

Department of Computer Science & Engineering

Practical File

**Subject: Data Analysis And Algorithms
Lab (BTCS405-18)**

B.Tech – 4th Semester

[Batch 2019 – 23]



**Chandigarh Group of Colleges
College of Engineering, Landran, Mohali-140307**

**Submitted To :
Mr. Kapil Mehta**

Submitted By :

INDEX

Sr. No.	Title	Pg. No.
i	Objective of the Lab	3
ii	List of Experiments	4
1.	Code and analyze to compute the greatest common divisor (GCD) of two numbers	5-6
2.	Code and analyze to find the median element in an array of integers.	7-8
3.	Code and analyze to find the majority element in an array of integers.	9-11
4.	Code and analyze to sort an array of integers using Heap sort.	12-14
5.	Code and analyze to sort an array of integers using Merge sort.	15-17
6.	Code and analyze to sort an array of integers using Quick sort.	18-20
7.	Code and analyze to find the edit distance between two character strings using dynamic programming.	21-23
8.	Code and analyze to find an optimal solution to matrix chain multiplication using dynamic programming.	24-27
9.	Code and analyze to do a depth first search (DFS) on an undirected graph. Implementing an application of DFS such as (i) to find the topological sort of a directed acyclic graph.	28-32
10.	Code and analyze to do a breadth first search (BFS) on an undirected graph. Implementing an application of BFS such as (i) to find connected components of an undirected graph.	33-37
11.	Code and analyze to find shortest paths in a graph with positive edge weights using Dijkstra's algorithm.	38-39
12.	Code and analyze to find shortest paths in a graph with arbitrary edge weights using Bellman Ford algorithm.	40-42
13.	Code and analyze to find the minimum spanning tree in a weighted, undirected graph.	43-47
14.	Code and analyze to find all occurrences of a pattern P in a given string S.	48-50
15.	Code and analyze to multiply two large integers using Karatsuba algorithm.	51-53
16.	Code and analyze to compute the convex hull of a set of points in the plane.	54-57
17.	Code and analyze to multiply two polynomials using Fast Fourier Transform.	58-60
Experiments Beyond Syllabus		
18.	Write a program to convert the infix expression into postfix expression.	62-67
19.	Write a program to evaluate postfix expression using stack.	68-69

LIST OF EXPERIMENTS

1. Code and analyze to compute the greatest common divisor (GCD) of two numbers
2. Code and analyze to find the median element in an array of integers.
3. Code and analyze to find the majority element in an array of integers.
4. Code and analyze to sort an array of integers using Heap sort.
5. Code and analyze to sort an array of integers using Merge sort.
6. Code and analyze to sort an array of integers using Quick sort.

7. Code and analyze to find the edit distance between two character strings using dynamic programming.
8. Code and analyze to find an optimal solution to matrix chain multiplication using dynamic programming.
9. Code and analyze to do a depth first search (DFS) on an undirected graph. Implementing an application of DFS such as (i) to find the topological sort of a directed acyclic graph.
10. Code and analyze to do a breadth first search (BFS) on an undirected graph. Implementing an application of BFS such as (i) to find connected components of an undirected graph.
11. Code and analyze to find shortest paths in a graph with positive edge weights using Dijkstra's algorithm.
12. Code and analyze to find shortest paths in a graph with arbitrary edge weights using Bellman Ford algorithm.
13. Code and analyze to find the minimum spanning tree in a weighted, undirected graph.
14. Code and analyze to find all occurrences of a pattern P in a given string S.
15. Code and analyze to multiply two large integers using Karatsuba algorithm.
16. Code and analyze to compute the convex hull of a set of points in the plane.
17. Code and analyze to multiply two polynomials using Fast Fourier Transform.

Experiments Beyond Syllabus

18. Write a program to convert the infix expression into postfix expression.

19. Write a program to evaluate postfix expression using stack.

EXPERIMENT NO. 1

Aim: Write a program to compute the greatest common divisor (GCD) of two numbers

Objective: To compute the greatest common divisor (GCD) of two numbers.

Procedure:

In mathematics, the Euclidean algorithm, or Euclid's algorithm, is a method for computing the greatest common divisor (GCD) of two (usually positive) integers, also known as the greatest common factor (GCF) or highest common factor (HCF). It is named after the Greek mathematician Euclid. The GCD of two positive integers is the largest integer that divides both of them without leaving a remainder (the GCD of two integers in general is defined in a more subtle way).

In its simplest form, Euclid's algorithm starts with a pair of positive integers, and forms a new pair that consists of the smaller number and the difference between the larger and smaller numbers. The process repeats until the numbers in the pair are equal. That number then is the greatest common divisor of the original pair of integers.

The main principle is that the GCD does not change if the smaller number is subtracted from the larger number. For example, the GCD of 252 and 105 is exactly the GCD of 147 (= 252 - 105) and 105. Since the larger of the two numbers is reduced, repeating this process gives successively smaller numbers, so this repetition will necessarily stop sooner or later — when the numbers are equal (if the process is attempted once more, one of the numbers will become 0).

Algorithm:

This can be done in following two ways:

Euclid's algorithm

- Get m, n.
- If n=0, return the value of m as the answer and stop; otherwise, proceed to step 3.
- Divide m by n and assign the value of the remainder to r.
- Assign the value of n to m and the value of r to n. Go to step 2.

Consecutive Integer Checking Algorithm

- Assign the value of min (m, n) to t.
- Divide m by t. If the remainder of this division is 0, go to step 3; otherwise, go to step 4.

- Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, go to step 4.
- Decrease the value of t by 1. Go to step 2.

Flowchart:

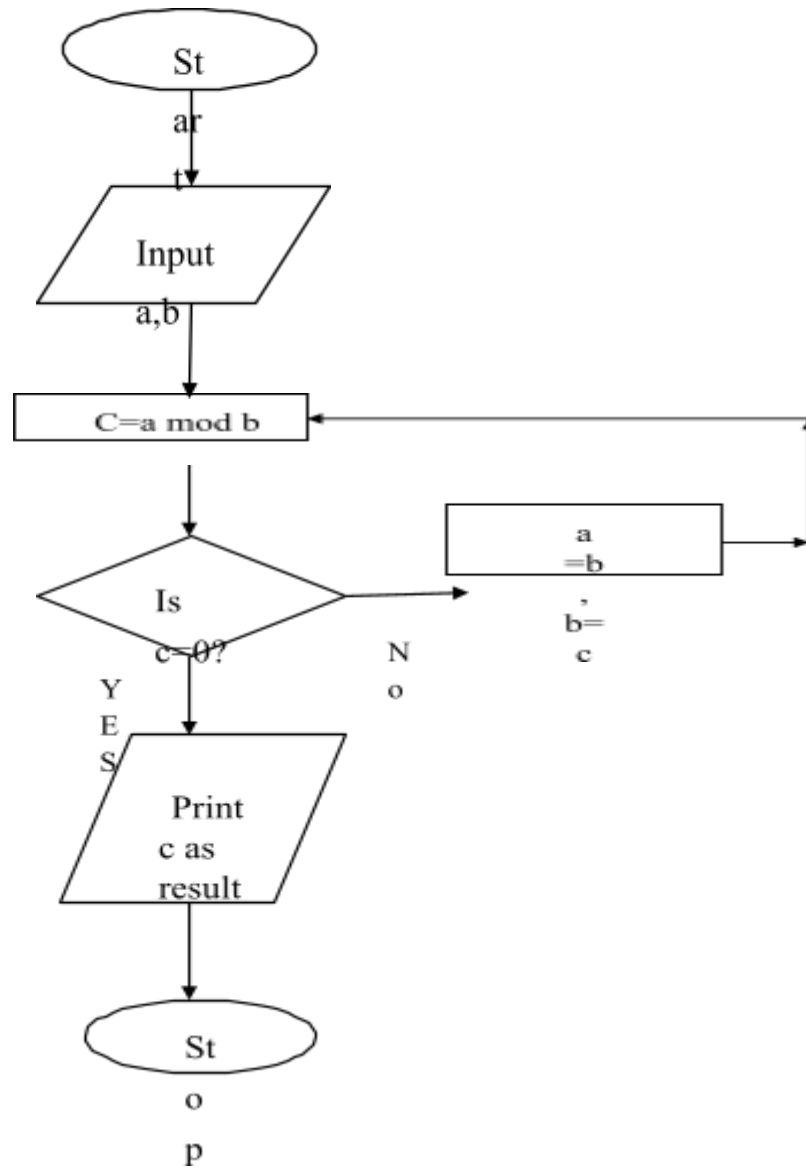


Fig 1.1

Outcome: - After performing this experiment, students will be able to tell what the GCD of two numbers actually is, and how it can be calculated using recursion as a computer science problem.

EXPERIMENT NO. 2

Aim: Write a program to find the median element in an array of integers.

Objective: To compute the median element in an array of integers.

Procedure:

This Problem Can be done is a **linear Time** $O(N)$, where $N=A.length()$. Yes, **Selection Algorithm** Finds the *Median of an unsorted Array without Sorting it*. The Selection Algorithm uses the **concept of Quick Sort** [But does not actually sort the array though], especially the partition Steps.

This algorithm works in two steps. The partitioning step works by picking some pivot element, then rearranging the elements of the array such that everything less than the pivot is to one side, everything greater than the pivot is to the other side, and the pivot is in the correct place.

This is the main concept used by the selection Algorithm.

Algorithm:

Step1: Start

Step2: Take N input for DATA array

Step3: Print: 'Sort the DATA array in ascending order'

Step4: Repeat for $i := 1$ to N by 1

Repeat for $j := i + 1$ to N by 1

1 If $DATA[i] > DATA[j]$

then: $temp = DATA[i]$

$DATA[i] = DATA[j]$

$DATA[j] = temp$

[End of If structure]

[End of inner loop]

[End of outer loop]

Step5: If $N\%2 \neq 0$

Median := $(N +$

$1)/2$

Print: 'Median is Median.'

Else

$P := N/2$

$Q := N/2 + 1$

Median := $(DATA[P] + DATA[Q])/2$

Print: 'Median is

Median.' [End of If

structure] Step6: Stop

Flowchart:

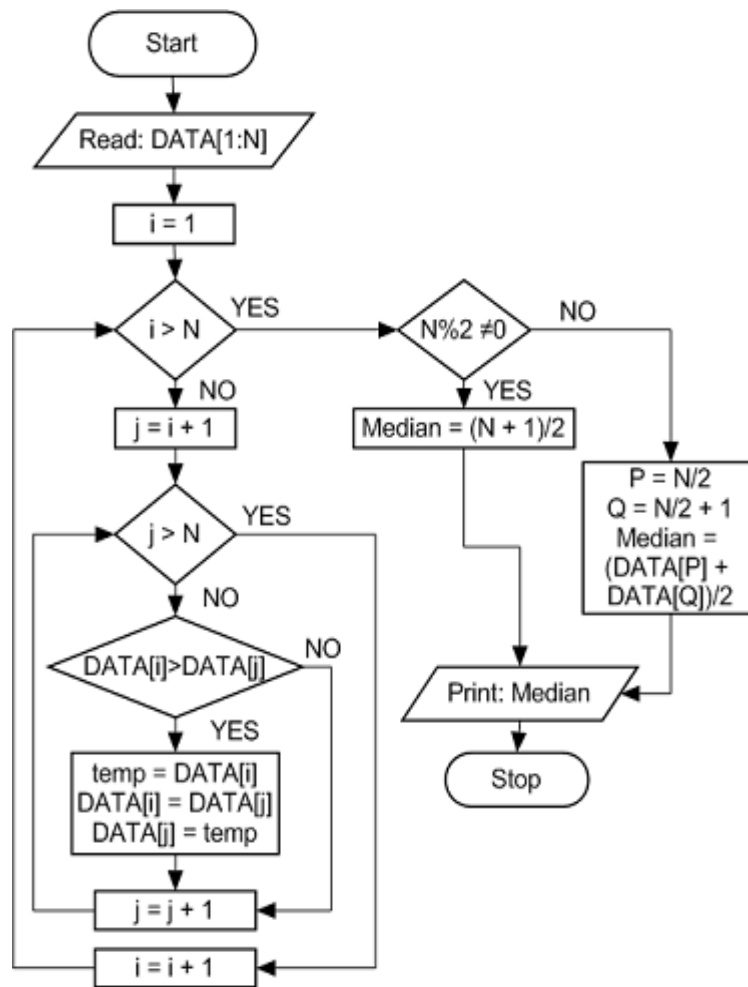


Fig 2.1

Outcome: - After performing this experiment, students will be able to tell what the median of a set of element is, and how it can be calculated.

EXPERIMENT NO. 3

Aim: Write a program to find the majority element in an array of integers.

Objective: To compute the majority element in an array of integers.

Procedure:

Given a large array of non-negative integer numbers, write a function which determines whether or not there is a number that appears in the array more times than all other numbers combined. If such element exists, function should return its value; otherwise, it should return a negative value to indicate that there is no majority element in the array.

Example: Suppose that array consists of values 2, 6, 1, 2, 2, 4, 7, 2, 2, 2, 1, 2. Majority element in this array is number 2, which appears seven times while all other values combined occupy five places in the array.

Keywords: Array, searching, majority, vote.

Algorithm:

The majority element is the element that occurs more than half of the size of the array.

Algorithm below loops through each element and maintains a count of a[maj_index], If next element is same then increments the count, if next element is not same then decrements the count, and if the count reaches 0 then changes the maj_index to the current element and sets count to 1. First Phase algorithm gives us a candidate element. In second phase we need to check if the candidate is really a majority element.

Second phase is simple and can be easily done in $O(n)$. We just need to check if count of the candidate element is greater than $n/2$.

Moore's Majority Algorithm:

c = particular candidate

Step 0: Initialize count to be zero

Step 1: Go through the vote

i. Check if the count is zero

if so then assign the c to this vote's candidate and then set count=1

if not then check if this vote's candidate is equal to c

if yes then set

count+=1 if no then set

count-=1

Step 2: Continue until the end

Step 3: For that c, Check once again by going through the votes if it is really the majority.

Function provided below returns -1 if there is no majority element otherwise that element.

```
int findMajorityElement(int * arr, int size)
{
    int count = 0, i, majorityElement;
    for (i = 0 ; i < size ; i++)
    {
        if (count == 0) {
            majorityElement = arr[i];
            count = 1;
        }
        else
        {
            if(arr[i] == majorityElement)
                count++;
            else
                count--;
        }
    }
    count = 0;
    for (i = 0; i < size; i++) {
        if (arr[i] == majorityElement) {
            count++;
        }
    }
    if (count > size/2) {
        return majorityElement;
    }
    else return -1;
}
```

Time Complexity : $O(n)$

Space Complexity : $O(1)$

Flowchart:

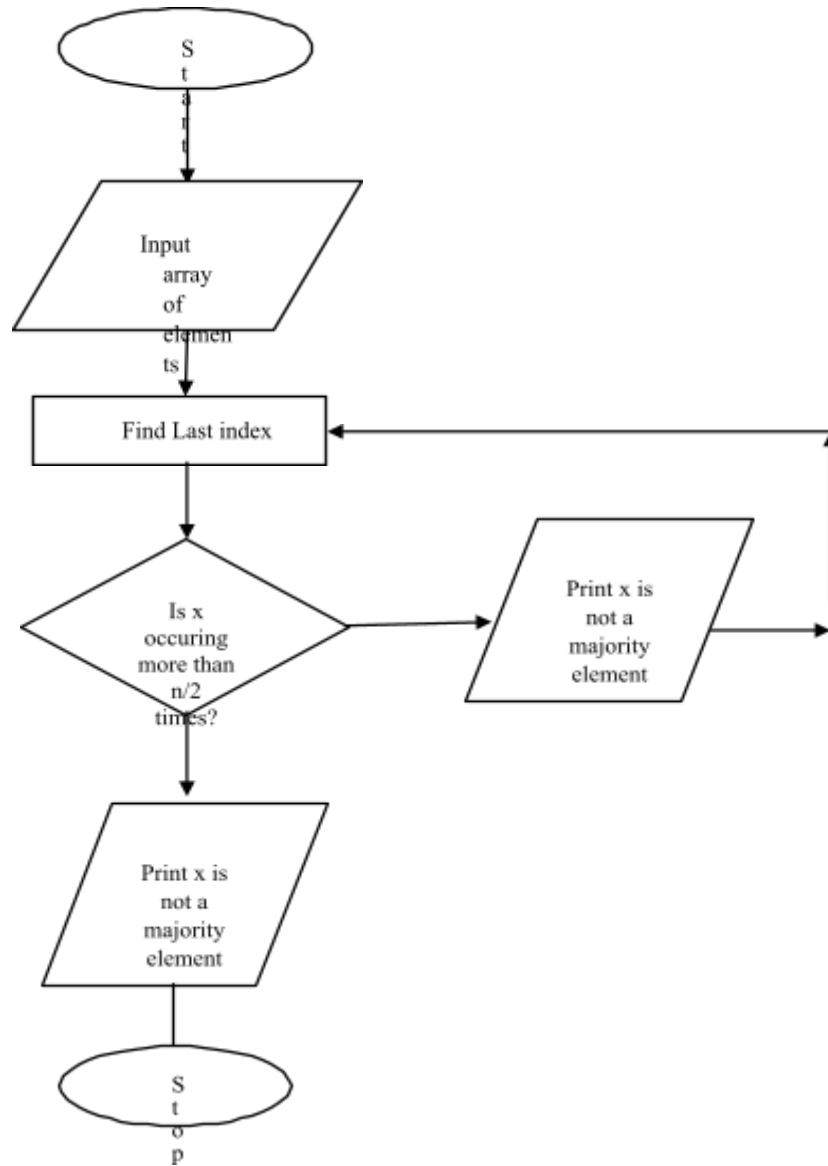


Fig 3.1

Outcome:- After performing this experiment, Students will be able to check whether an element is the majority element or not in an array.

EXPERIMENT NO. 4

Aim: Write a program to Code and analyze to sort an array of integers using Heap sort.

Objective: To sort an array of integers using Heap sort.

Procedure

Heap Sort Algorithm

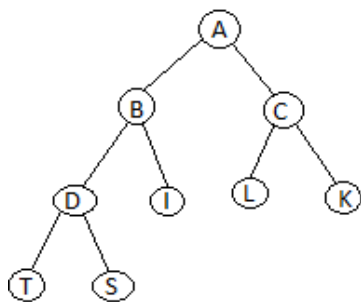
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

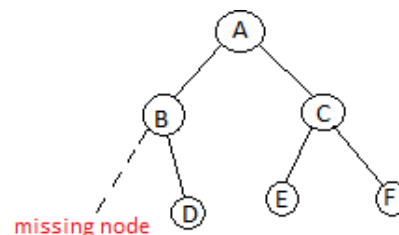
What is a Heap?

Heap is a special tree-based data structure that satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

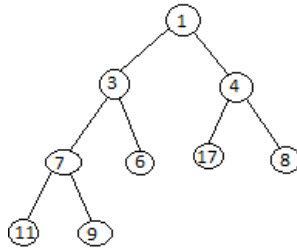


Complete Binary Tree



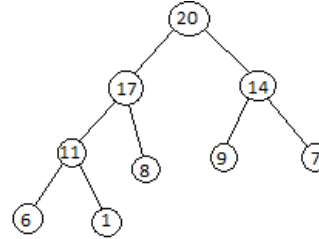
In-Complete Binary Tree

Heap Property: All nodes are either *[greater than or equal to]* or *[less than or equal to]* each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and pick the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

Algorithm:

1. $l \leftarrow \text{left}[i]$
2. $r \leftarrow \text{right}[i]$
3. if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$
4. then $\text{largest} \leftarrow l$
5. else $\text{largest} \leftarrow i$
6. if $r \leq \text{heap-size}[A]$ and $A[r] > A[\text{largest}]$
7. then $\text{largest} \leftarrow r$
8. if $\text{largest} \neq i$
9. then exchange $A[i] \leftrightarrow A[\text{largest}]$
10. Heapify ($A, \text{largest}$)

Flowchart:

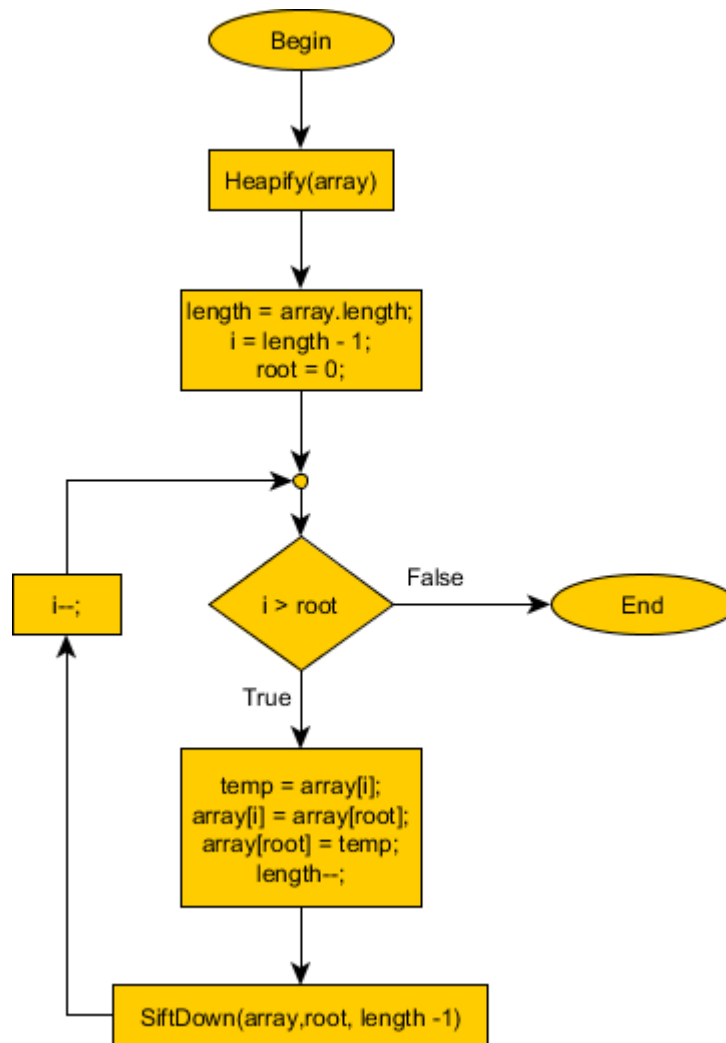


Fig 4.1

Outcome:

Students will be having knowledge of what a heap as a data structure actually means and how it can be used to sort an array of elements through heap sort.

EXPERIMENT NO. 5

Aim: Write a program to sort an array of integers using Merge sort.

Objective: To sort an array of integers using Merge sort.

Procedure:

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being $O(n \log n)$, it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How merge sort works

To understand merge sort, we take an unsorted array as depicted below –



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



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



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

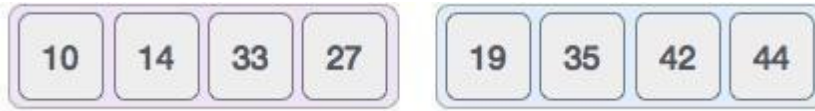


Now, we combine them in exactly same manner they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order 19 and 35. 42 and 44 are placed sequentially.



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



After final merging, the list should look like this –



Algorithm:

MERGE (A, p, q, r)

1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. Create arrays $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$
4. **FOR** $i \leftarrow 1$ **TO** n_1
5. **DO** $L[i] \leftarrow A[p + i - 1]$
6. **FOR** $j \leftarrow 1$ **TO** n_2
7. **DO** $R[j] \leftarrow A[q + j]$
8. $L[n_1 + 1] \leftarrow \infty$
9. $R[n_2 + 1] \leftarrow \infty$
10. $i \leftarrow 1$
11. $j \leftarrow 1$
12. **FOR** $k \leftarrow p$ **TO** r
13. **DO IF** $L[i] \leq R[j]$
14. **THEN** $A[k] \leftarrow L[i]$
15. $i \leftarrow i + 1$
16. **ELSE** $A[k] \leftarrow R[j]$
17. $j \leftarrow j + 1$

Flowchart:

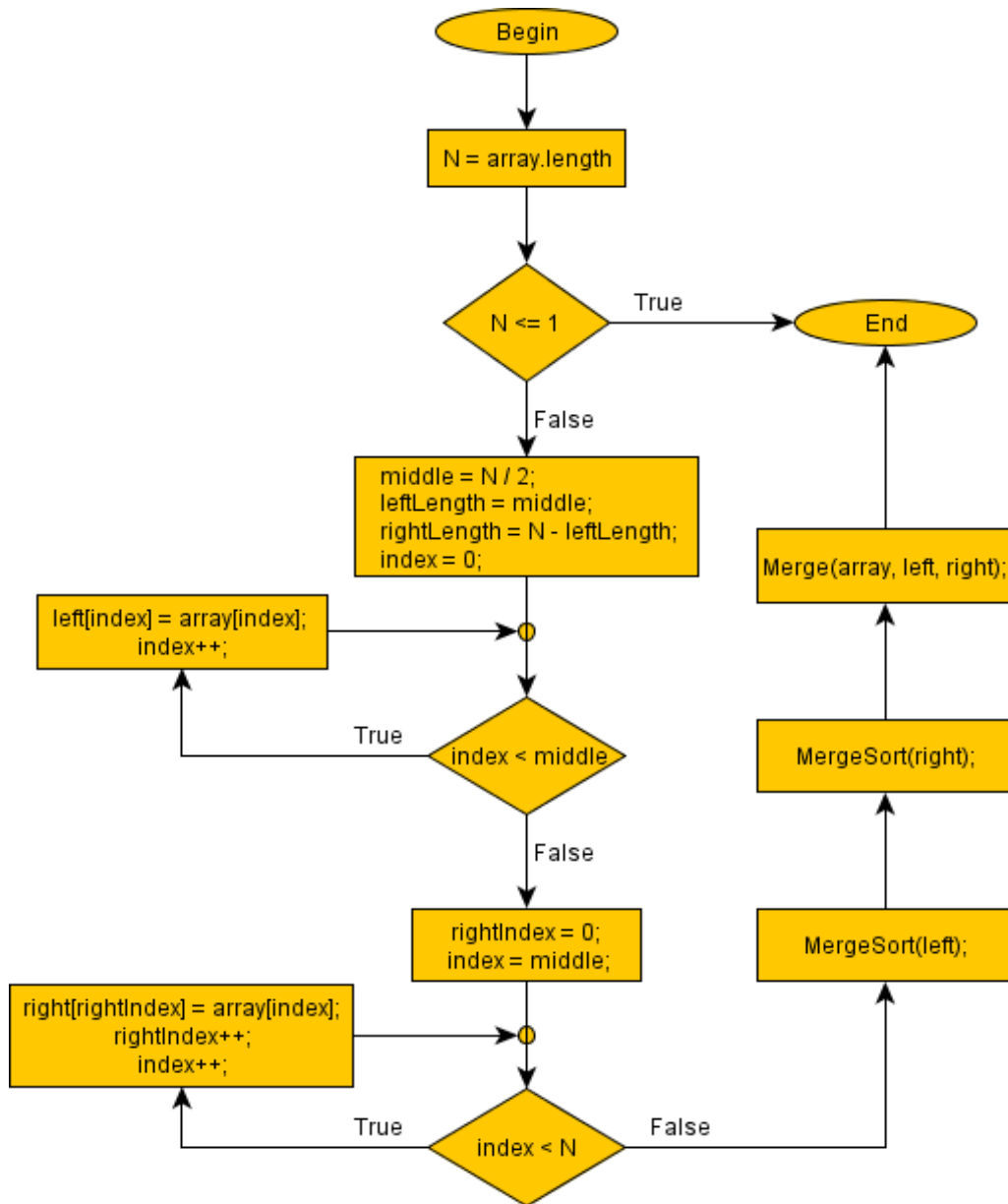


Fig 5.1

Outcome: - After performing this experiment, students will be able to sort an array of elements using Merge sort and be able to have an exact idea of Divide and Conquer strategy.

EXPERIMENT NO. 6

Aim: Write a program to sort an array of integers using Quick sort.

Objective: To sort an array of integers using Quick sort.

Procedure:

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than specified value say pivot based on which the partition is made and another array holds values greater than pivot value.

The quick sort partitions an array and then calls itself recursively twice to sort the resulting two subarray. This algorithm is quite efficient for large sized data sets as its average and worst case complexity are of $O(n \log n)$ where n are no. of items.

Partition in Quicksort

The pivot value divides the list in to two parts. And recursively we find pivot for each sub-lists until all lists contains only one element. We initialize i and j pointers at the start and end of the array indices initially. We increment i pointer until we find the greater element than pivot and decrement j pointer until we find the smaller element than pivot simultaneously. In due course if $i < j$, we swap $a[i]$ and $a[j]$ else if $i > j$, we swap $a[j]$ with pivot element and thus the pivot element is set at its own position after one recursive definition of Quicksort. Recursively applying the above strategy (known as Quick Sort) on smaller arrays i.e. left part and right part, we will eventually get the final sorted array.

Algorithm:

First call: QuickSort (Data, 0, Count-1)

Left

Check if at least two values have to be sorted

PivotIndex := Partition (Data, Left, Right)

The right element will be used as pivot element. The function reorders the list so that all elements which are less than the pivot element come before the pivot element and so that all elements greater than the pivot element come after it (equal values can go either way). After this partitioning, the pivot is in its final position. The return value is the new position of the pivot element.

QuickSort (Data, Left, PivotIndex - 1)

Sort all Data left of the pivot element.

QuickSort (Data, PivotIndex + 1, Right)

Sort all Data right of the pivot element

i := Left

Counter from the left hand side

j := Right - 1

Counter from the right hand side

Pivot := Data [Right]

Value of the pivot element. After sorting has finished, all values less than the pivot element are on his left hand side and all values greater than the pivot element are on his right hand side.

Data [i] <= Pivot and i < Right

Search a wrong sorted value on the left hand side of the pivot element

Data [j] >= Pivot and j > Left

Search a wrong sorted value on the right hand side of the pivot element

i < j

Are there any values remaining which have to be sorted

Swap Data [i] and Data [j]

Swap the wrong sorted values

Swap Data [i] and Data [Right]

Copy the pivot element on his correct position

End Return value: i

Flowchart:

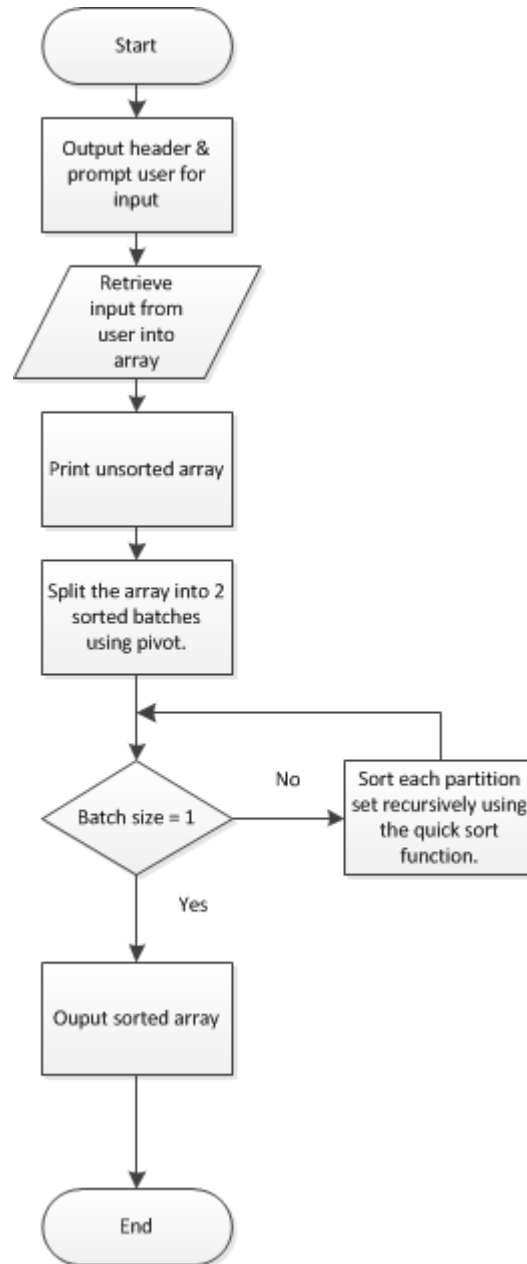


Fig 6.1

Outcome: - After performing this experiment, students will be able to sort an array of elements using Quicksort.

EXPERIMENT NO. 7

Aim: Write a program to find the edit distance between two character strings using dynamic programming.

Objective: To find the edit distance between two character strings using dynamic programming.

Procedure:

Edit Distance is quite an interesting and popular problem. Here I present an efficient bottom up C++ program to solve it.

Problem – We are given 2 strings. We have to find the “edit distance” or the cost of converting one string to other. We are allowed 3 operations – Insert, Delete, Replace. All 3 have equal cost that is 1 unit.

Example – Edit distance between “abc” and “abd” is 1. We replaced “c” with “d”. You can also see this as “Delete c”, then “Insert d” which would give a edit distance of 2. But since we try to minimize edit distance, we consider it as single operation that is Replace. Few more examples-

EditDistance(“ab”, “bc”) = 2

EditDistance(“man”, “woman”) = 2

EditDistance(“c”, “java”) = 4

EditDistance(“abcd”, “acd”) = 2

One of the above 4 examples has wrong answer! Tell me in comments which one is wrong, and what is the correct answer.

We will use bottom up version of dynamic programming, as it is much cleaner and efficient (than top down). If the strings have length N and M, we will need a table of size (N+1)*(M+1). First we need to initialize the top row i.e TABLE[0][i] = i and first column i.e TABLE[i][0] = i. These are the base cases in which one of the string is empty. Now filling the rest of the TABLE column wise is simple-

$TABLE[i][j] = \min(TABLE[i][j-1]+1, TABLE[i-1][j]+1, TABLE[i-1][j-1]+cost,)$

Here cost = 0 if STR1[i] = STR2[j], 1 if unequal.

The TABLE for “money” and “monkey” looks like->

		m	o	n	k	e	y
	0	1	2	3	4	5	6
m	1	0	1	2	3	4	5
o	2	1	0	1	2	3	4
n	3	2	1	0	1	2	3
e	4	3	2	1	1	1	2
y	5	4	3	2	2	2	1

Fig 7.1

1 is the final answer.

Algorithm:

Function for Edit_Distance between two strings is given below:-

```
int EditDistance(string word1, string word2)
{
    int i, j, l1, l2, m;
    l1 =
    word1.length(); l2
    = word2.length();
    vector< vector<int> > t(l1 + 1, vector<int>(l2 + 1));

    for (i = 0; i <= l1;
        i++) t[i][0] = i;
    for (i = 1; i <= l2;
        i++) t[0][i] = i;

    for (i = 1; i <= l1; i++)
    {
        for (j = 1; j <= l2; j++)
        {
            m = min(t[i-1][j], t[i][j-1]) + 1;
            t[i][j] = min(m, t[i-1][j-1] + (word1[i-1] == word2[j-1] ? 0 : 1));
        }
    }
    return t[l1][l2];
}
```

Flowchart:

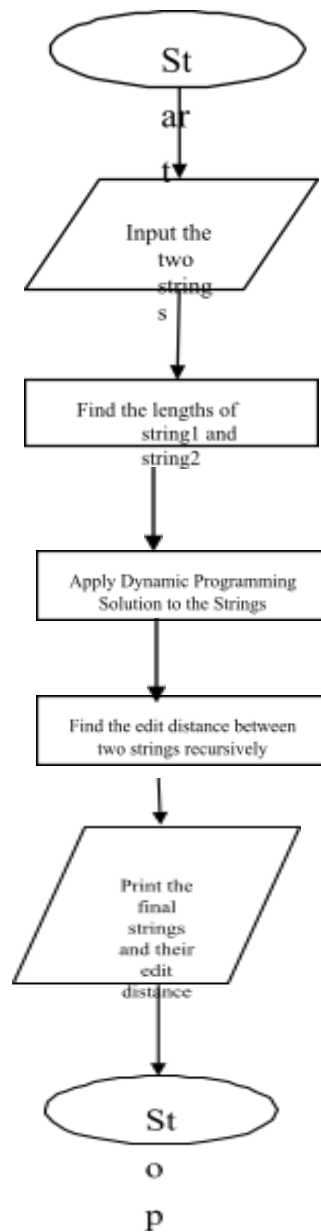


Fig 7.2

Outcome: - After performing this experiment, students will be able to use the concept of Dynamic Programming in solving the famous Edit Distance problem.

EXPERIMENT NO. 8

Aim: Write a program to find an optimal solution to matrix chain multiplication using dynamic programming.

Objective: To an optimal solution to matrix chain multiplication using dynamic programming.

Procedure:

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 10×30 matrix, B is a 30×5 matrix, and C is a 5×60 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.

Input: $p[] = \{40, 20, 30, 10, 30\}$

Output: 26000

There are 4 matrices of dimensions 40×20 , 20×30 , 30×10 and 10×30 .

Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

$(A(BC))D \rightarrow 20 \times 30 \times 10 + 40 \times 20 \times 10 + 40 \times 10 \times 30$

Input: $p[] = \{10, 20, 30, 40, 30\}$

Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30. Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way $((AB)C)D \rightarrow 10 \times 20 \times 30 + 10 \times 30 \times 40 + 10 \times 40 \times 30$

Input: $p[] = \{10, 20, 30\}$

Output: 6000

There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, cost of which is $10 \times 20 \times 30$

Optimal Substructure:

A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size n , we can place the first set of parenthesis in $n-1$ ways. For example, if the given chain is of 4 matrices. Let the chain be ABCD, then there are 3 way to place first set of parenthesis: A(BCD), (AB)CD and (ABC)D. So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size $n = \text{Minimum of all } n-1 \text{ placements (these placements create subproblems of smaller size)}$

Overlapping Subproblems

Following is a recursive implementation that simply follows the above optimal substructure property.

// Matrix A_i has dimension $p[i-1] \times p[i]$ for $i =$

1..n int MatrixChainOrder(int p[], int i, int j)

{


```

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

// place parenthesis at different places between first
// and last matrix, recursively calculate count of
// multiplications for each parenthesis placement and
// return the minimum
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 minimum
count return min;
}

```

Algorithm:

MATRIX-MULTIPLY (A, B)

Matrix-Chain-Order (p)

1. $n \leftarrow \text{length}[p] - 1$
2. for $l \leftarrow 1$ to n
3. do $m[i, i] \leftarrow 0$
4. For $k \leftarrow 2$ to n l is the chain length.
5. do for $i \leftarrow 1$ to $n-l+1$
6. do $j \leftarrow i+l-1$
7. $m[i, j] \leftarrow \infty$
8. for $k \leftarrow i$ to $j-1$
9. do $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
10. if $q < m[i, j]$
11. then $m[i, j] \leftarrow q$
12. $s[i, j] \leftarrow k$
13. return m and s

Flowchart:

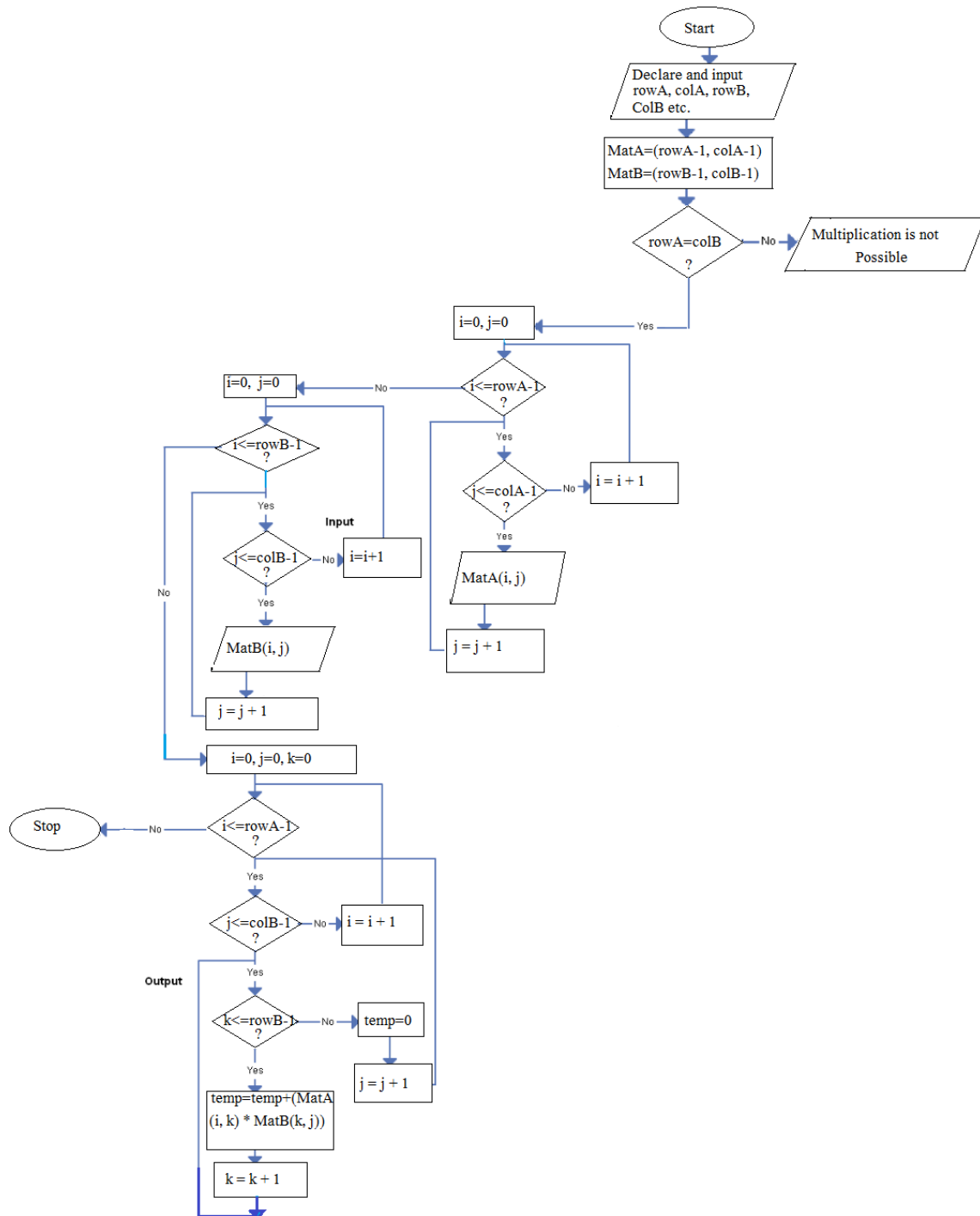


Fig 8.1

Outcome: - After performing this experiment, students will have knowledge of Matrix chain Multiplication – Optimization Problem – Best Parenthesization finding problem; when a chain of matrices is given.

EXPERIMENT NO. 9

Aim: Write a program to do a depth first search (DFS) on an undirected graph. Implement an application of DFS to find the topological sort of a directed acyclic graph.

Objective: To do a depth first search (DFS) on an undirected graph and implement an application of DFS to find the topological sort of a DAG.

Procedure:

Depth First Search algorithm(DFS) traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search when a dead end occurs in any iteration.

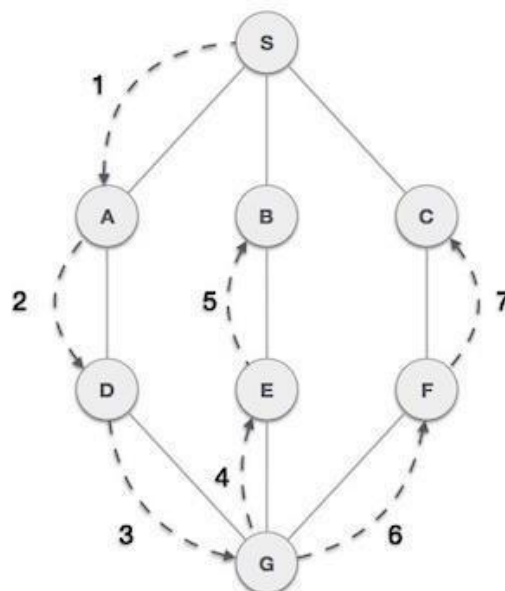
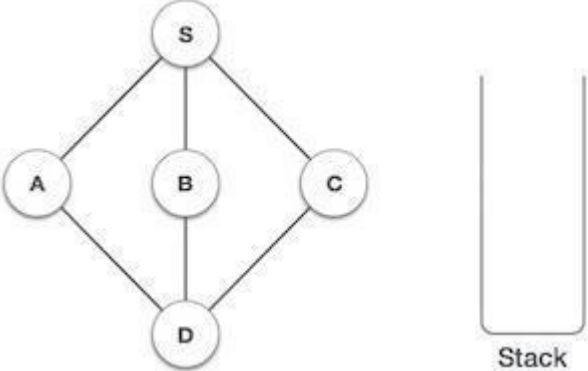
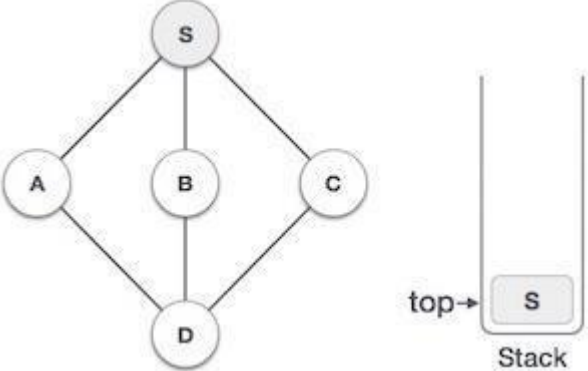
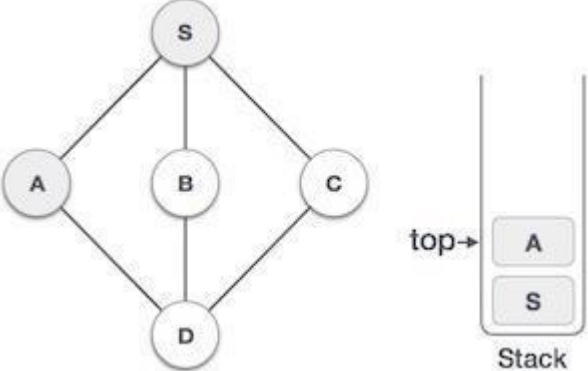
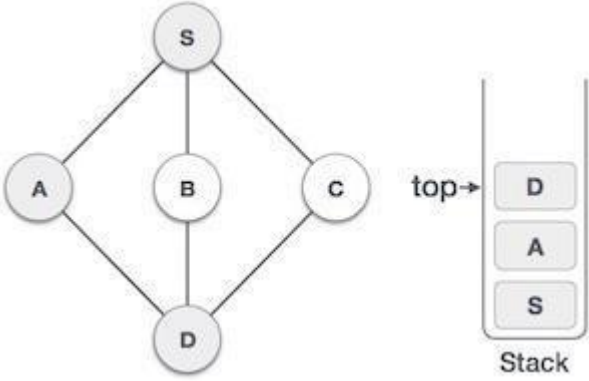
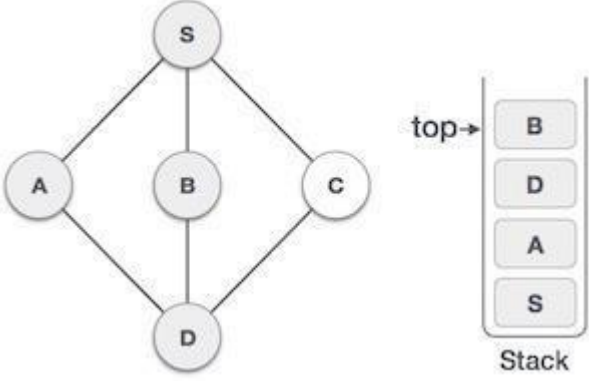
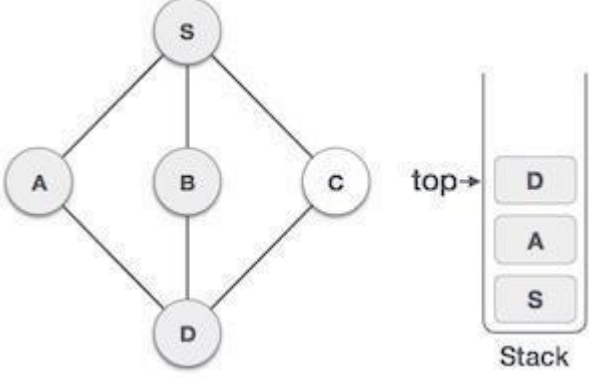


Fig 9.1

As in example given above, DFS algorithm traverses from A to B to C to D first then to E, then to F and lastly to G. It employs following rules.

- **Rule 1** – Visit adjacent unvisited vertex. Mark it visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex found, pop up a vertex from stack. (It will pop up all the vertices from the stack which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until stack is empty.

Step	Traversal	Description
1.	 <p>The graph consists of five nodes: S at the top, A on the left, B in the center, C on the right, and D at the bottom. Edges connect S to A, B, and C; A to D; B to D; and C to D. To the right of the graph is an empty container labeled 'Stack'.</p>	Initialize the stack
2.	 <p>The graph is the same as in step 1. Node S is shaded gray. In the 'Stack' container, the letter 'S' is present, and an arrow labeled 'top' points to it.</p>	Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in alphabetical order.
3.	 <p>The graph is the same as in step 2. Nodes S and A are shaded gray. In the 'Stack' container, 'A' is on top of 'S', and an arrow labeled 'top' points to 'A'.</p>	Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A . Both S and D are adjacent to A but we are concerned for unvisited nodes only.

4.		<p>Visit D and mark it visited and put onto the stack. Here we have B and C nodes which are adjacent to D and both are unvisited. But we shall again choose in alphabetical order.</p>
5.		<p>We choose B, mark it visited and put onto stack. Here B does not have any unvisited adjacent node. So we pop B from the stack.</p>
6.		<p>We check stack top for return to previous node and check if it has any unvisited nodes. Here, we find D to be on the top of stack.</p>

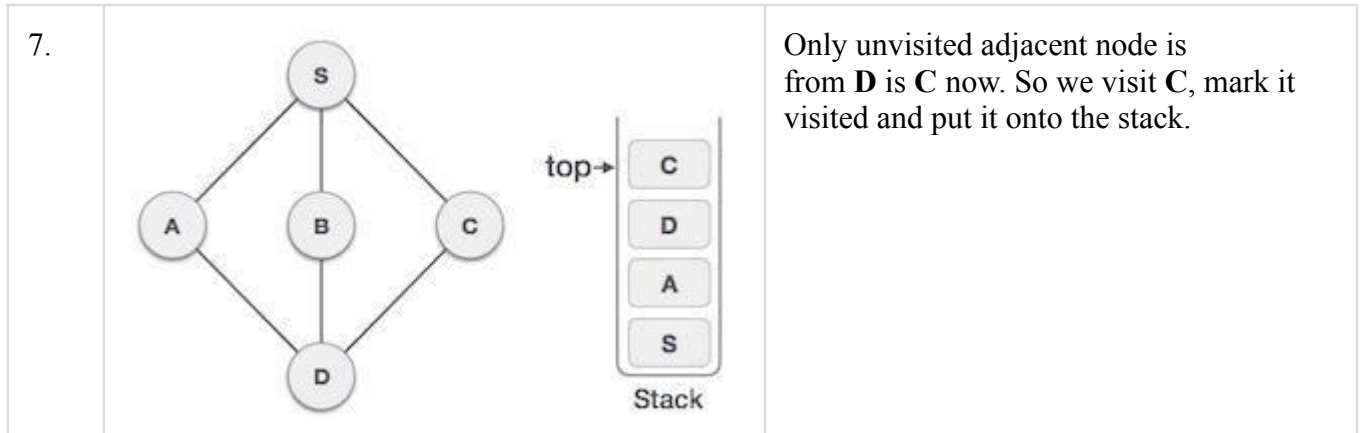


Fig 9.2

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node which has unvisited adjacent node. In this case, there's none and we keep popping until stack is empty.

Algorithm:

Algorithm DFS(G)

1. for each vertex $u \in V(G)$
2. $color[u] \leftarrow white$
3. $\pi[u] \leftarrow nil$
4. $time \leftarrow 0$
5. for each vertex $u \in V(G)$
6. if $color[u] == white$
7. DFS-visit(u)

DFS-visit(u)

1. $color[u] \leftarrow gray$ {white vertex u has just been discovered}
2. $d[u] \leftarrow time \leftarrow time + 1$
3. for each vertex $v \in Adj[u]$ {explore edge($u;v$) }
4. if $color[v]$
5. $\pi[v] \leftarrow u$ {Blacken u ; it is finished}
6. DFS-visit(v)
7. $color[u] \leftarrow black$
8. $f[u] \leftarrow time \leftarrow time + 1$

TOPOLOGICAL-SORT(G)

1. call DFS(G) to compute finishing times $v.f$ for each vertex v .

2. as each vertex is finished, insert it onto the front of a linked list.
3. return the linked list of vertices.

Flowchart:

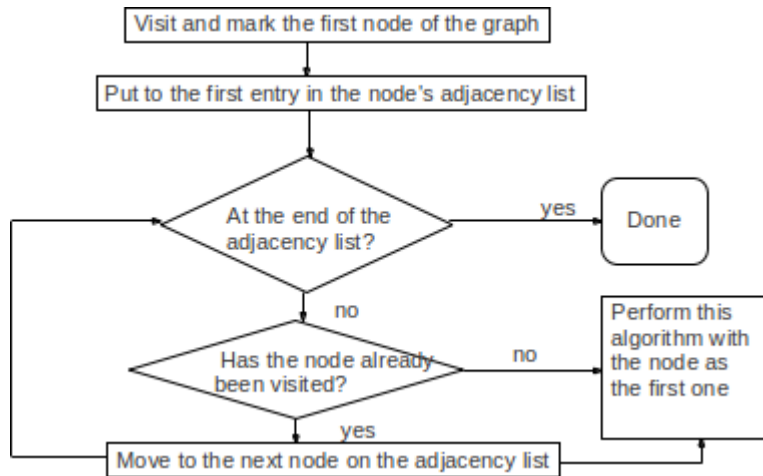


Fig 9.3

Outcome: - After performing this experiment, students will be able to have a good insight to Depth first search (Graph traversal) and its application Topological sort.

EXPERIMENT NO. 10

Aim: Write a program to do a breadth first search (BFS) on an undirected graph. Implement an application of BFS to find connected components of an undirected graph.

Objective: To do a breadth first search (BFS) on an undirected graph and implement an application of BFS to find the connected components of an undirected graph.

Procedure:

Breadth First Search algorithm(BFS) traverses a graph in a breadthwards motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.

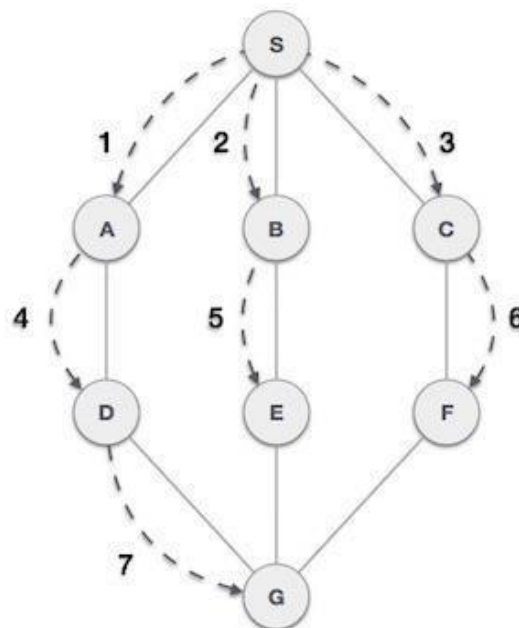
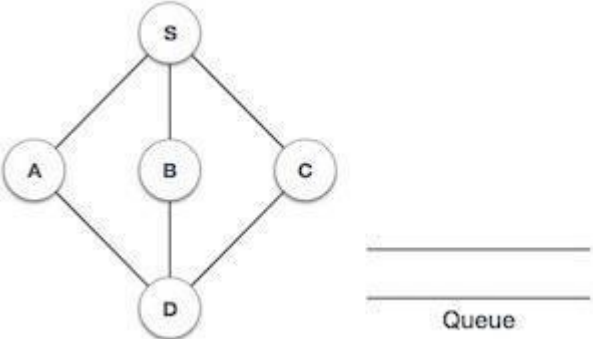
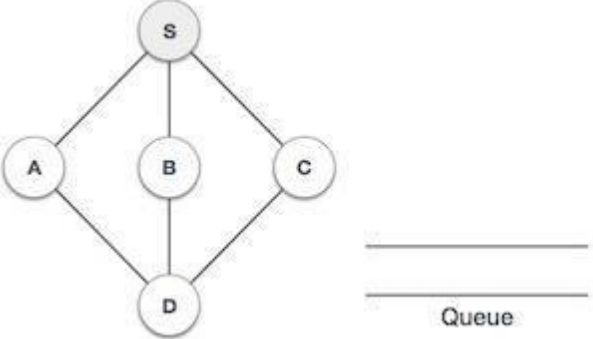
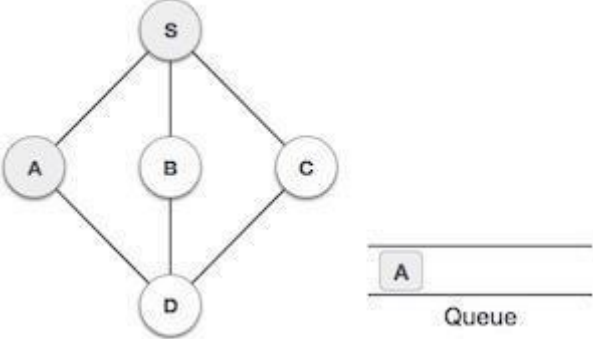


Fig 10.1

As in example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs following rules.

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

Step	Traversal	Description
1.		Initialize the queue.
2.		We start from visiting S(starting node), and mark it visited.
3.		We then see unvisited adjacent node from S. In this example, we have three nodes but alphabetically we choose A mark it visited and enqueue it.

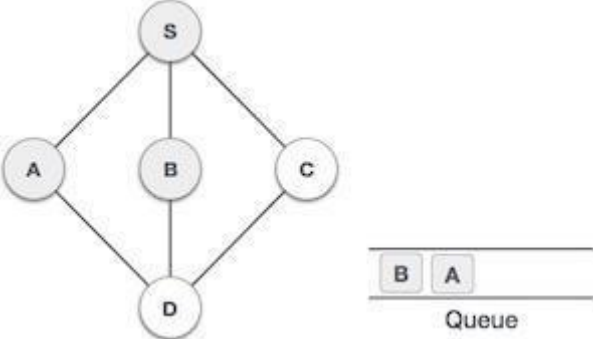
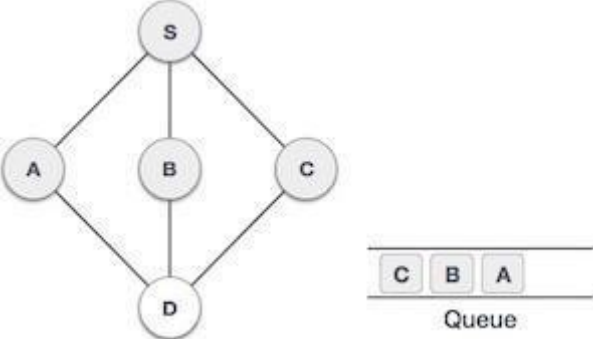
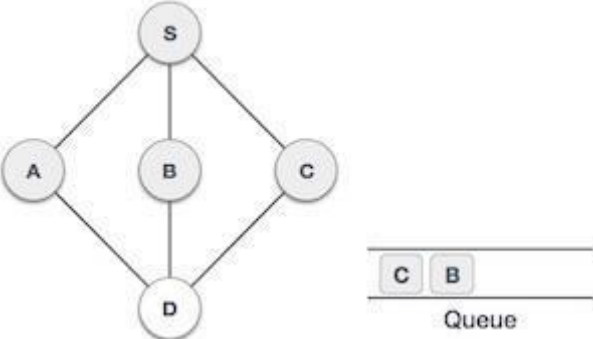
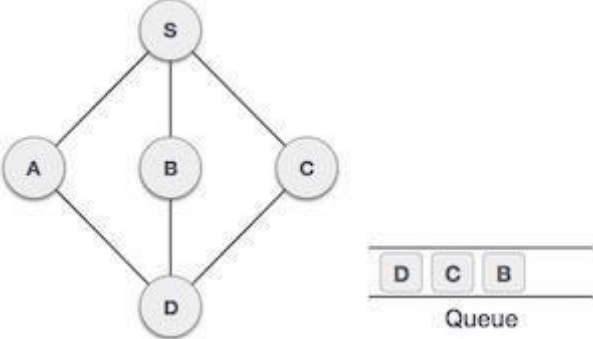
4.		<p>Next unvisited adjacent node from S is B. We mark it visited and enqueue it.</p>
5.		<p>Next unvisited adjacent node from S is C. We mark it visited and enqueue it.</p>
6.		<p>Now S is left with no unvisited adjacent nodes. So we dequeue and find A.</p>
7.		<p>From A we have D as unvisited adjacent node. We mark it visited and enqueue it.</p>

Fig 10.2

At this stage we are left with no unmarked (unvisited) nodes. But as per algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied the program is over.

Algorithm:

```
BFS( $G, s$ )
1 for each vertex  $u \in V[G] - \{s\}$ 
2 do  $color[u] \leftarrow$  WHITE
3  $d[u] \leftarrow \infty$ 
4  $\pi[u] \leftarrow$  NIL
5  $color[s] \leftarrow$ 
GRAY 6  $d[s] \leftarrow 0$ 
7  $\pi[s] \leftarrow$  NIL
8  $Q \leftarrow \emptyset$ 
9 ENQUEUE( $Q, s$ )
10 while  $Q \neq \emptyset$ 
11 do  $u \leftarrow$  DEQUEUE( $Q$ )
12 for each  $v \in Adj[u]$ 
13 do if  $color[v] =$ 
WHITE 14 then  $color[v]$ 
 $\leftarrow$  GRAY 15  $d[v] \leftarrow d[u]$ 
+ 1
16  $\pi[v] \leftarrow u$ 
17 ENQUEUE( $Q, v$ )
18  $color[u] \leftarrow$ 
BLACK
PRINT-PATH( $G, s, v$ )
1 if  $v = s$ 
2 then print  $s$ 
3 else if  $\pi[v] =$  NIL
4 then print "no path from"  $s$  "to"  $v$ 
"exists" 5 else PRINT-PATH( $G, s, \pi[v]$ )
6 print  $v$ 
```

Flowchart:

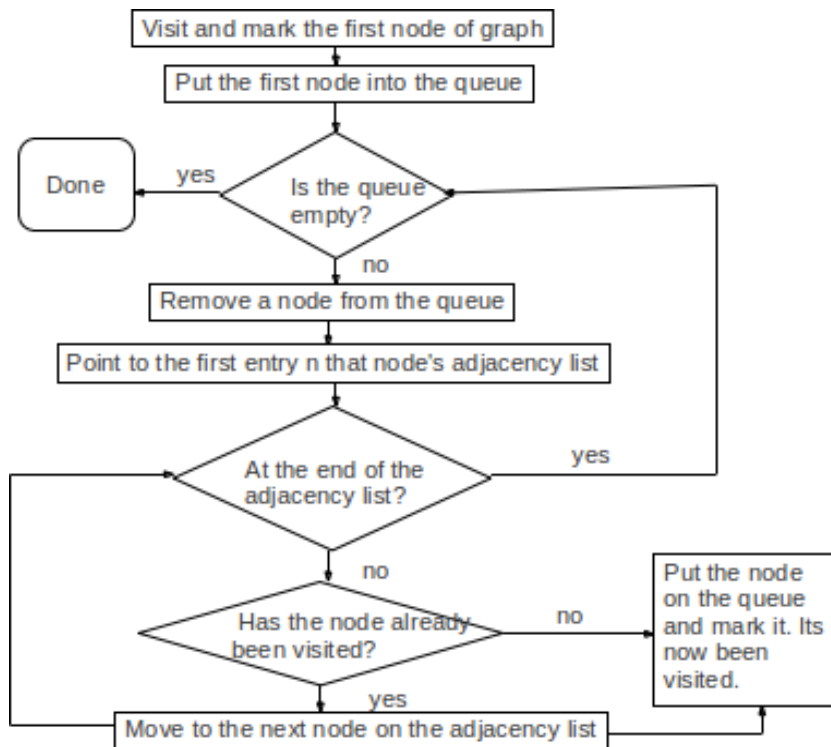


Fig 10.3

Outcome: - After performing this experiment, students will be able to have a good insight to Breadth first search (Graph traversal) and its application to find the connected components of a given graph.

EXPERIMENT NO. 11

Aim: Code and analyze to find shortest paths in a graph with positive edge weights using Dijkstra's algorithm.

Objective: To find Single Source Shortest Path in a directed graph that does not have any negative cost edges.

Procedure:

The idea of the algorithm is very simple.

1. It maintains a list of unvisited vertices.
2. It chooses a vertex (the source) and assigns a maximum possible cost (i.e. infinity) to every other vertex.
3. The cost of the source remains zero as it actually takes nothing to reach from the source vertex to itself.
4. In every subsequent step of the algorithm it tries to improve (minimize) the cost for each vertex. Here the cost can be distance, money or time taken to reach that vertex from the source vertex. The minimization of cost is a multi-step process.
 - a) For each unvisited neighbor (vertex 2, vertex 3, vertex 4) of the current vertex (vertex 1) calculate the new cost from the vertex (vertex 1).
 - b) For e.g. the new cost of vertex 2 is calculated as the minimum of the two (existing cost of vertex 2) or (sum of cost of vertex 1 + the cost of edge from vertex 1 to vertex 2)
5. When all the neighbors of the current node are considered, it marks the current node as visited and is removed from the unvisited list.
6. Select a vertex from the list of unvisited nodes (which has the smallest cost) and repeat step 4.
7. At the end there will be no possibilities to improve it further and then the algorithm ends

Algorithm:-

```
DIJKSTRA( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE( $G, s$ )
2  $S \leftarrow \emptyset$ 
3  $Q \leftarrow V[G]$ 
4 while  $Q \neq \emptyset$ 
5 do  $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6  $S \leftarrow S \cup \{u\}$ 
7 for each vertex  $v \in \text{Adj}[u]$ 
8 do RELAX( $u, v, w$ )
```

Flowchart:-

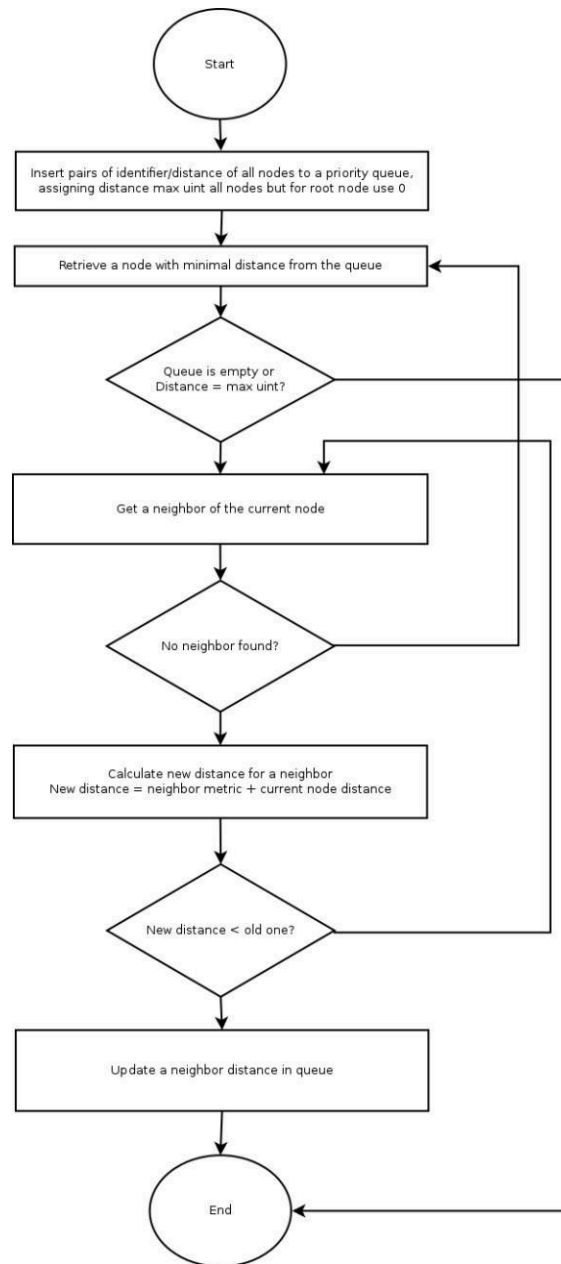


Fig 11.1

Outcome: - After performing this experiment, students will be able to have a good insight to Dijkstra's Algorithm and they will know Single Source Shortest path as an application of Greedy Method Strategy to design algorithms.

EXPERIMENT NO. 12

Aim:-Code and analyze to find shortest paths in a graph with arbitrary edge weights using Bellman Ford algorithm.

Objective: To find Single Source Shortest Path in a directed graph that does have any arbitrary weight / cost edges.

Procedure:

Given the following graph, calculate the length of the shortest path from **node 1** to **node 2**.

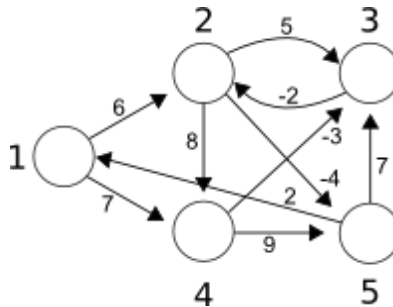


Fig 12.1

It's obvious that there's a direct route of length 6, but take a look at path: $1 \rightarrow 4 \rightarrow 3 \rightarrow 2$. The length of the path is $7 - 3 - 2 = 2$, which is less than 6. BTW, you don't need negative edge weights to get such a situation, but they do clarify the problem.

This also suggests a property of shortest path algorithms: to find the shortest path from x to y , you need to know, beforehand, the shortest paths to y 's neighbours. For this, you need to know the paths to y 's neighbours' neighbours. In the end, you must calculate the shortest path to the connected of the graph in which x and y are found.

The Bellman-Ford algorithm is one of the classic solutions to this problem. It calculates the shortest path to all nodes in the graph from a single source.

The basic idea is simple:

Start by considering that the shortest path to all nodes, less the source, is infinity. Mark the length of the path to the source as 0:

Take every edge and try to *relax* it:

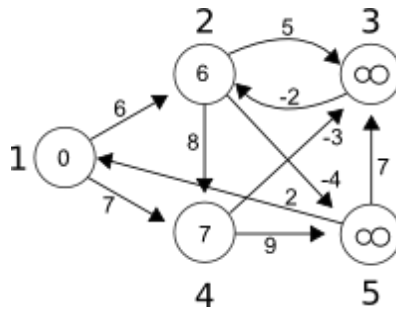
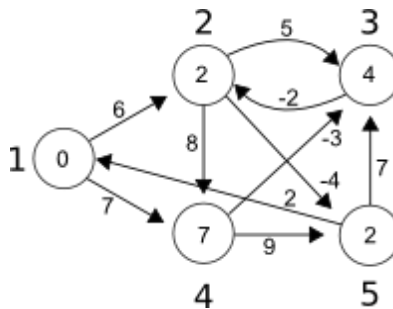
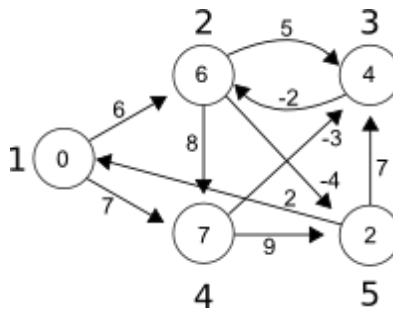


Fig 12.2

Relaxing an edge means checking to see if the path to the node the edge is pointing to can't be shortened, and if so, doing it. In the above graph, by checking the **edge 1 -> 2** of length 6, you find that the length of the shortest path to **node 1** plus the length of the **edge 1 -> 2** is less than infinity. So, you replace infinity in **node 2** with 6. The same can be said for edge 1 -> 4 of length 7. It's also worth noting that, practically, you can't relax the edges whose start has the shortest path of length infinity to it.

Now, you apply the previous step $n - 1$ times, where n is the number of nodes in the graph. In this example, you have to apply it 4 times (that's 3 more times).



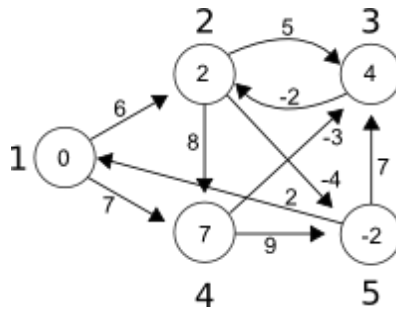


Fig 12.3

That's it, here's the algorithm in a condensed form:

Algorithm:-

```

BELLMAN-FORD ( $G, w, s$ )
1 INITIALIZE-SINGLE-SOURCE ( $G, s$ )
2 for  $i \leftarrow 1$  to  $|V[G]| - 1$ 
3   do for each edge  $(u, v) \in E[G]$ 
4     do RELAX ( $u, v, w$ )
5 for each edge  $(u, v) \in E[G]$ 
6 do if  $d[v] > d[u] + w(u, v)$ 
7 then return FALSE
8 return TRUE
  
```

Outcome: - Students will be able to find the shortest path in a graph from source to all other vertices even when the edges may have negative weights on their edges.

EXPERIMENT NO. 13

Aim:- Code and analyze to find the minimum spanning tree in a weighted, undirected graph.

Objective:- To study and implement MST using Prim's Algorithm.

Procedure:-

Like Kruskal's algorithm, Prim's algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST; the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

A group of edges that connects two set of vertices in a graph is called cut in graph theory.

So, at every step of Prim's algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the vertices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim's Algorithm Work? 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*.

Let us understand with the following example:

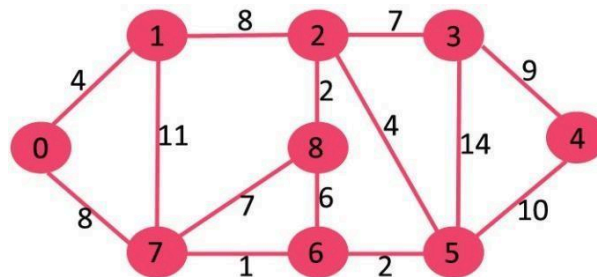


Fig. 13.1

The set *mstSet* is initially empty and keys assigned to vertices are {0, 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 green color.

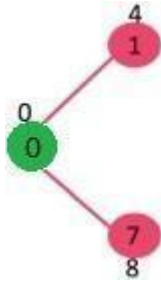


Fig. 13.2

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.

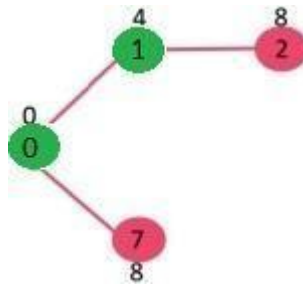


Fig. 13.3

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).

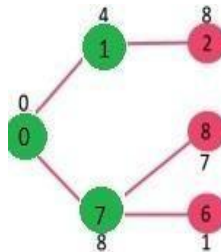


Fig. 13.4

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.

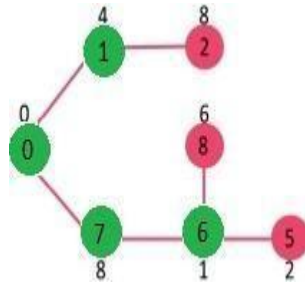


Fig. 13.5

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

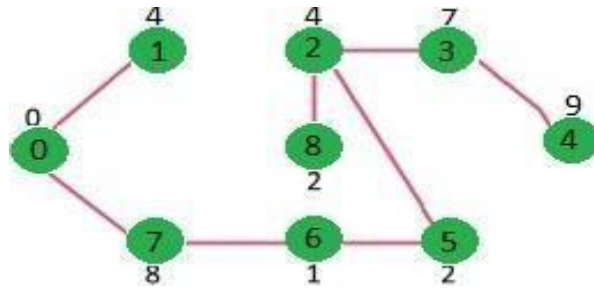


Fig. 13.6

Algorithm:-

PRIM'S ALGORITHM FOR IMPLEMENTING MINIMUM COST SPANNING TREE. E=SET OF EDGES IN G, N=NO. OF VERTICES, T=FINAL ARRAY MATRIX, FINALLY MINCOST IS RETURNED.

ALGORITHM(E, COST, N, T)

```
{
let (K, L) be an edge m cost in
E; mincost:=cost (K, L);
T [1,1]=K; T[1,2]=L;
for i=1 to N do;
if COST [i,L]<COST[i then near[i]=L;
else near [i]:=K;
near [K]=near [L]=0;
for i=2 to N-1 do;
{
let j be an index such that near [j]!=0
and cost [j,near[j]] is minimum;
T [i, 1]=j;T[i,2]=near[j];
mincost=mincost+cost [j,near[j])
near [j]: =0;
for K=1 to N do
if ((near [K]! =0and (cost
[K,near[K]>cost[K,j]) then near [K]=j;
}
return mincost;
}
```

Flowchart:-

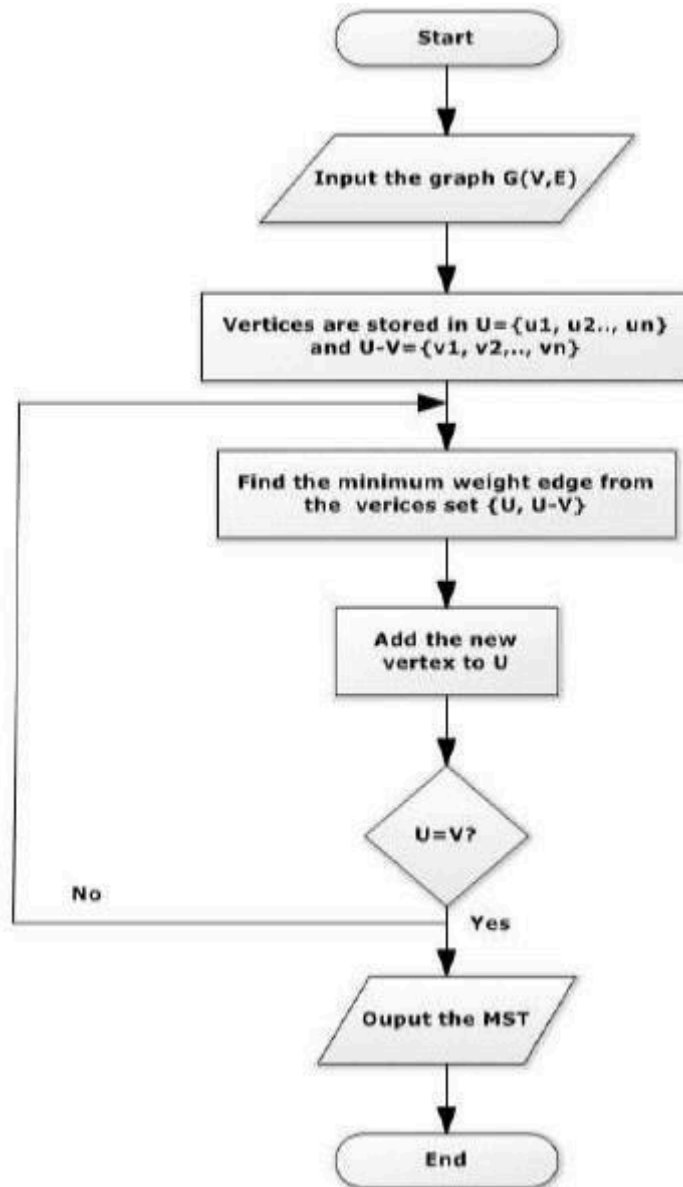


Fig. 13.7

Outcome:- Students will be able to find Minimum cost spanning tree of a given graph with Prim's strategy.

EXPERIMENT NO. 14

Aim:- Code and analyze to find all occurrences of a pattern P in a given string S.

Objective:- To find all occurrences of a pattern P in a given string S.

Procedure:- Given a text $txt[0..n-1]$ and a pattern $pat[0..m-1]$, write a function $search(char pat[], char txt[])$ that prints all occurrences of $pat[]$ in $txt[]$. You may assume that $n > m$.

Examples:

1) Input:

```
txt[] = "THIS IS A TEST TEXT"
```

```
pat[] = "TEST"
```

Output:

```
Pattern found at index 10
```

2) Input:

```
txt[] =
```

```
"AABAACAADAABAAABAA"
```

```
pat[] = "AABA"
```

Output:

```
Pattern found at index 0
```

```
Pattern found at index 9
```

```
Pattern found at index 13
```

Pattern searching is an important problem in computer science. When we do search for a string in notepad/word file or browser or database, pattern searching algorithms are used to show the search results.

KMP (Knuth Morris Pratt) Pattern Searching

The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to $O(n)$. The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text (since they matched the pattern

characters prior to the mismatch). We take advantage of this information to avoid matching the characters that we know will anyway match.

KMP algorithm does some preprocessing over the pattern `pat[]` and constructs an auxiliary array `lps[]` of size `m` (same as size of pattern).

Here **name lps indicates longest proper prefix which is also suffix..** For each sub-pattern `pat[0...i]` where `i = 0` to `m-1`, `lps[i]` stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern `pat[0..i]`.

`lps[i] = the longest proper prefix of pat[0..i]`
`which is also a suffix of pat[0..i].`

Examples:

For the pattern “AABAACAABAA”, `lps[]` is `[0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]`

For the pattern “ABCDE”, `lps[]` is `[0, 0, 0, 0, 0]`

For the pattern “AAAAA”, `lps[]` is `[0, 1, 2, 3, 4]`

For the pattern “AAABAAA”, `lps[]` is `[0, 1, 2, 0, 1, 2, 3]`

For the pattern “AAACAAAAC”, `lps[]` is `[0, 1, 2, 0, 1, 2, 3, 3, 3, 4]`

Algorithm:

Naive string matching:

```
for (i=0; T[i] != '\0';  
    i++)  
{  
    for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;  
    if (P[j] == '\0') found a match  
}
```

KMP, version 1:

```
i=0;  
o=0;  
while (i<n)  
{  
    for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;  
    if (P[j] == '\0') found a match;  
    o = overlap(P[0..j-1],P[0..m]);  
    i = i + max(1, j-o);  
}
```


KMP, version 2:

```
j = 0;
for (i = 0; i < n; i++)
for (;;) { // loop until break
    if (T[i] == P[j]) { //
        matches?
        j++; // yes, move on to next state
        if (j == m) { // maybe that was the last state
            found a match;
            j = overlap[j];
        }
        break;
    } else if (j == 0) break; // no match in state j=0, give up
    else j = overlap[j]; // try shorter partial match
}
```

KMP overlap computation:

```
overlap[0] = -1;
for (int i = 0; pattern[i] != '\0'; i++) {
    overlap[i + 1] = overlap[i] + 1;
    while (overlap[i + 1] > 0 &&
        pattern[i] != pattern[overlap[i + 1] - 1])
        overlap[i + 1] = overlap[overlap[i + 1] - 1] + 1;
}
return overlap;
```

Outcome:- Students will be able to find all occurrences of a pattern P in a given string S.

EXPERIMENT NO. 15

Aim:- Write a program to multiply two large integers using Karatsuba algorithm.

Objective: - To multiply two large integers using Karatsuba algorithm.

Procedure:

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is "1100" and second bit string is "1010", output should be 120.

For simplicity, let the length of two strings be same and be n .

A **Naive Approach** is to follow the process we study in school. One by one take all bits of second number and multiply it with all bits of first number. Finally add all multiplications. This algorithm takes $O(n^2)$ time.

$$\begin{array}{r} \mathbf{x} = 101001 = 41 \\ \mathbf{Y} = 101010 = 42 \\ \hline \\ \\ \\ \\ + 101001 \\ \hline 11010111010 = 1722 \end{array}$$

Using **Divide and Conquer**, we can multiply two integers in less time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y .

For simplicity let us assume that n is even

$$\begin{aligned} X &= X_l * 2^{n/2} + X_r && [X_l \text{ and } X_r \text{ contain leftmost and rightmost } n/2 \text{ bits of } X] \\ Y &= Y_l * 2^{n/2} + Y_r && [Y_l \text{ and } Y_r \text{ contain leftmost and rightmost } n/2 \text{ bits of } Y] \end{aligned}$$

The product XY can be written as following.

$$\begin{aligned} XY &= (X_l * 2^{n/2} + X_r)(Y_l * 2^{n/2} + Y_r) \\ &= 2^n X_l Y_l + 2^{n/2}(X_l Y_r + X_r Y_l) + X_r Y_r \end{aligned}$$

If we take a look at the above formula, there are four multiplications of size $n/2$, so we basically divided the problem of size n into four sub-problems of size $n/2$. But that doesn't help because solution of recurrence $T(n) = 4T(n/2) + O(n)$ is $O(n^2)$. The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

$$X_l Y_r + X_r Y_l = (X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r$$

So the final value of XY becomes

$$XY = 2^n XIYI + 2^{n/2} * [(Xl + Xr)(Yl + Yr) - XIYI - XrYr] + XrYr$$

With above trick, the recurrence becomes $T(n) = 3T(n/2) + O(n)$ and solution of this recurrence is $O(n^{1.59})$.

What if the lengths of input strings are different and are not even? To handle the different length case, we append 0's in the beginning. To handle odd length, we put $\text{floor}(n/2)$ bits in left half and $\text{ceil}(n/2)$ bits in right half. So the expression for XY changes to following.

$$XY = 2^{2\text{ceil}(n/2)} XIYI + 2^{\text{ceil}(n/2)} * [(Xl + Xr)(Yl + Yr) - XIYI - XrYr] + XrYr$$

The above algorithm is called Karatsuba algorithm and it can be used for any base.

Algorithm:

```

procedure karatsuba(num1, num2)
  if (num1 < 10) or (num2 < 10)
    return num1 * num2
  /* calculates the size of the numbers */
  m = max(size_base10(num1), size_base10(num2))
  m2 = m/2
  /* split the digit sequences about the middle */
  high1, low1 = split_at(num1, m2)
  high2, low2 = split_at(num2, m2)
  /* 3 calls made to numbers approximately half the size */
  z0 = karatsuba(low1, low2)
  z1 = karatsuba((low1 + high1), (low2 + high2))
  z2 = karatsuba(high1, high2)
  return (z2 * 10^(2 * m2)) + ((z1 - z2 - z0) * 10^(m2)) + (z0)

```

Flowchart:

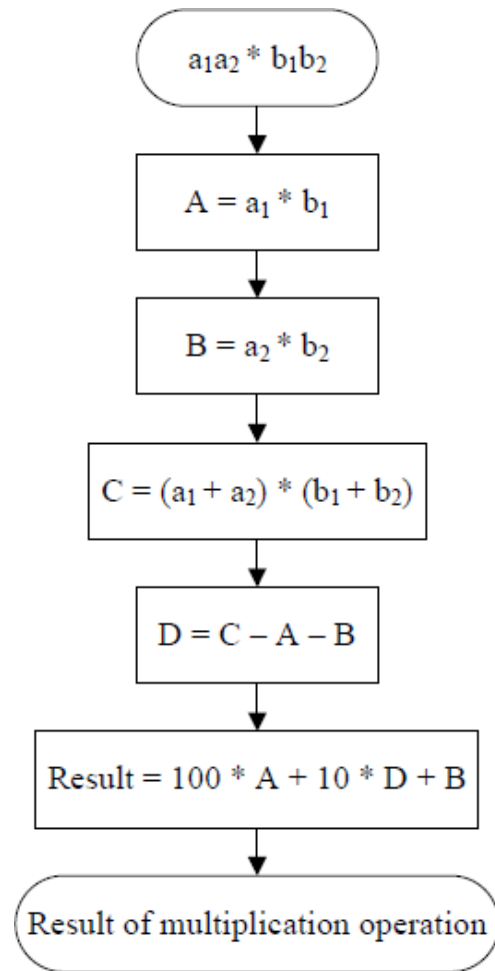


Fig. 15.1

Outcome:

After studying and implementing this experiment, students will be able to multiply two large numbers quickly and easily. They will also have a sound knowledge of Karatsuba's Algorithm for future purpose.

EXPERIMENT NO. 16

Aim: Write a program to compute the convex hull of a set of points in the plane.

Objective: To compute the convex hull of a set of points in the plane.

Procedure:

Given a set of points in the plane. The convex hull of the set is the smallest convex polygon that contains all the points of it.

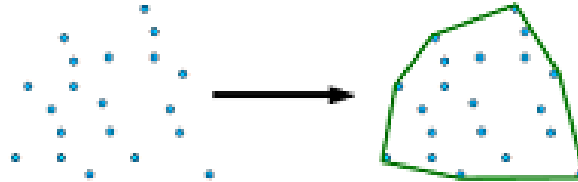


Fig. 16.1

Graham's Algorithm

Let $points[0..n-1]$ be the input array.

- 1) Find the bottom-most point by comparing y coordinate of all points. If there are two points with same y value, then the point with smaller x coordinate value is considered. Let the bottom-most point be P_0 . Put P_0 at first position in output hull.
- 2) Consider the remaining $n-1$ points and sort them by polar angle in counterclockwise order around $points[0]$. If polar angle of two points is same, then put the nearest point first.
- 3 After sorting, check if two or more points have same angle. If two more points have same angle, then remove all same angle points except the point farthest from P_0 . Let the size of new array be m .
- 4) If m is less than 3, return (Convex Hull not possible)
- 5) Create an empty stack 'S' and push $points[0]$, $points[1]$ and $points[2]$ to S.
- 6) Process remaining $m-3$ points one by one. Do following for every point ' $points[i]$ '
 - 6.1) Keep removing points from stack while orientation of following 3 points is not counterclockwise (or they don't make a left turn).
 - a) Point next to top in stack
 - b) Point at the top of stack
 - c) $points[i]$
 - 6.2) Push $points[i]$ to S
- 7) Print contents of S

The above algorithm can be divided in two phases.

Phase 1 (Sort points): We first find the bottom-most point. The idea is to pre-process points by sorting them with respect to the bottom-most point. Once the points are sorted, they form a simple closed path (See following diagram).

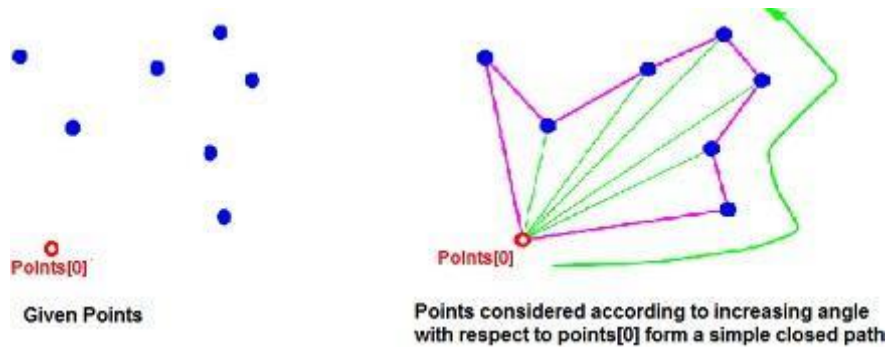
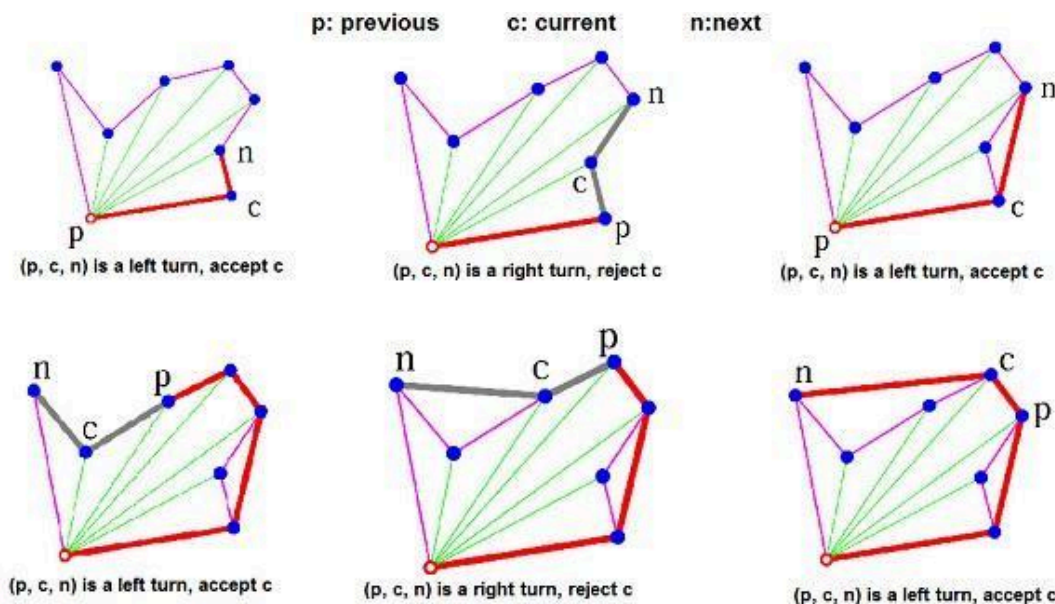


Fig. 16.2

What should be the sorting criteria? computation of actual angles would be inefficient since trigonometric functions are not simple to evaluate. The idea is to use the orientation to compare angles without actually computing them (See the compare() function below)

Phase 2 (Accept or Reject Points): Once we have the closed path, the next step is to traverse the path and remove concave points on this path. How to decide which point to remove and which to keep? Again, orientation helps here.

The first two points in sorted array are always part of Convex Hull. For remaining points, we keep track of recent three points, and find the angle formed by them. Let the three points be prev(p), curr(c) and next(n). If orientation of these points (considering them in same order) is not counterclockwise, we discard c, otherwise we keep it. Following diagram shows step by step process of this phase.



In the above algorithm and below code, a stack of points is used to store convex hull points. With reference to the code, p is next-to-top in stack, c is top of stack and n is points[i].

Fig. 16.3

Algorithm:

```
# Three points are a counter-clockwise turn if ccw > 0, clockwise if  
# ccw < 0, and collinear if ccw = 0 because ccw is a determinant that  
# gives twice the signed area of the triangle formed by p1, p2 and p3.  
function ccw(p1, p2, p3):  
    return (p2.x - p1.x)*(p3.y - p1.y) - (p2.y - p1.y)*(p3.x - p1.x)
```

Then let the result be stored in the array points.

```
le      = number of  
let points[N] # points is the array of points  
Swap points[1] with the point with the lowest y-coordinate  
sort points by polar angle with points[1]  
  
# We want points[0] to be a sentinel point that will stop the  
loop.  
let points[0] = points[N]  
  
# M will denote the number of points on the convex hull.  
let M = 1  
for i = 2 to N:  
    # Find next valid point on convex hull.  
    while ccw(points[M-1], points[M], points[i]) <= 0:  
        if M > 1:  
            M -= 1  
        # All points are collinear  
        else if i == N:  
            br  
        eak  
        else  
            i += 1  
  
# Update M and swap points[i] to the correct place.  
    M += 1  
    swap points[M] with points[i]
```

Input: a set of points $S = \{P = (P.x, P.y)\}$

Select the rightmost lowest point P_0 in S
Sort S radially (ccw) about P_0 as a center {

```

Use isLeft() comparisons
  For ties, discard the closer points
}
Let P[N] be the sorted array of points with P[0]=P0
Push P[0] and P[1] onto a stack Ω
while i < N
{
  Let PT1 = the top point on Ω
  If (PT1 == P[0]) {
    Push P[i] onto Ω
    i++ // increment i
  }
  Let PT2 = the second top point on Ω
  If (P[i] is strictly left of the line PT2 to PT1) {
    Push P[i] onto Ω
    i++ // increment i
  }
  else
    Pop the top point PT1 off the stack
}

```

Output: Ω = the convex hull of S.

Outcome:

After studying and implementing this experiment, students will be able compute the convex hull of a set of given points, and also they will have an insight of Graham's Algorithm. They may also perform this experiment by Divide and Conquer strategy.

EXPERIMENT NO. 17

Aim: Write a program to multiply two polynomials using Fast Fourier Transform.

Objective: To multiply two polynomials using Fast Fourier Transform.

```
Procedure: #include <cstdio>
#include <cstring>
#include <iostream>

using namespace std;
typedef long long ll;

// polynomial coefficients are saved in increasing order of degree
// coefficient of x**i in polynomial p = p[i]

// multiply polynomials p and q, both of size sz,
// where sz is multiple of 2
void karatsuba(ll *res, const ll *p, const ll *q, int
sz){ ll t0[sz], t1[sz], r[sz<<1];

memset(r, 0, (sz<<1) * sizeof(ll));

if ( sz <= 4 ){ // base case, no recursion, do basic school multiplication
    for ( int i = 0 ; i < sz ; i++ )
        for ( int j = 0 ; j < sz ; j++ ){
            r[i + j] += p[i] * q[j];
        }
} else {
    // let p = a*x**nSz + b
    //   q = c*x**nSz + d
    //   r = ac*x**sz + ((a+b)*(c+d) - ac - bd)*x**nSz +
    bd int nSz = (sz >> 1);

    for ( int i = 0 ; i < nSz ; i++ ){
        t0[i] = p[i] + p[nSz + i]; // t0 = a + b
        t1[i] = q[i] + q[nSz + i]; // t1 = c + d
        t0[i + nSz] = t1[i + nSz] = 0; // initialize
    }

    karatsuba(r + nSz, t0, t1, nSz); // r[nSz...sz] = (a+b) (c+d)
```

```

    karatsuba(t0, p, q, nSz);          // t0 = bd
    karatsuba(t1, p + nSz, q + nSz, nSz); // t1 =
    ac

    for ( int i = 0 ; i < sz ; i++ ){
        r[i] += t0[i];                // bd
        r[i + nSz] -= t0[i] + t1[i]; // ((a+b)(c+d) - ac - bd) * x**nSz
        r[i + sz] += t1[i];          // ac * x**sz
    }
}

memcpy(res, r, (sz<<1) * sizeof(ll));
}

// multiply two polynomials p and q, both of size sz = degree + 1
// save the output in array r
// NOTE: the maximum capacity of p, q, r should be power of two
// NOTE: r should be at least double of p or q in
size void polyMult(ll *r, ll *p, ll *q, int sz){
    if ( sz & (sz - 1) ){ // if size is not power of
        two int k = 1;
        while ( k < sz ) k <<= 1;
        while ( ++sz <= k ) p[sz - 1] = q[sz - 1] = 0;
        sz--;
    }

    karatsuba(r, p, q, sz);
}

// print polynomial in descending order of
degree void polyPrint(ll *p, int sz){
    while ( --sz >= 0 ) cout << p[sz] <<" ";
    puts("");
}

int main(){
    ll p[4] = {1,3,3,1};
    ll q[4] = {-1,3,-3,1};
    ll r[8];
    int degree = 3;

```

```
polyMult(r, p, q, degree + 1);
polyPrint(r, degree * 2 + 1);
return 0;
}
```

Algorithm:

FFT (a[0:n-1],n,w,b[0:n-1])

Input: a[0:n-1] (an array of coefficients of the polynomial $P(x) = a_{n-1}x^{n-1} + \dots + a_1x + a_0$),

n (a positive integer) $\{n=2^k\}$, w(a positive n^{th} root of the unity)

Output: b[0:n-1] (an array of values $b[i] = P(w^i)$, $i=0, \dots, n-1$)

Call ReverseBinPerm(R[0:n-1])

for i=0 to n-1

b[i]=a[R[i]]

end for

Outcome: After studying and implementing this experiment, students will be able to multiply two polynomials using FFT which is based on Dynamic programming. Also, they will have a better insight of the advanced problems and their solutions in Algorithms.

**EXPERIMENTS
BEYOND SYLLABUS**

EXPERIMENT NO. 18

Aim: Write a program to convert the infix expression into postfix expression.

Objective: To understand the concept of infix to postfix conversion.

Procedure:

There is an algorithm to convert an infix expression into a postfix expression. It uses a stack; but in this case, the stack is used to hold operators rather than numbers. The purpose of the stack is to reverse the order of the operators in the expression. It also serves as a storage structure, since no operator can be printed until both of its operands have appeared. In this algorithm, all operands are printed (or sent to output) when they are read. There are more complicated rules to handle operators and parentheses.

Example:

1. $A * B + C$ becomes $A B * C +$

The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.

We will show this in a table with three columns. The first will show the symbol currently being read. The second will show what is on the stack and the third will show the current contents of the postfix string. The stack will be written from left to right with the 'bottom' of the stack to the left.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

The rule used in lines 1, 3 and 5 is to print an operand when it is read. The rule for line 2 is to push an operator onto the stack if it is empty. The rule for line 4 is if the operator on the top of the stack has higher precedence than the one being read, pop

and print the one on top and then push the new operator on. The rule for line 6 is that when the end of the expression has been reached, pop the operators on the stack one at a time and print them.

2. $A + B * C$ becomes $A B C * +$

Here the order of the operators must be reversed. The stack is suitable for this, since operators will be popped off in the reverse order from that in which they were pushed.

	current symbol	operator stack	postfix string
1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

In line 4, the '*' sign is pushed onto the stack because it has higher precedence than the '+' sign which is already there. Then when they are both popped off in lines 6 and 7, their order will be reversed.

3. $A * (B + C)$ becomes $A B C + *$

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(* (A B
4	B	* (A B
5	+	* (+	A B
6	C	* (+	A B C
7)	*	A B C +
8			A B C + *

Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker. When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

4. $A - B + C$ becomes $A B - C +$

When operators have the same precedence, we must consider association. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. $A * B ^ C + D$ becomes $A B C ^ * D +$

Here both the exponentiation and the multiplication must be done before the addition.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B

5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack. Since it has lower precedence, the '^' is popped and printed. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. $A * (B + C * D) + E$ becomes $A B C D * + * E +$

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(* (A
4	B	* (A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

A summary of the rules follows:

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

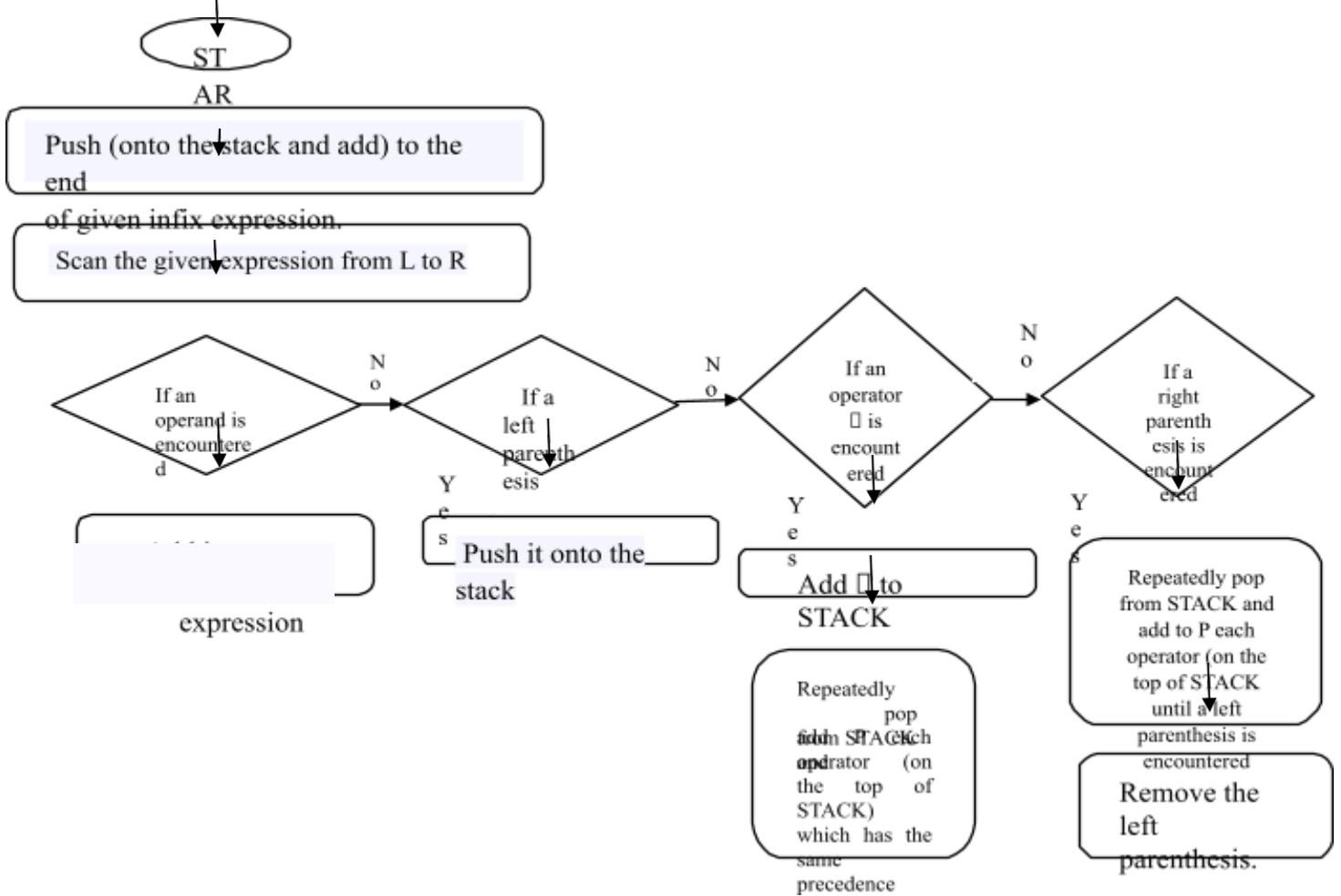
Algorithm:

Let Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push (“onto STACK, and add”)” to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the stack is empty.
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator \otimes is encountered, then :
 - (a) Add \otimes to STACK.
[End of If].
 - (b) Repeatedly pop from STACK and add P each operator (on the top of STACK) which has the same precedence as or higher precedence than \otimes .
6. If a right parenthesis is encountered, then :

- (a) Repeatedly pop from STACK and add to P each operator (on the top of STACK until a left parenthesis is encountered).
- (b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
- [End of if]
- [End of Step 2 loop].
- 7. Exit.

Flowchart:



Outcome: After studying and implementing this experiment, students will be able to convert infix expressions into postfix expressions using stack. This will also help them to understand the stack data structure from application point of view.

EXPERIMENT NO. 19

Aim: Write a program to evaluate postfix expression.

Objective: To understand the concept of evaluation of a given postfix expression.

Procedure:

The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed infix to postfix conversion. In this post, evaluation of postfix expressions is discussed.

Following is algorithm for evaluation postfix expressions.

1) Create a stack to store operands (or values).

2) Scan the given expression and do following for every scanned element.

.....a) If the element is a number, push it into the stack

.....b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack

3) When the expression is ended, the number in the stack is the final answer

Example:

Let the given expression be “2 3 1 * + 9 -“. We scan all elements one by one.

1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’

2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)

3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’

4) Scan ‘*’, it’s an operator, pop two operands from stack, apply the * operator on operands, we get $3*1$ which results in 3. We push the result ‘3’ to stack. Stack now becomes ‘2 3’.

5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get $3 + 2$ which results in 5. We push the result ‘5’ to stack. Stack now becomes ‘5’.

6) Scan ‘9’, it’s a number, we push it to the stack. Stack now becomes ‘5 9’.

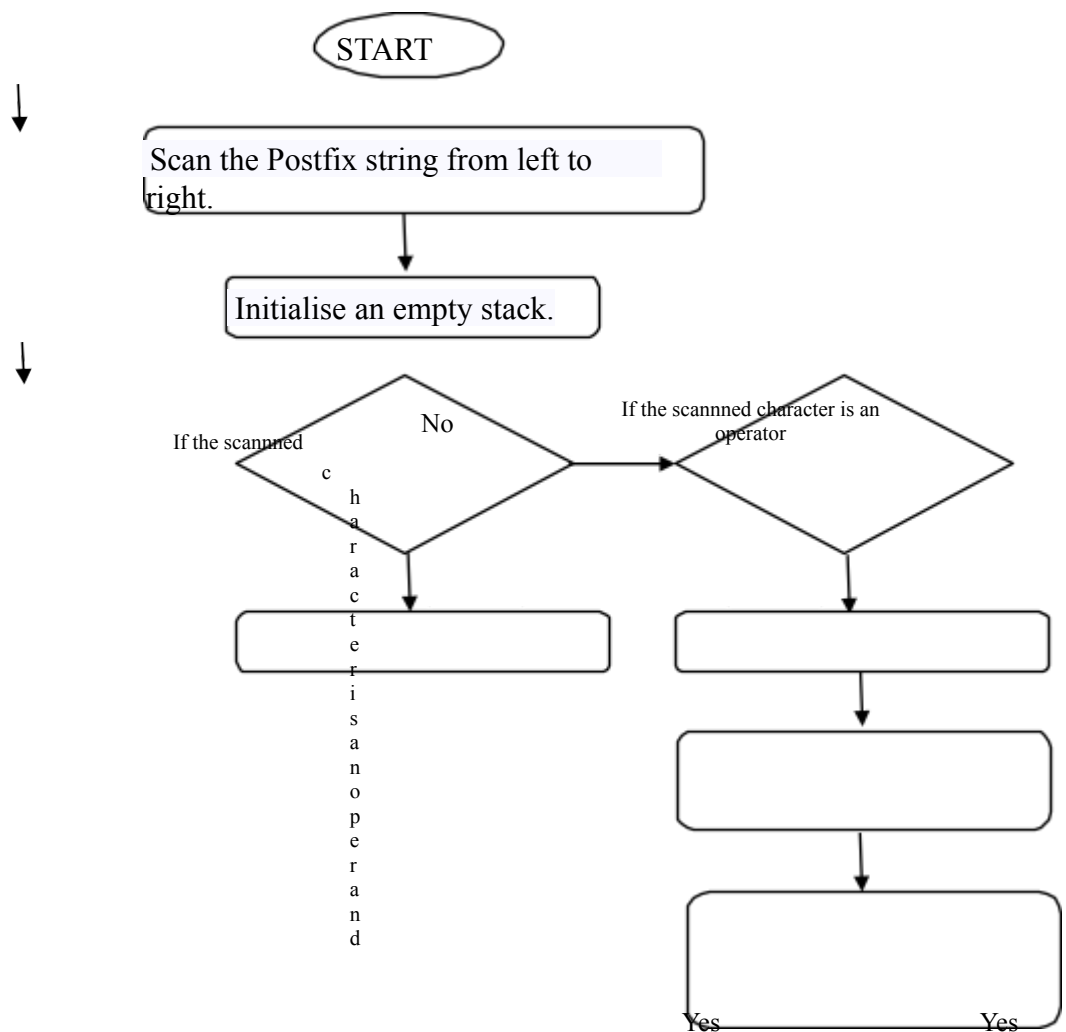
7) Scan ‘-’, it’s an operator, pop two operands from stack, apply the – operator on operands, we get $5 - 9$ which results in -4. We push the result ‘-4’ to stack. Stack now becomes ‘-4’.

8) There are no more elements to scan, we return the top element from stack (which is the only element left in stack).

Algorithm:

- Scan the Postfix string from left to right.
- Initialise an empty stack.
- If the scanned character is an operand, add it to the stack. If the scanned character is an operator, there will be atleast two operands in the stack.
 - If the scanned character is an Operator, then we store the top most element of the stack(topStack) in a variable temp. Pop the stack. Now evaluate topStack(Operator)temp. Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.
 - Repeat this step till all the characters are scanned.
- After all characters are scanned, we will have only one element in the stack. Return topStack.

Flowchart:



Repeat these steps

until last element is not popped from the stack

Add it to stack.

Pop up the stack

store this

element in a

variable
temp

Evaluate the operator
and Push the result back
to the stack.

Outcome: After studying and implementing this experiment, students will be able to evaluate postfix expressions using stack. This will also give them a better insight to applications of stack.