

# **Indira Gandhi Delhi Technical University for Women**

**(Established by Govt. of Delhi vide Act 09 of 2012) (Formerly Indira  
Gandhi Institute of Technology) Kashmere Gate, Delhi - 110006**

---



## **LABORATORY MANUAL**

**For**

## **Design and Analysis of Algorithms Lab**

**Submitted to:**

Ms. Bhawna Narwal

**Submitted by:**

Radhika  
(00304092019)  
MCA (II year)

**1(a) Implement Recursive Binary search and Linear search and determine the time taken to search an element**

**Binary Search**

```
#include<iostream>
#include<time.h>
using namespace std;
clock_t begin, end;
double time_spent;

int binarySearch(int arr[], int low, int high, int num)
{
    if (low <= high)
    {
        int mid = (low + high)/2;
        if (arr[mid] == num)
            return mid ;
        else if (arr[mid] > num)
            return binarySearch(arr, low, mid-1, num); else
            return binarySearch(arr, mid+1, high, num);
    }
    return -1;
}

int main()
{
    int arr[50],size,f1,num;
    cout<<"Enter the size of an array(max size:50): ";
    cin>>size;
    cout<<"\nEnter the array (In sorted array): ";
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }
    cout<<"\nEnter the element to be searched: ";
    cin>>num;
    begin = clock();
    f1 = binarySearch(arr,0,size-1,num);
    if(f1 == -1)
```

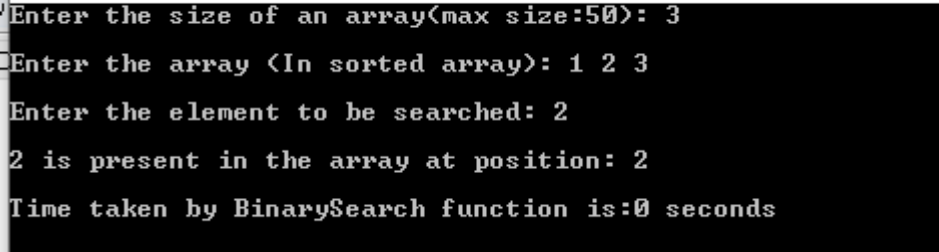
```

        cout<<"\n"<<num<<" is not present in the array"; else
        cout<<"\n"<<num<<" is present in the array at position: "<<f1+1; end =
        clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        cout<<"\n\nTime taken by BinarySearch function is:
"<<time_spent<<" seconds";

return 0;
}

```

#### OUTPUT-



```

C:\Users\Radhika\Desktop\DAA\daa.exe
Enter the size of an array(max size:50): 3
Enter the array (In sorted array): 1 2 3
Enter the element to be searched: 2
2 is present in the array at position: 2
Time taken by BinarySearch function is:0 seconds

```

#### Analysis

##### Time Complexity-

Best Case -  $O(1)$

Average Case –  $O(\log n)$

Worst Case -  $O(\log n)$

**Recurrence Relation-**  $T(n) = T(n/2) + 1, n > 1$   $T(0) = 0, n = 0$

$$T(1) = 1, n = 1$$

## **Linear Search**

```
#include<iostream>
#include<time.h>
using namespace std;
clock_t begin, end;
double time_spent;

int linearSearch(int arr[],int num,int index,int size)
{
    if(index<=size)
    {
        if(arr[index] == num)
        {
            return index;
        }
    }
    else
    {
        return linearSearch(arr, num, index+1, size);
    }
}

return -1 ;
}

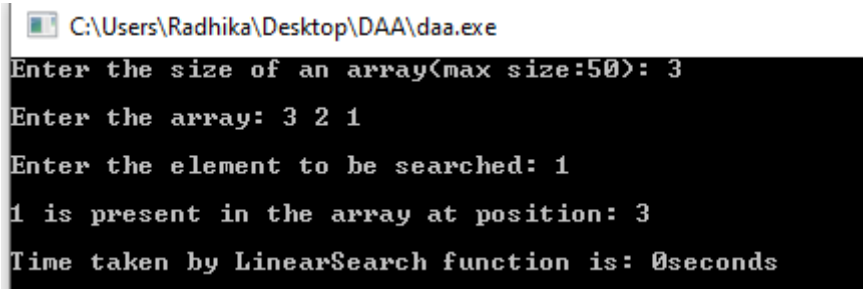
int main()
{
    int arr[50],size,f1,num;
    cout<<"Enter the size of an array(max size:50): ";
    cin>>size;
    cout<<"\nEnter the array: ";
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }
    cout<<"\nEnter the element to be searched: ";
    cin>>num;
    begin = clock();
```

```

f1 = linearSearch(arr,num,0,size);
    if(f1 == -1)
        cout<<"\n"<<num<<" is not present in the array"; else
        cout<<"\n"<<num<<" is present in the array at position: "<<f1+1; end =
        clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC; cout<<"\n\nTime taken
        by LinearSearch function is: "<<time_spent<<"
        seconds";
        return 0;
    }

```

OUTPUT-



```

C:\Users\Radhika\Desktop\DAA\daa.exe
Enter the size of an array(max size:50): 3
Enter the array: 3 2 1
Enter the element to be searched: 1
1 is present in the array at position: 3
Time taken by LinearSearch function is: 0seconds

```

## Analysis

### Time Complexity-

Best Case -  $O(1)$   
 Average Case –  $O(n)$   
 Worst Case -  $O(n)$

**Recurrence Relation-**  $T(n) = T(n-1) + 1, n > 1$   $T(0) = 0, n = 0$

$T(1) = 1, n = 1$

**(b) Use divide and conquer method to recursively implement Binary Search**

**rch**

### **Binary Search**

```
#include<iostream>
#include<time.h>
using namespace std;
clock_t begin, end;
double time_spent;

int binarySearch(int arr[], int low, int high, int num)
{
    if (low <= high)
    {
        int mid = (low + high)/2;
        if (arr[mid] == num)
            return mid ;
        else if (arr[mid] > num)
            return binarySearch(arr, low, mid-1, num); else
            return binarySearch(arr, mid+1, high, num);
    }
    return -1;
}

int main()
{
    int arr[50],size,f1,num;
    cout<<"Enter the size of an array(max size:50): ";
    cin>>size;
    cout<<"\nEnter the array (In sorted array): ";
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }
    cout<<"\nEnter the element to be searched: ";
    cin>>num;
    begin = clock();
    f1 = binarySearch(arr,0,size-1,num);
    if(f1 == -1)
```

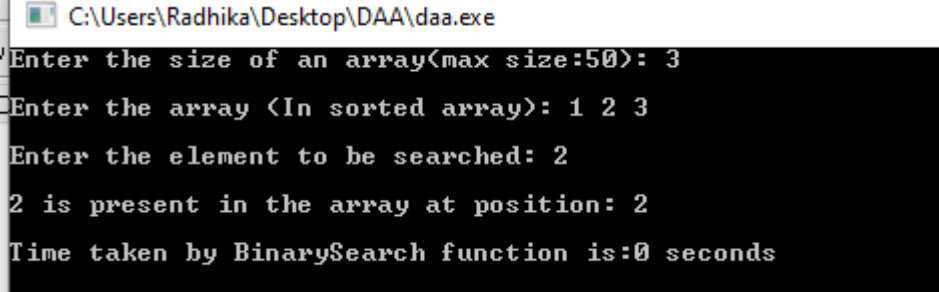
```

        cout<<"\n"<<num<<" is not present in the array"; else
        cout<<"\n"<<num<<" is present in the array at position: "<<f1+1; end =
        clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        cout<<"\n\nTime taken by BinarySearch function is:
"<<time_spent<<" seconds";

return 0;
}

```

#### OUTPUT-



```

C:\Users\Radhika\Desktop\DAA\daa.exe
Enter the size of an array(max size:50): 3
Enter the array (In sorted array): 1 2 3
Enter the element to be searched: 2
2 is present in the array at position: 2
Time taken by BinarySearch function is:0 seconds

```

#### Analysis

##### Time Complexity-

Best Case -  $O(1)$

Average Case –  $O(\log n)$

Worst Case -  $O(\log n)$

**Recurrence Relation-**  $T(n) = T(n/2) + 1, n > 1$   $T(0) = 0, n = 0$

$$T(1) = 1, n = 1$$

## (c) Use divide and conquer method to recursively implement Linear

### Search

```
#include<iostream>
#include<time.h>
using namespace std;
clock_t begin, end;
double time_spent;

int linearSearch(int arr[],int num,int index,int size)
{
    if(index<=size)
    {
        if(arr[index] == num)
        {
            return index;
        }
        else
        {
            return linearSearch(arr, num, index+1, size);
        }
    }
}
return -1 ;
}
int main()
{
    int arr[50],size,f1,num;
    cout<<"Enter the size of an array(max size:50): ";
    cin>>size;
    cout<<"\nEnter the array: ";
    for(int i=0;i<size;i++)
    {
        cin>>arr[i];
    }
    cout<<"\nEnter the element to be searched: ";
    cin>>num;
    begin = clock();
```

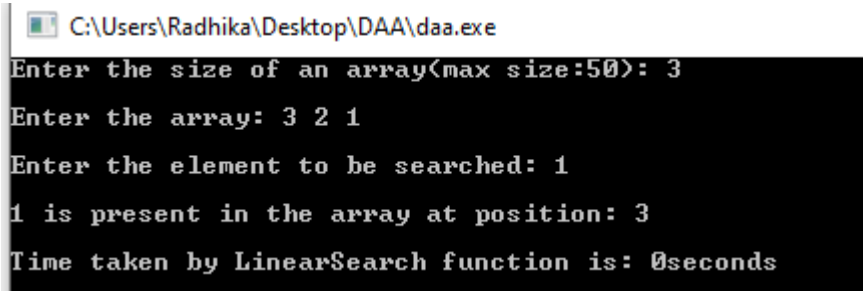


```

f1 = linearSearch(arr,num,0,size);
    if(f1 == -1)
        cout<<"\n"<<num<<" is not present in the array"; else
        cout<<"\n"<<num<<" is present in the array at position: "<<f1+1; end =
        clock();
        time_spent = (double)(end - begin) / CLOCKS_PER_SEC; cout<<"\n\nTime taken
        by LinearSearch function is: "<<time_spent<<"
        seconds";
        return 0;
    }

```

OUTPUT-



```

C:\Users\Radhika\Desktop\DAA\daa.exe
Enter the size of an array(max size:50): 3
Enter the array: 3 2 1
Enter the element to be searched: 1
1 is present in the array at position: 3
Time taken by LinearSearch function is: 0seconds

```

## Analysis

### Time Complexity-

Best Case -  $O(1)$

Average Case –  $O(n)$

Worst Case -  $O(n)$

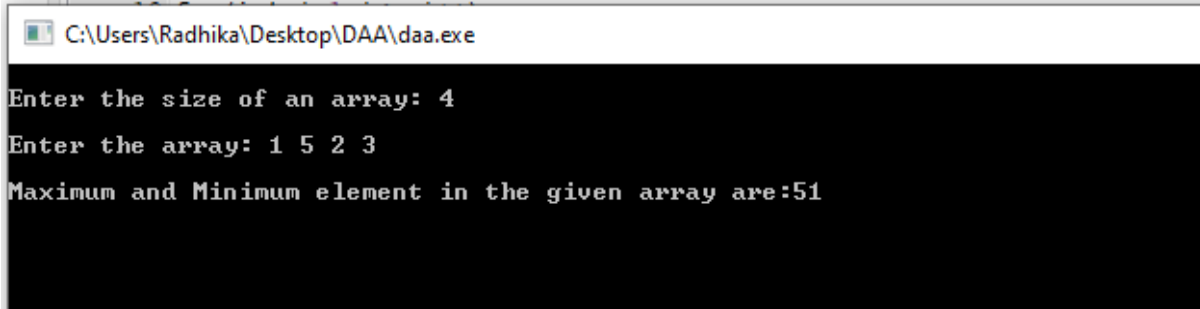
**Recurrence Relation-**  $T(n) = T(n-1) + 1, n > 1$   $T(0) = 0, n=0$   
 $T(1) = 1, n=1$

**(d) Use divide and conquer method to recursively implement and to find the maximum and minimum in a given list of n elements.**

**Without Divide and Conquer**

```
#include<iostream>
using namespace std;
int*maxmin(int a[],int n)
{
    int max,min;
    max=min=a[0];
    for(int i=1;i<n;i++)
    {
        if(max<a[i])
        {
            max=a[i];
        }
        else if(min>a[i])
        {
            min=a[i];
        }
    }
    int res[2]={ max,min };
    return res;
}
int main()
{
    int n,a[100],res;
    cout<<"\nEnter the size of an array: ";
    cin>>n;
    cout<<"\nEnter the array: ";
    for(int i=0;i<n;i++)
    cin>>a[i];
    int *r=maxmin(a,n);
    cout<<"\nMaximum and Minimum element in the given array are:
"<<*r<<" "<<*(r+1);
    return 0;
}
```

## OUTPUT



```
C:\Users\Radhika\Desktop\DAA\daa.exe

Enter the size of an array: 4
Enter the array: 1 5 2 3
Maximum and Minimum element in the given array are:51
```

## With Divide and Conquer

```
#include <iostream>
#include <climits>
using namespace std;
struct element
{
    int min;
    int max;
};

struct element minmax(int arr[], int low, int high)
{
    struct element temp, mml, mmr;
    int mid;
    if (low == high)
    {
        temp.max = arr[low];
        temp.min = arr[low];
        return temp;
    }
    if (high == low + 1)
    {
        if (arr[low] > arr[high])
        {
            temp.max = arr[low];
            temp.min = arr[high];
        }
    }
}
```

```

else
{
    temp.max = arr[high];
    temp.min = arr[low];
}
return temp;
}
mid = (low + high)/2;
mml = minmax(arr, low, mid);
mmr = minmax(arr, mid+1, high);

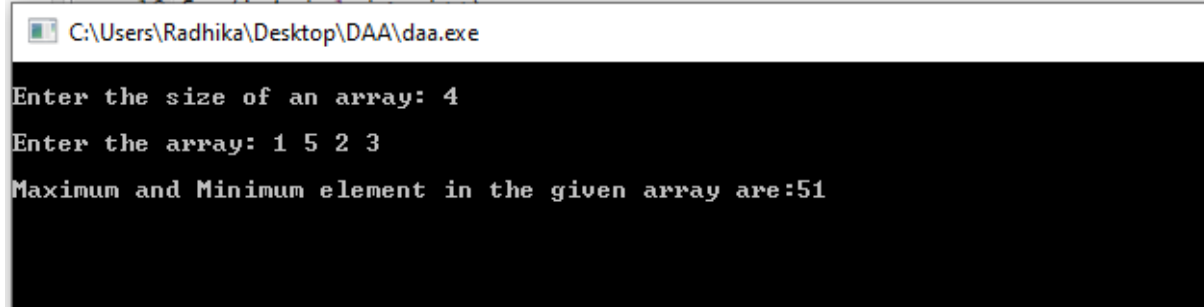
if (mml.min < mmr.min)
    temp.min = mml.min;
else
    temp.min = mmr.min;

if (mml.max > mmr.max)
    temp.max = mml.max;
else
    temp.max = mmr.max;
return temp;
}

int main()
{
    int n,a[100],res;
    cout<<"\nEnter the size of an array: ";
    cin>>n;
    cout<<"\nEnter the array: ";
    for(int i=0;i<n;i++)
    {
        cin>>a[i];
    }
    struct element temp = minmax(a, 0,n-1);
    cout<<"\nMaximum and Minimum element in the given array are:
"<<temp.max<<" "<<temp.min;
    return 0;
}

```

## OUTPUT



```
C:\Users\Radhika\Desktop\DAA\daa.exe  
Enter the size of an array: 4  
Enter the array: 1 5 2 3  
Maximum and Minimum element in the given array are:51
```

**2.Sort a given set of elements using Selection sort and hence find the time required to sort elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted.**

```
#include <iostream>
using namespace std;

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

void display(int arr[], int size)
{
    for (int i=0;i<size;i++)
        cout<<arr[i]<<" ";
}

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


int main()
{
    int n;
    cout<<"Enter the size of an array: ";
    cin>>n;
    int arr[n];
    cout<<"\nEnter the array: ";
    for(int i=0;i<n;i++)
    {
        cin>>arr[i];
    }
}
```

```

selectionSort(arr,n);
cout<<"\nArray after Selection Sort: ";
display(arr,n);
}

```

OUTPUT

 C:\Users\Radhika\Desktop\DAA\daa.exe

```

Enter the size of an array: 4
Enter the array: 5 3 1 2
Array after merge sort: 1 2 3 5

```

**3.Sort a given set of elements using the Merge sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted.**

```

#include<iostream>
using namespace std;

void merge(int arr[],int p, int q, int r)
{
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[n1], R[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[q + 1 + j];

    int i,j,k;
    i=0;
    j=0;
    k=p;

    while(i<n1&& j<n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;

```

```
    }  
    else  
    {  
        arr[k] = R[j];  
        j++;  
    }  
    k++;  
}  
  
while (i < n1) {  
    arr[k] = L[i];  
    i++;  
    k++;  
}
```



```


        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

void mergesort(int arr[],int p,int r)
{
    int q;
    if(p<r)
    {
        q=(p+r)/2    ;
        mergesort(arr,p,q);
        mergesort(arr,q+1,r);
        merge(arr,p,q,r);
    }
}

int main(){
    int n,arr[100];
    cout<<"\nEnter the size of an array: ";
    cin>>n;
    cout<<"\nEnter the array: ";
    for(int i=0;i<n;i++)
        cin>>arr[i];
    mergesort(arr,0,n-1);
    cout<<"\nArray after merge sort: ";
    for(int i=0;i<n;i++)
        cout<<" "<<arr[i];
    return 0;
}

```

OUTPUT:

 C:\Users\Radhika\Desktop\DAA\daa.exe

```

Enter the size of an array: 4
Enter the array: 5 3 1 2
Array after merge sort:  1 2 3 5

```

O

**4.Sort a given set of elements using the Quick sort method and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted**

```
#include <iostream>
using namespace std;

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

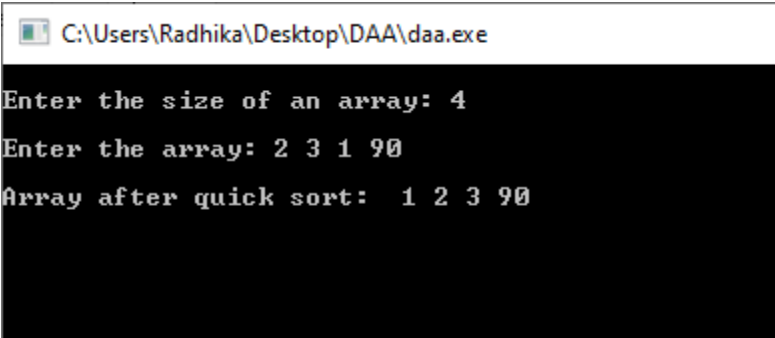
int partition(int arr[], int p, int r )
{
    int pivot = arr[r];
    int i = p - 1;

    for(int j = p; j<r; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

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

void quickSort(int arr[], int p, int r)
{
    if (p < r)
    {
        int q = partition(arr, p, r);
        quickSort(arr, p, q - 1);
        quickSort(arr, q + 1, r);
    }
}
```

```
int main()
{
    int n,arr[100];
    cout<<"\nEnter the size of an array: ";
    cin>>n;
    cout<<"\nEnter the array: ";
    for(int i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    quickSort(arr,0,n-1);
    cout<<"\nArray after quick sort: ";
    for(int i=0;i<n;i++)
    {
        cout<<" "<<arr[i];
    }
    return 0;
}
```



```
C:\Users\Radhika\Desktop\DAA\daa.exe
Enter the size of an array: 4
Enter the array: 2 3 1 90
Array after quick sort:  1 2 3 90
```

OUTPUT

## 5.Sort a given set of elements using the Heap sort method

```
#include<conio.h>

#include<iostream>

#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root

    int l = 2 * i + 1; // left = 2*i + 1

    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])

        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])

        largest = r;

    // If largest is not root
    if (largest != i) {
```

```

        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)

```

```
{  
  
    for (int i = 0; i < n; ++i)  
  
        cout << arr[i] << " ";  
  
    cout << "\n";  
  
}
```

// Driver code

```
int main()
```

```
{
```

```
    cout<<"Enter The Size Of Array:  ";
```

```
    int n;
```

```
    cin>>n;
```

```
    int arr[n], key,i;
```

// Taking Input In Array

```
    for(int j=0;j<n;j++){
```

```
        cout<<"Enter "<<j<<" Element : ";
```

```
        cin>>arr[j];
```

```
    }
```

//Your Entered Array Is

```
    for(int a=0;a<n;a++){
```

```
        cout<<"arr[ "<<a<<" ] = ";
```

```
        cout<<arr[a]<<endl;
```

```
    }
```

```
    heapSort(arr, n);

    cout << "Sorted array is \n";

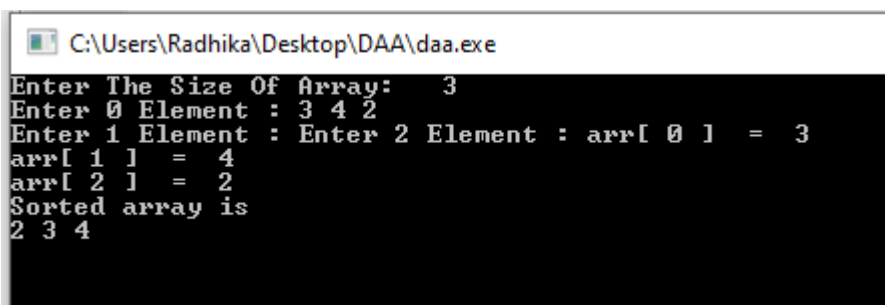
    printArray(arr, n);

    getch();

    return 0;

}
```

OUTPUT:



```
C:\Users\Radhika\Desktop\DAA\daa.exe
Enter The Size Of Array: 3
Enter 0 Element : 3 4 2
Enter 1 Element : Enter 2 Element : arr[ 0 ] = 3
arr[ 1 ] = 4
arr[ 2 ] = 2
Sorted array is
2 3 4
```


## 6.Sort a given set of elements using Insertion sort method.

```
#include<iostream>
using namespace std;
void display(int arr[],int size)
{
    for(int i=0;i<size;i++)
    {
        cout<<arr[i]<<" ";
    }
}
void insertion_Sort(int arr[],int size)
{
    for(int i=1;i<size;i++)
    {
        int key = arr[i];
        int j = i-1;
        while(key<arr[j] && j>=0)
        {
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=key;
    }
}

int main()
{
    int n;
    cout<<"\nEnter the size of an array: ";
    cin>>n;
    int arr[n];
    cout<<"\nEnter the array: ";
    for(int i=0;i<n;i++)
    {
        cin>>arr[i];
    }
    insertion_Sort(arr,n);
    cout<<"\nArray after Insertion Sort: ";
    display(arr,n);
    return 0;
}
```

OUTPUT:



 C:\Users\Radhika\Desktop\DAA\daa.exe

Enter the size of an array: 5

Enter the array: 2 45 23 12 132

Array after Insertion Sort: 2 12 23 45 132

## 7.Sort a given set of elements using Selection sort method

```
#include <iostream>
using namespace std;

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


void display(int arr[], int size)
{
    for (int i=0;i<size;i++)
        cout<<arr[i]<<" ";
}

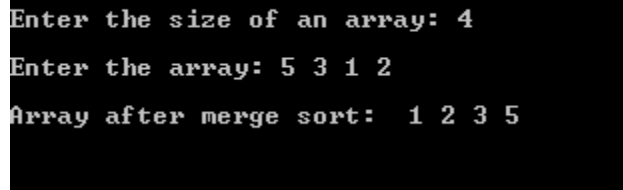
void selectionSort(int arr[], int size)
{
    for (int i = 0; i<size-1;i++)
    {
        int min_idx = i;
        for (int j=i+1; j<size; j++)
        {
            if (arr[j]<arr[min_idx])
                min_idx = j;
        }
        swap(&arr[min_idx], &arr[i]);
    }
}

int main()
{
    int n;
    cout<<"Enter the size of an array: ";
    cin>>n;
    int arr[n];
    cout<<"\nEnter the array: ";
    for(int i=0;i<n;i++)
    {
        cin>>arr[i];
    }
}
```

```
selectionSort(arr,n);  
cout<<"\nArray after Selection Sort: ";  
display(arr,n);  
}
```

## OUTPUT

 C:\Users\Radhika\Desktop\DAA\daa.exe



```
Enter the size of an array: 4  
Enter the array: 5 3 1 2  
Array after merge sort: 1 2 3 5
```

**8.Print all the nodes reachable from a given starting node in a given digraph using Breadth First Search method.**

```
#include <iostream>
```

```
#include <list>
```

```
#include <vector>
```

```
using std::cout;
```

```
using std::endl;
```

```
using std::list;
```

```
using std::vector;
```

```
class Graph {
```

```
private:
```

```
// Number of vertices
```

```
int V;  
  
// Pointer to adjacency list  
vector<list<int>> adj;  
  
public:  
  
// Constructor prototype  
Graph(int v);  
  
// Method to add an edge  
void addEdge(int v, int w);  
  
// Method for BFS traversal give a source "s"  
void BFS(int s);  
};  
  
// Constructer with number of vertices  
Graph::Graph(int v) {  
    // Set number of verticest  
    V = v;  
  
    // Resize the number of adjacency lists  
    adj.resize(v);  
}
```

```
// Implementation of method to add edges

void Graph::addEdge(int v, int w) { adj[v].push_back(w); }


// Perform BFS given a starting vertex

void Graph::BFS(int s) {

    // Start with all vertices as not visited

    // Value initialized to false

    vector<bool> visited(V);


    // Create a queue for BFS

    list<int> queue;


    // Starting vertex marked as visited and added to queue

    visited[s] = true;

    queue.push_back(s);


    // Continue until queue is empty

    while (!queue.empty()) {

        // Get the front of the queue and remove it

        s = queue.front();

        queue.pop_front();

    }

}
```

```

// Get all adjacent vertices from that vertex

cout << "Checking adjacent vertices for vertex " << s << endl;

for (auto i : adj[s]) {

    // We only care about nodes not visited yet
    if (!visited[i]) {

        // Mark as visited

        cout << "Visit and enqueue " << i << endl;

        visited[i] = true;

        // Push back to check this vertex's vertices

        queue.push_back(i);

    }

}

}

}

}

int main() {

    // Create a new graph

    Graph g(6);

    // Create some edges to the vertices

    // Connections for vertex 0

    g.addEdge(0, 1);

    g.addEdge(0, 2);

```

```
// Connections for vertex 1
```

```
g.addEdge(1, 0);
```

```
g.addEdge(1, 3);
```

```
g.addEdge(1, 4);
```

```
// Connections for vertex 2
```

```
g.addEdge(2, 0);
```

```
g.addEdge(2, 4);
```

```
// Connections for vertex 3
```

```
g.addEdge(3, 1);
```

```
g.addEdge(3, 4);
```

```
g.addEdge(3, 5);
```

```
// Connections for vertex 4
```

```
g.addEdge(4, 1);
```

```
g.addEdge(4, 2);
```

```
g.addEdge(4, 3);
```

```
g.addEdge(4, 5);
```

```
// Connections for vertex 5
```

```
g.addEdge(5, 3);
```

```
g.addEdge(5, 4);

// Perform BFS and print result

g.BFS(2);

return 0;

}
```

OUTPUT:

```
Checking adjacent vertices for vertex 2
Visit and enqueue 0
Visit and enqueue 4
Checking adjacent vertices for vertex 0
Visit and enqueue 1
Checking adjacent vertices for vertex 4
Visit and enqueue 3
Visit and enqueue 5
Checking adjacent vertices for vertex 1
Checking adjacent vertices for vertex 3
Checking adjacent vertices for vertex 5

...Program finished with exit code 0
Press ENTER to exit console.□
```



**9.Print all the nodes reachable from a given starting node in a given digraph using Depth First Search method.**

```
#include <iostream>

#include <list>

using namespace std;

class Graph{
private:
    // Number of vertices
    int V;

    // Pointer to adjacency list
    list<int> *adj;

    // DFS recursive helper functions
    void DFS_helper(int s, bool *visited);

public:
    // Constructor prototype
    Graph(int v);

    // Method to add an edge
    void addEdge(int v, int w);
```

```
// Method for BFS traversal give a source "s"

void DFS(int s);

};


// Constructer with number of vertices

Graph::Graph(int v){

    // Set number of vertices

    V = v;


    // Create new adjacency list

    adj = new list<int>[v];

}


// Implementation of method to add edges

void Graph::addEdge(int v, int w){

    adj[v].push_back(w);

}


void Graph::DFS_helper(int s, bool *visited){

    // Mark the current node as visited

    cout << "Visiting node " << s << endl;

    visited[s] = true;
```

```

// Go through the adjacency list

for(auto i = adj[s].begin(); i != adj[s].end(); i++){

    // If not visited, travel through that vertex

    if(!visited[*i]){

        cout << "Going to vertex " << *i << " from vertex " << s << endl;

        DFS_helper(*i, visited);

    }

}

}

// Perform BFS given a starting vertex

void Graph::DFS(int s){

    // Start with all vertices as not visited

    bool *visited = new bool[V];

    for(int i = 0; i < V; i++){

        visited[i] = false;

    }

    // Beginning of recursive call

    DFS_helper(s, visited);

}

int main(){

```

```
// Create a new graph
Graph g(6);

// Create some edges to the vertices

// Connections for vertex 0
g.addEdge(0, 1);
g.addEdge(0, 2);

// Connections for vertex 1
g.addEdge(1, 0);
g.addEdge(1, 3);
g.addEdge(1, 4);

// Connections for vertex 2
g.addEdge(2, 0);
g.addEdge(2, 4);

// Connections for vertex 3
g.addEdge(3, 1);
g.addEdge(3, 4);
g.addEdge(3, 5);

// Connections for vertex 4
```

```
g.addEdge(4, 1);

g.addEdge(4, 2);

g.addEdge(4, 3);

g.addEdge(4, 5);


// Connections for vertex 5

g.addEdge(5, 3);

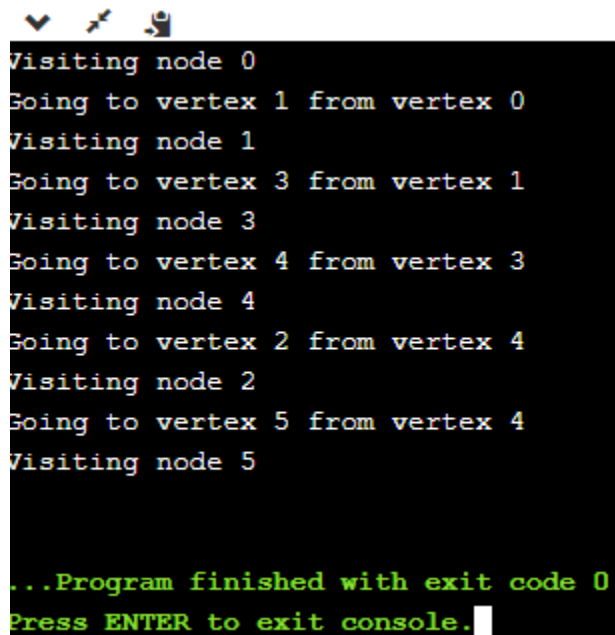
g.addEdge(5, 4);


// Perform DFS and print result

g.DFS(0);


return 0;

}
```

A terminal window with a black background and green text. It shows the output of a Depth-First Search (DFS) algorithm. The output starts with 'Visiting node 0' and then shows the sequence of nodes visited: 0, 1, 3, 4, 2, 5. Each node visit is preceded by a message indicating the path taken, such as 'Going to vertex 1 from vertex 0'. The terminal ends with a message indicating the program finished with exit code 0 and a prompt to press ENTER to exit the console.

```
Visiting node 0
Going to vertex 1 from vertex 0
Visiting node 1
Going to vertex 3 from vertex 1
Visiting node 3
Going to vertex 4 from vertex 3
Visiting node 4
Going to vertex 2 from vertex 4
Visiting node 2
Going to vertex 5 from vertex 4
Visiting node 5

...Program finished with exit code 0
Press ENTER to exit console.
```

**10. From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.**

```
#include
<iostream>

#include <list>
#include <utility>
#include <set>
#include <vector>
#define INF 1000000
using namespace std;
class Graph{
private:
    // Number of vertices
    int V;
    // Pointer to adjacency list
    list<pair<int, int>> *adj;
public:
    // Constructor prototype
    Graph(int v);
    // Method to add an edge/weight pair
    void addEdge(int v1, int v2, int weight);
    // Method for shortest path
    void shortestPath(int s);
};
// Constructor with number of vertices
Graph::Graph(int v){
    // Set number of vertices
    V = v;
    // Create new adjacency list
```

```

        adj = new list<pair<int, int>>[v];
    }
    // Implementation of method to add edges
    void Graph::addEdge(int v1, int v2, int weight){
        adj[v1].push_back(make_pair(v2, weight));
    }
    // Implemenation of method to find the shortest paths
    void Graph::shortestPath(int s){
        // Create set to store vertices
        // Use this to extract the shortest path
        set<pair<int, int>> extract_set;
        // Vector for distances
        // All paths are initialized to a large value
        vector<int> distances(V, INF);
        // Insert the entry point into the set
        // Initialize distance to 0
        extract_set.insert(make_pair(0, s));
        distances[s] = 0;
        // Continue until all shortest distances are finalized
        while(!extract_set.empty()){
            // Extract the minimum distances
            pair<int, int> tmp = *(extract_set.begin());
            extract_set.erase(extract_set.begin());
            // Get the vertex number
            int u = tmp.second;
            // Go over the adjacency list
            for(auto i = adj[u].begin(); i != adj[u].end(); i++){
                // Get the vertex and weight
                int v = (*i).first;
                int weight = (*i).second;
                // Check if we have found a shorter path to v
                if(distances[v] > distances[u] + weight){
                    // Remove the current distance if it is in the
                    set

```

```

        if(distances[v] != INF){
            extract_set.erase(extract_set.find(make_pair(distances[v], v)));
        }

        // Update the distance
        distances[v] = distances[u] + weight;
        extract_set.insert(make_pair(distances[v], v));
    }
}

cout << "Minimum distances from vertex: " << s << endl;
for(int i = 0; i < V; i++){
    cout << "Vertex: " << i << "\tDistance: " <<
distances[i] << endl;
}
}

int main(){
    // Create a graph
    Graph g(9);
    // Gives some edges and weights to the vertices
    // Add node 0
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);

    // Add node 1
    g.addEdge(1, 0, 4);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);

    // Add node 2
    g.addEdge(2, 1, 8);
    g.addEdge(2, 8, 2);
    g.addEdge(2, 5, 4);

```



```
g.addEdge(2, 3, 7);
// Add node 3
g.addEdge(3, 2, 7);
g.addEdge(3, 5, 14);
g.addEdge(3, 4, 9);
// Add node 4
g.addEdge(4, 3, 9);
g.addEdge(4, 5, 10);

// Add node 5
g.addEdge(5, 6, 2);
g.addEdge(5, 3, 14);
g.addEdge(5, 4, 10);

// Add node 6
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(6, 5, 2);
// Add node 7
g.addEdge(7, 0, 8);
g.addEdge(7, 1, 11);
g.addEdge(7, 8, 7);
g.addEdge(7, 6, 1);
// Add node 8
g.addEdge(8, 2, 2);
g.addEdge(8, 7, 7);
g.addEdge(8, 6, 6);
g.shortestPath(0);
return 0;
```

```
}
```

OUTPUT:

```
Minimum distances from vertex: 0
Vertex: 0      Distance: 0
Vertex: 1      Distance: 4
Vertex: 2      Distance: 12
Vertex: 3      Distance: 19
Vertex: 4      Distance: 21
Vertex: 5      Distance: 11
Vertex: 6      Distance: 9
Vertex: 7      Distance: 8
Vertex: 8      Distance: 14

...Program finished with exit code 0
Press ENTER to exit console.
```

## **11.Find Minimum Cost Spanning Tree of a given undirected graph using Prim's algorithm.**

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <bits/stdc++.h>

using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
```

```

        return min_index;
    }

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<" \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

```

```
// To represent set of vertices included in MST

bool mstSet[V];

// Initialize all keys as INFINITE

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

    key[i] = INT_MAX, mstSet[i] = false;

// Always include first 1st vertex in MST.

// Make key 0 so that this vertex is picked as first vertex.

key[0] = 0;

parent[0] = -1; // First node is always root of MST

// The MST will have V vertices

for (int count = 0; count < V - 1; count++)

{

    // Pick the minimum key vertex from the

    // set of vertices not yet included in MST

    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set

    mstSet[u] = true;
```

```

        // Update key value and parent index of
        // the adjacent vertices of the picked vertex.

        // Consider only those vertices which are not
        // yet included in MST
        for (int v = 0; v < V; v++)

            // graph[u][v] is non zero only for adjacent vertices of m
            // mstSet[v] is false for vertices not yet included in MST
            // Update the key only if graph[u][v] is smaller than key[v]
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    // print the constructed MST
    printMST(parent, graph);
}

// Driver code
int main()
{
    /* Let us create the following graph
        2 3
        (0)--(1)--(2)
    */
}

```

|/\|

6|8/\5|7

|/\|

(3)----- (4)

9       \*/

```
int graph[V][V] = { { 0, 2, 0, 6, 0 },
```

```
                  { 2, 0, 3, 8, 5 },
```

```
                  { 0, 3, 0, 0, 7 },
```

```
                  { 6, 8, 0, 0, 9 },
```

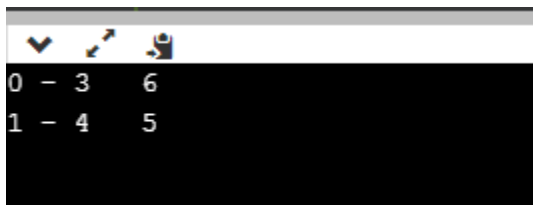
```
                  { 0, 5, 7, 9, 0 } };
```

```
// Print the solution
```

```
primMST(graph);
```

```
return 0;
```

```
}
```



```
0 - 3 6
1 - 4 5
```

## **12.Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm**

```
#include <algorithm>

#include <iostream>

#include <vector>

using namespace std;

#define edge pair<int, int>

class Graph {

    private:

        vector<pair<int, edge> > G; // graph

        vector<pair<int, edge> > T; // mst

        int *parent;

        int V; // number of vertices/nodes in graph

    public:

        Graph(int V);

        void AddWeightedEdge(int u, int v, int w);

        int find_set(int i);

        void union_set(int u, int v);

        void kruskal();

        void print();

};
```



```

Graph::Graph(int V) {
    parent = new int[V];

    //i 0 1 2 3 4 5
    //parent[i] 0 1 2 3 4 5
    for (int i = 0; i < V; i++)
        parent[i] = i;

    G.clear();
    T.clear();
}

void Graph::AddWeightedEdge(int u, int v, int w) {
    G.push_back(make_pair(w, edge(u, v)));
}

int Graph::find_set(int i) {
    // If i is the parent of itself
    if (i == parent[i])
        return i;
    else
        // Else if i is not the parent of itself
        // Then i is not the representative of his set,
        // so we recursively call Find on its parent
        return find_set(parent[i]);
}

```

```
}
```

```
void Graph::union_set(int u, int v) {
```

```
    parent[u] = parent[v];
```

```
}
```

```
void Graph::kruskal() {
```

```
    int i, uRep, vRep;
```

```
    sort(G.begin(), G.end()); // increasing weight
```

```
    for (i = 0; i < G.size(); i++) {
```

```
        uRep = find_set(G[i].second.first);
```

```
        vRep = find_set(G[i].second.second);
```

```
        if (uRep != vRep) {
```

```
            T.push_back(G[i]); // add to tree
```

```
            union_set(uRep, vRep);
```

```
        }
```

```
    }
```

```
}
```

```
void Graph::print() {
```

```
    cout << "Edge :"
```

```
        << " Weight" << endl;
```

```
    for (int i = 0; i < T.size(); i++) {
```

```
        cout << T[i].second.first << " - " << T[i].second.second << " : "
```

```
        << T[i].first;
```

```
        cout << endl;
    }
}

int main() {
    Graph g(6);

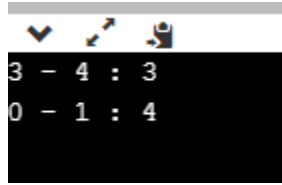
    g.AddWeightedEdge(0, 1, 4);
    g.AddWeightedEdge(0, 2, 4);
    g.AddWeightedEdge(1, 2, 2);
    g.AddWeightedEdge(1, 0, 4);
    g.AddWeightedEdge(2, 0, 4);
    g.AddWeightedEdge(2, 1, 2);
    g.AddWeightedEdge(2, 3, 3);
    g.AddWeightedEdge(2, 5, 2);
    g.AddWeightedEdge(2, 4, 4);
    g.AddWeightedEdge(3, 2, 3);
    g.AddWeightedEdge(3, 4, 3);
    g.AddWeightedEdge(4, 2, 4);
    g.AddWeightedEdge(4, 3, 3);
    g.AddWeightedEdge(5, 2, 2);
    g.AddWeightedEdge(5, 4, 3);

    g.kruskal();

    g.print();

    return 0;
}
```

}



### 13.Implement 0/1 Knapsack problem using dynamic programming.

```
#include<conio.h>

#include <iostream>

using namespace std;

int max(int x, int y) {

    return (x > y) ? x : y;

}

int knapSack(int W, int w[], int v[], int n) {

    int i, wt;

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

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

        for (wt = 0; wt <= W; wt++) {

            if (i == 0 || wt == 0)

                K[i][wt] = 0;

            else if (w[i - 1] <= wt)

                K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i - 1][wt]);

            else

                K[i][wt] = K[i - 1][wt];

        }

    }

    return K[n][W];

}

int main() {
```

```
cout << "Enter the number of items in a Knapsack:";

int n, W;

cin >> n;

int v[n], w[n];

for (int i = 0; i < n; i++) {

    cout << "Enter value and weight for item " << i << ":";

    cin >> v[i];

    cin >> w[i];

}

cout << "Enter the capacity of knapsack";

cin >> W;

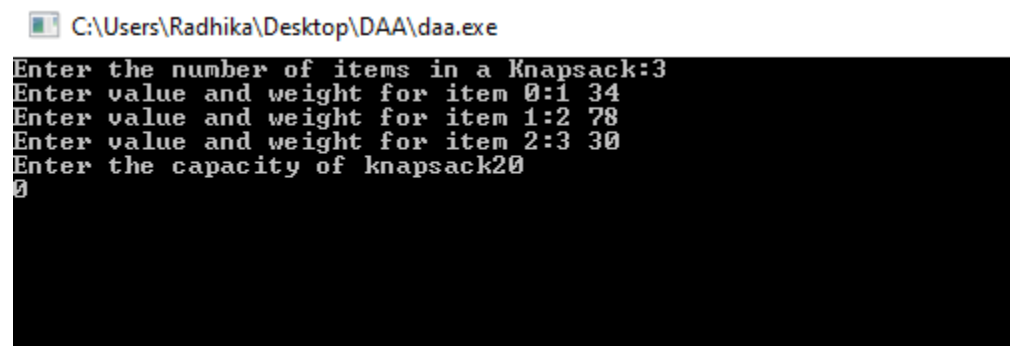
cout << knapSack(W, w, v, n);

getch();

return 0;

}
```

## OUTPUT



```
C:\Users\Radhika\Desktop\DAA\daa.exe
Enter the number of items in a Knapsack:3
Enter value and weight for item 0:1 34
Enter value and weight for item 1:2 78
Enter value and weight for item 2:3 30
Enter the capacity of knapsack20
0
```