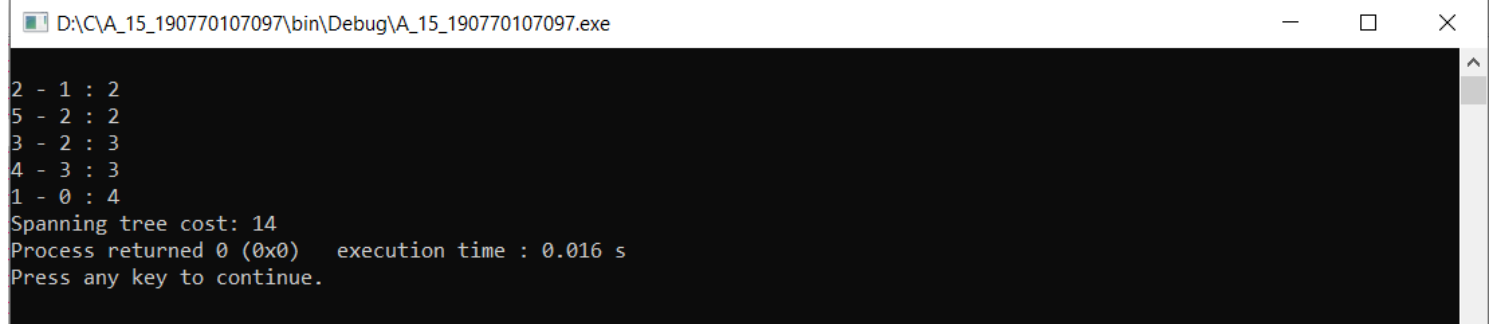
    Graph[5][0] = 0;
    Graph[5][1] = 0;

```



```
Graph[5][2] = 2;  
Graph[5][3] = 0;  
Graph[5][4] = 3;  
Graph[5][5] = 0;  
Graph[5][6] = 0;  
  
kruskalAlgo();  
print();  
}
```

## Output:



The screenshot shows a Windows command prompt window with the title bar "D:\C\A\_15\_190770107097\bin\Debug\A\_15\_190770107097.exe". The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt displays the following output:

```
2 - 1 : 2  
5 - 2 : 2  
3 - 2 : 3  
4 - 3 : 3  
1 - 0 : 4  
Spanning tree cost: 14  
Process returned 0 (0x0)   execution time : 0.016 s  
Press any key to continue.
```

### Give the Answer to the below Questions:

1. What is the time complexity to extract a vertex from the priority queue in Krushkal's algorithm?

a)  $\log(V)$   
c)  $E.E$

b)  $V.V$   
d)  $E \log(E)$

Answer: ( D )

2. The output of Krushkal's algorithm is SPANNING TREE .

3. Define Krushkal's Algorithm.

**Kruskal's algorithm** finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest

4. Assume a graph is having 10 vertices and 20 edges. In Krushkal's minimum spanning tree method, 5 edges are rejected. How many edges are not considered during execution of algorithm on the given graph?

a) 6

c) 5

b) 4

d) 10

Answer: ( b )

5. PRIMS Algorithm technique is used in the implementation of Krushkal's solution for the MST?

### Conclusion/ Outcome:

Here we conclude how krushkal algorithm works and implement.

## References

1. <http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html>
2. [http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge\\_sort.html](http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Merge_sort.html)
3. <http://cs.uef.fi/pages/franti/asa/notes.html>
4. <https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/INDEX.HTM>
5. IETE e Material
6. <https://www.geeksforgeeks.org>
7. <https://www.programiz.com/>
8. <https://www.thecrazyprogrammer.com>
9. <http://c-program-example.com/2011/10>
10. [https://www.tutorialspoint.com/data\\_structures\\_algorithms](https://www.tutorialspoint.com/data_structures_algorithms)
11. <http://www.c4learn.com/c-programs>
12. <http://c-program-example.com/>



