Practical- 1

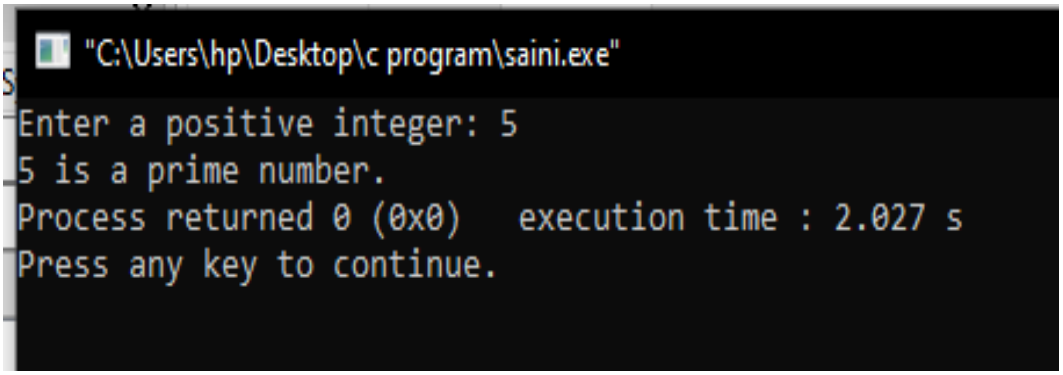
AIM- Write a program to find a prime number in C

THEORY : Prime numbers are numbers greater than 1 that only have two factors, 1 and the number itself. This means that a prime number is only divisible by 1 and itself. If you divide a prime number by a number other than 1 and itself, you will get a non-zero remainder.

Source Code :

```
#include <stdio.h>-

int main()

 {

int n, i, flag = 0;

printf("Enter a positive integer: ");

scanf("%d", &n);

  // 0 and 1 are not prime numbers

  // change flag to 1 for non-prime number

if (n == 0 || n == 1)

flag = 1;

for (i = 2; i <= n / 2; ++i) {

    // if n is divisible by i, then n is not prime

    // change flag to 1 for non-prime number

if (n % i == 0) {

flag = 1;

      break;


    }
```

```
  }
  // flag is 0 for prime numbers
if (flag == 0)
printf("%d is a prime number.", n);
 else
printf("%d is not a prime number.", n);
  return 0;
}
```



PRACTICAL – 2

AIM- Write a program to find a factorial of a number in C

THEORY : The factorial of a number is the multiplication of all the numbers between 1 and the number itself. It is written like this: $n!$. So the factorial of 2 is $2!$ (= 1 × 2).

Source Code :

```
#include <stdio.h>

int main() {

   int n, i;

   unsigned long long fact = 1;

   printf("Enter an integer: ");

   scanf("%d", &n);

   // shows error if the user enters a negative integer

   if (n < 0)

      printf("Error! Factorial of a negative number doesn't exist.");

   else {

      for (i = 1; i <= n; ++i) {

         fact *= i;

      }

      printf("Factorial of %d = %llu", n, fact);

   }

   return 0;

}
```

```
Enter an integer: 6
Factorial of 6 = 720
Process returned 0 (0x0)   execution time : 3.571 s
Press any key to continue.
```

PRACTICAL – 3

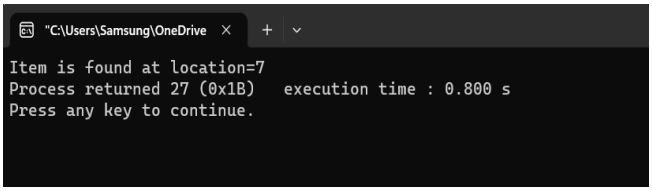
AIM - Write a program for linear search method.

THEORY: Linear Search is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set.

Source Code:

```
#include <stdio.h>

void main()

{

   int flag=0,array[10]={12,15,16,13,14,6,2,9,3,8};

   int l=10 ;

   int item=9,i=0;

   while (i<=l-1)

   {

      if (array[i]==item)

      {

         flag=1;

         break;

      }

   else

   {

      i++;

   }

   }
```

```
    if (flag==1)
    {
     printf("Item is found at location=%d",i);
    }
    else
    {
       printf("Item not found");
    }
}
```



PRACTICAL – 4

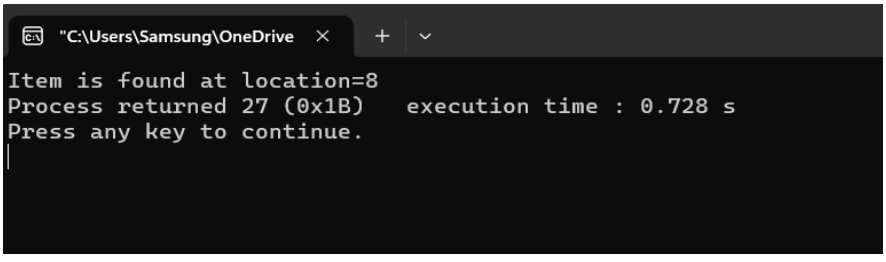
AIM - Write a program for binary search method.

THEORY: Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(log N).

Source Code:

```
#include <stdio.h>
void main()
{
  int array[10]={10,20,30,40,50,60,70,80,90,100};
  int lower=0,upper=9,flag=0;
  int item=90,mid;
  while(lower<=upper)
  {
     mid=(lower+upper)/2;
     if (array[mid]==item)
     {
        flag=1;
        break;
     }
     else
     {
        if (array[mid]<item)
```

```
      {
         lower=mid+1;
      }
      else
       {
       lower=mid-1;
       }
   }
}
if (flag==1)
{
   printf("Item is found at location=%d",mid);
 }
 else
 {
    printf("Item not found");
 }
}
```



PRACTICAL – 5

AIM – Write a program for insertion sort, selection sort and bubble sort.

THEORY – Selection sort: repeatedly pick the smallest element to append to the result. Insertion sort: repeatedly add new element to the sorted result. Bubble sort: repeatedly compare neighbor pairs and swap if necessary.

Source Code:

```
// Sorting array using Insertion sort, Bubble sort and Selection sort.

#include<stdio.h>

void insertion(int len,int arr[len])

{

   int i,j,key,temp;

   for(j=1;j<=len-1;j++)

   {

     i=j;

     while (i>0&&arr[i-1]>arr[i])

     {

      temp=arr[i];

      arr[i]=arr[i-1];

      arr[i-1]=temp;

      i--;

     }

   }

   printf("Sorted Array is \n");

for(int k=0;k<len;k++)
```

```
    {
        printf("%d\n",arr[k]);
    }
}
void selection(int len,int arr[len])
{
int i,j,smallest,temp;
for(j=0;j<len-1;j++)
{
    smallest=j;
    for(i=j+1;i<len;i++)
        {
            if(arr[i]<arr[smallest])
            {
                smallest=i;
            }
        }
    if (smallest!=j)
        {
          temp=arr[j];
          arr[j]=arr[smallest];
          arr[smallest]=temp;
        }
}
printf("Sorted Array is \n");
```

```
    for(int k=0;k<len;k++)
    {
        printf("%d\n",arr[k]);
    }
}
void bubble(int len,int arr[len])
{
    int i,k,n,flag=0,temp;
        n=len;
    for (k=1;k<n;k++)
    {
        for (i=0;i<n-k+1;i++)
        {
            if (arr[i]>arr[i+1])
            {
                temp=arr[i];
                arr[i]=arr[i+1];
                arr[i+1]=temp;
                flag=1;
                if(flag==0)
                {
                    break;
                }

            }
```

```
        }
    }
printf("Sorted Array is \n");
    for(int k=0;k<len;k++)
    {
        printf("%d\n",arr[k]);
    }
}
void main()
{
    int i,len,sort;
    printf("Enter the length of array=");
    scanf("%d",&len);
    int a[len];
    printf("\nEnter the elements of array=");
    for(i=0;i<=len-1;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Which type of sorting you want ?\n 1 for Insertion sort.\n 2 for Selection sort.\n 3 for Bubble
sort.\n");
    scanf("%d",&sort);
    switch(sort)
    {
        case 1: insertion(len,a);
        break;
```

```
        case 2: selection(len,a);
        break;
        case 3: bubble(len,a);
        break;
    }
}
```

```
Enter the length of array=5
Enter the elements of array=7
4
1
6
2
Which type of sorting you want ?
  1 for Insertion sort.
  2 for Selection sort.
  3 for Bubble sort.
1
Sorted Array is
1
2
4
6
7
```

PRACTICAL – 6

AIM- Write a program to implement Stack and its operation.

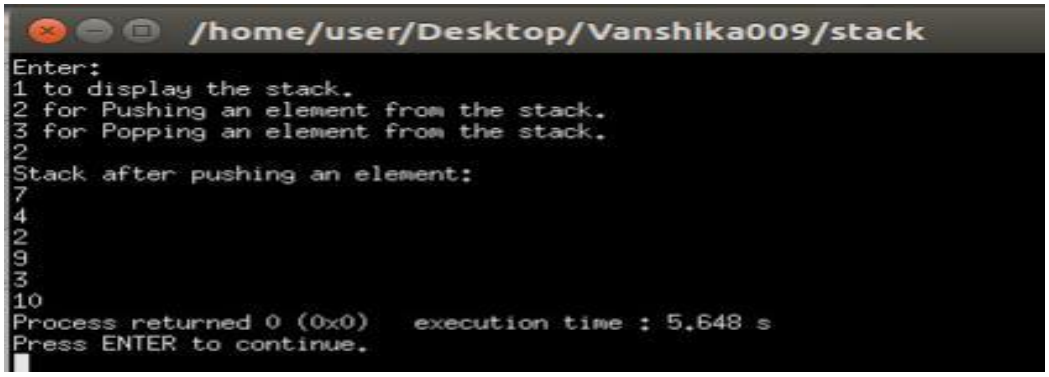
THEORY- A stack is a linear data structure in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.

Source Code:

```
#include<stdio.h>

void push()

{

   int stack[6]={7,4,2,9,3},Top=4,Maxstack=5,Item=10;

   if (Top==Maxstack)

   {

      printf(“Stack Overflow”);

   }

   else

   {

      Top=Top+1;

      stack[Top]=Item;

      printf(“Stack after pushing an element: “);

     for(int i=0;i<=Top;i++)

   {

    Printf (“\n%d”,stack[i]);

   }

      }
```

```
}
void pop()
{
   int Top=4,stack[6]={7,4,2,9,3},Item;
   if(Top==0)
   {
      printf(“Stack underflow”);
   }
   else
   {
      Item=stack[Top];
      Top=Top-1;
      printf(“Stack after popping an element: “);
     for(int i=0;i<=Top;i++)
   {
    printf(“\n%d”,stack[i]);
   }
   }

}
void show()
{
   int a=4,stack[6]={7,4,2,9,3};
   for(int i=0;i<=a;i++)
   {
```

```
    printf("\n%d",stack[i]);
  }
}
void main()
{
  int choice;
  printf("Enter: \n1 to display the stack.\n2 for Pushing an element from the stack.\n3 for Popping an element from the stack.\n");
  scanf("%d",&choice);
  switch(choice)
  {
  case 1:show();
  break;
  case 2:push();
  break;
  case 3:pop();
  break;
  }
}
```



PRACTICAL – 7

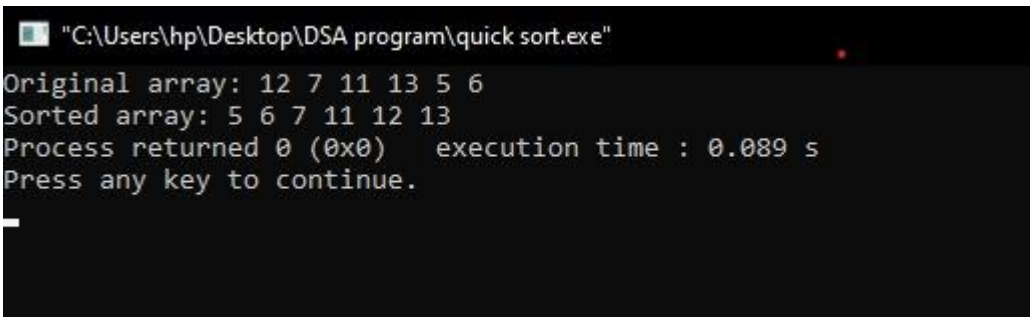
AIM- Write a program for quick sort.

THEORY-Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

Source Code:

```
#include <stdio.h>

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

int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = low - 1;


   for (int j = low; j < high; j++)
      if (arr[j] <= pivot)
         swap(&arr[++i], &arr[j]);


   swap(&arr[i + 1], &arr[high]);
   return i + 1;
}

void quickSort(int arr[], int low, int high) {
```

```
    if (low < high) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}

int main() {
    int arr[] = {12, 7, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    for (int i = 0; i < n; printf("%d ", arr[i++]));

    quickSort(arr, 0, n - 1);

    printf("\nSorted array: ");
    for (int i = 0; i < n; printf("%d ", arr[i++]));

    return 0;
```


"C:\Users\hp\Desktop\DSA program\quick sort.exe"
Original array: 12 7 11 13 5 6
Sorted array: 5 6 7 11 12 13
Process returned 0 (0x0)   execution time : 0.089 s
Press any key to continue.

}

PRACTICAL – 8

AIM- Write a program for merge sort.

THEORY – Merge sort is a divide-and-conquer algorithm. This means that it breaks down a problem into smaller sub problems, solves the sub problems recursively, and then combines the solutions to the sub problems to solve the original problem.

Source Code:

```
#include <stdio.h>

void merge(int arr[], int left, int mid, int right) {

   int i, j, k;

   int n1 = mid - left + 1;

   int n2 = right - mid;

   int left_half[n1], right_half[n2];

   for (i = 0; i < n1; i++)

      left_half[i] = arr[left + i];

   for (j = 0; j < n2; j++)

      right_half[j] = arr[mid + 1 + j];

   i = j = 0;
```

```
 k = left;

 while (i < n1 && j < n2) {

    if (left_half[i] <= right_half[j]) {

       arr[k] = left_half[i];

       i++;

    } else {

       arr[k] = right_half[j];

       j++;

    }

    k++;

 }

 while (i < n1) {

    arr[k] = left_half[i];

    i++;

    k++;

 }

while (j < n2) {
```

```
        arr[k] = right_half[j];
        j++;
        k++;
    }
}
void merge_sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
int main() {
    int my_array[] = {38, 27, 43, 3, 9, 82, 10};
    int array_size = sizeof(my_array) / sizeof(my_array[0]);
    printf ("Original array: ");
```

```
    for (int i = 0; i < array_size; i++)
        printf("%d ", my_array[i]);
    merge_sort(my_array, 0, array_size - 1);
    printf("\nSorted array: ");
    for (int i = 0; i < array_size; i++)
        printf("%d ", my_array[i]);
    return 0;
}
```

```
Original array: 38 27 43 3 9 82 10
Sorted array: 3 9 10 27 38 43 82
Process returned 0 (0x0)   execution time : 0.128 s
Press any key to continue.
```

PRACTICAL – 9

AIM- Write a program for BFS.

THEORY- BFS, or Breadth-First Search, is a node method for obtaining the graph's shortest path. It makes use of a queue data structure with FIFO (first in, first out) ordering.

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

struct node {

   int vertex;

   struct node* next;

};

struct adj_list {

   struct node* head;

};

struct graph {

   int num_vertices;

   struct adj_list* adj_lists;

   int* visited;

};

struct node* new_node(int vertex) {

   struct node* new_node = (struct node*)malloc(sizeof(struct node));

new_node->vertex = vertex;

new_node->next = NULL;

   return new_node;
```

```
}
struct graph* create_graph(int n) {
   struct graph* graph = (struct graph*)malloc(sizeof(struct graph));
   graph->num_vertices = n;
   graph->adj_lists = (struct adj_list*)malloc(n * sizeof(struct adj_list));
   graph->visited = (int*)malloc(n * sizeof(int));
   int i;
   for (i = 0; i< n; i++) {
      graph->adj_lists[i].head = NULL;
      graph->visited[i] = 0;
   }
   return graph;
}
void add_edge(struct graph* graph, int src, int dest) {
   struct node* new_node1 = new_node(dest);
   new_node1->next = graph->adj_lists[src].head;    graph->adj_lists[src].head = new_node1;
   struct node* new_node2 = new_node(src);
   new_node2->next = graph->adj_lists[dest].head;
   graph->adj_lists[dest].head = new_node2;
}
void bfs(struct graph* graph, int v) {
   int queue[1000];
   int front = -1;
   int rear = -1;
   graph->visited[v] = 1;
```

```
    queue[++rear] = v;
    while (front != rear) {
        int current_vertex = queue[++front];
printf("%d ", current_vertex);
        struct node* temp = graph->adj_lists[current_vertex].head;
        while (temp != NULL) {
            int adj_vertex = temp->vertex;
            if (graph->visited[adj_vertex] == 0) {
                graph->visited[adj_vertex] = 1;
                queue[++rear] = adj_vertex;
            }
            temp = temp->next;
        }
    }
}

int main() {
    struct graph* graph = create_graph(6);
add_edge(graph, 0, 1);
add_edge(graph, 0, 2);
add_edge(graph, 1, 3);
add_edge(graph, 1, 4);
add_edge(graph, 2, 4);
add_edge(graph, 3, 4);
add_edge(graph, 3, 5);
```

```
add_edge(graph, 4,5);

printf("BFS traversal starting from vertex 0: ");

bfs(graph, 0);

   return 0;

}
```

```
BFS traversal starting from vertex 0: 0 2 1 4 3 5
Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

PRACTICAL – 10

AIM- Write a program for DFS.

THEORY- Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 5

void addVertex(char);

void addEdge(int,int );

void displayVertex(int);

void depthFirstSearch();

int getAdjUnvisitedVertex(int);

struct Vertex {

  char label;

  bool visited;

};

//stack variables

int stack[MAX];

int top = -1;
```

```
//graph variables
//array of vertices
struct Vertex* lstVertices[MAX];
//adjacency matrix
int adjMatrix[MAX][MAX];
//vertex count
int vertexCount = 0;
//stack functions
void push(int item) {
  stack[++top] = item;
}
int pop() {
  return stack[top--];
}
int peek() {
  return stack[top];
}
bool isStackEmpty() {
  return top == -1;
}
//graph functions
//add vertex to the vertex list
void addVertex(char label) {
  struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
  vertex->label = label;
```

```
   vertex->visited = false;
   lstVertices[vertexCount++] = vertex;
}
//add edge to edge array
void addEdge(int start,int end) {
   adjMatrix[start][end] = 1;
   adjMatrix[end][start] = 1;
}
//display the vertex
void displayVertex(int vertexIndex) {
   printf("%c ",lstVertices[vertexIndex]->label);
}
//get the adjacent unvisited vertex
int getAdjUnvisitedVertex(int vertexIndex) {
   int i;
   for(i = 0; i < vertexCount; i++) {
      if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false) {
         return i;
      }
   }
   return -1;
}
void depthFirstSearch() {
   int i;
   //mark first node as visited
```

```
  lstVertices[0]->visited = true;
  //display the vertex
  displayVertex(0);
  //push vertex index in stack
  push(0);
  while(!isStackEmpty()) {
    //get the unvisited vertex of vertex which is at top of the stack
    int unvisitedVertex = getAdjUnvisitedVertex(peek());
    //no adjacent vertex found
    if(unvisitedVertex == -1) {
      pop();
    } else {
      lstVertices[unvisitedVertex]->visited = true;
      displayVertex(unvisitedVertex);
      push(unvisitedVertex);
    }
  }
  //stack is empty, search is complete, reset the visited flag
  for(i = 0;i < vertexCount;i++) {
    lstVertices[i]->visited = false;
  }
}
int main() {
  int i, j;
  for(i = 0; i < MAX; i++) // set adjacency {
```

```
    for(j = 0; j < MAX; j++) // matrix to 0
      adjMatrix[i][j] = 0;
    addVertex('S'); // 0
    addVertex('A'); // 1
    addVertex('B'); // 2
    addVertex('C'); // 3
    addVertex('D'); // 4
    addEdge(0, 1); // S - A
    addEdge(0, 2); // S - B
    addEdge(0, 3); // S - C
    addEdge(1, 4); // A - D
    addEdge(2, 4); // B - D
    addEdge(3, 4); // C - D
    printf("Depth First Search: ");
    depthFirstSearch();
return 0;
}
```

```
Depth First Search: S A D B C
Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```