FINAL LAB REPORT

Title: Algorithm I Lab

Course code: CSE-210

2nd Year 1st Semester

Submitted to

Mohammad Ashraful Islam	Bulbul Ahammad
Assistant Professor Department of Computer Science and Engineering	Assistant Professor Department of Computer Science and Engineering

NAME	EXAM ROLL	CLASS ROLL	REGISTRATION NUMBER
Md Tanvir Hossain Saon	**202200**	**388**	**20200650758**

TABLE OF CONTENT

Algorithm Type	Topic Name
Searching Algorithms	Linear Search Binary Search
Sorting Algorithms	Insertion Sort Bubble Sort Selection Sort Merge Sort Quick Sort Heap Sort Radix Sort Counting Sort
Greedy Algorithms	Activity Selection Problem Optimal Caching Minimizing Maximum Lateness Fractional Knapsack Huffman Coding
Dynamic Programming	LCS LIS 0/1 Knapsack
Graph Algorithms	BFS DFS Topological Sort Dijkstra Bellman-Ford Floyd Warshall Prims Kruskal

Searching Algorithm

Linear Search

Linear Search

Use of the Algorithm : To Find an element from a list. The list can be sorted or unsorted.

Input of the Algorithm : An Array and a target value.

Output of the Algorithm : Yes, if target value is found, NO, if target value is not found.

Complexity : O(N).

(I) Best Case :- O(1), Array = {51, 5, 91, 4, 103}, Target = 51
output :- 1

(II) Worst case :- O(N), Array = {5, 3, 9, 4, 03}, Target = 0
output 5.

(III) Avanage case :- Array = {51, 3, 91, 4, 03} Target = 91,
output = 3. O(N)

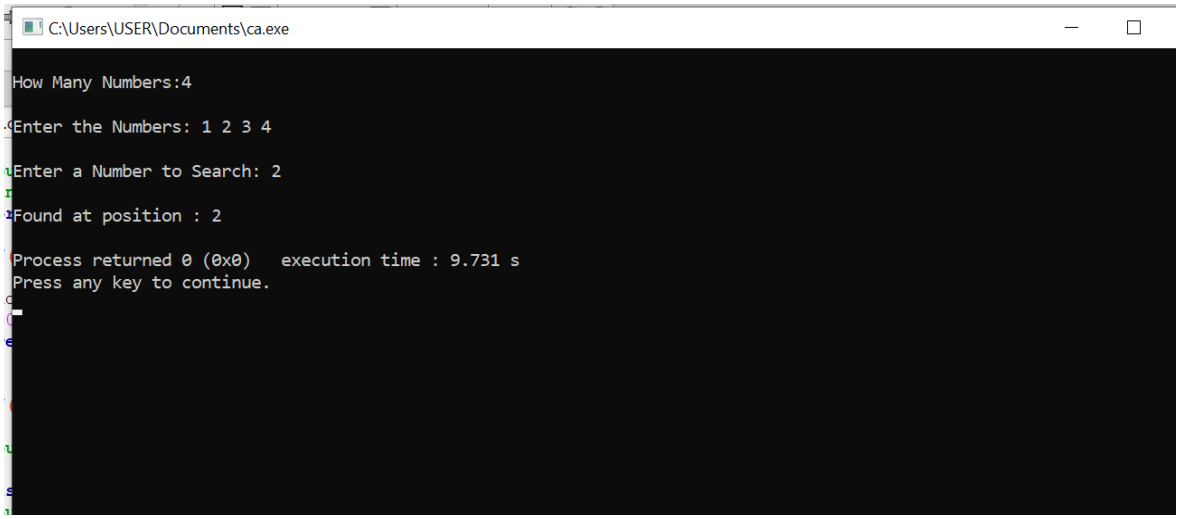
Basic Workflow :- In this search, a sequential search is performed. Each item is checked. If a match is found the particular item is returned. Otherwise the search continues untill end.

Source Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
int arr[1001],i,f,n,num,index;
cout<<"\nHow Many Numbers:";
cin>>n;
cout<<"\nEnter the Numbers: ";
for(i=0; i<n; i++)
{
cin>>arr[i];
}
cout<<"\nEnter a Number to Search:
";
cin>>num;
for(i=0; i<n; i++)
{
if(arr[i]==num)
{
index = i+1;
f=0;
break;
}
}
```

```
if(f==0)
{
cout<<"\nFound at position :
"<<index;
}
else{cout<<"\nNot Found!!!";}
cout<<endl;
}
```

Output:



Binary Search

Binary Search

Use of the Algorithm :- To Find an element from a list. The list should be sorted.

Input of the Algorithm :- An array and a target value.

Output of the Algorithm :- Yes, if target is found.
No, if target is not found.

Complexity :- $O(\log_2 N)$.

Example :-

(I) Best case :- A = {51, 5, 14, 93}, T = 1, output = Yes.

(II) Worst case :- A = {1, 32, 4, 8, 93}, T = 1, output = Yes.

(III) Avarage Case :- A = {1, 4, 8, 12, 9, 73}, T = 4, output = Yes.

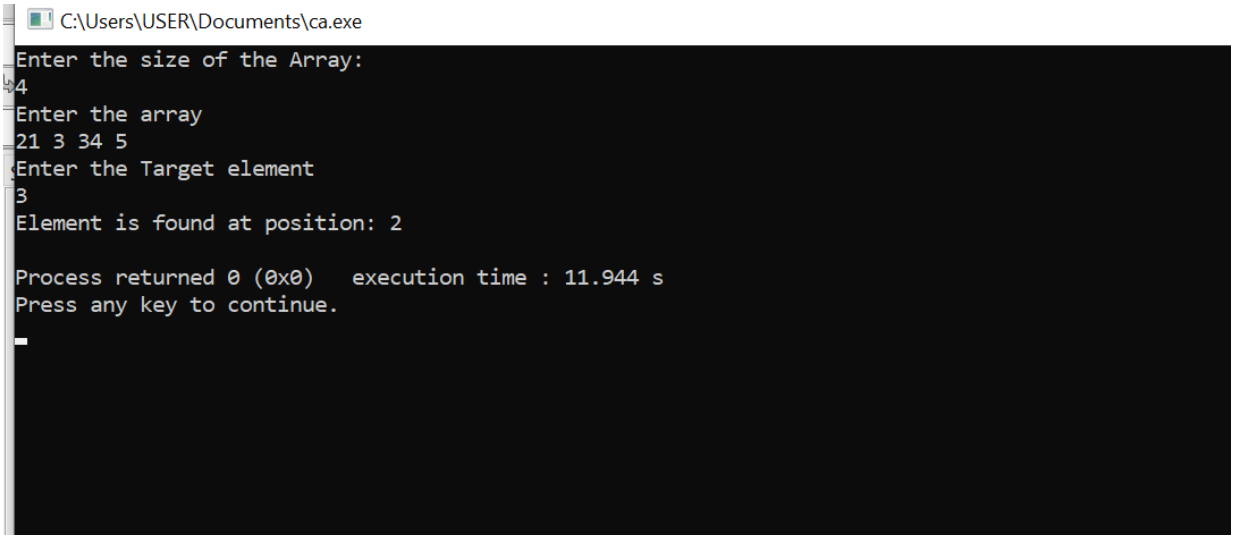
Basic Workflow of the Algorithm :- The Search process begins by identifying the mid element of data of the sorted array. After that the value is compared to the element, then comparison and matching searches analyze the value above the mid element.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int array[], int x,
int low, int high)
 {
  while (low <= high)
    {
    int mid = low + (high - low) /
2;
    if (array[mid] == x)
      return mid;
    if (array[mid] < x)
      low = mid + 1;
    else
      high = mid - 1;
    }
  return -1;
}
```

```
int main() {
  int array[1001],x,n;
  cout<<"Enter the size of the
Array:"<<endl;
  cin>>n;
  cout<<"Enter the array"<<endl;
  for(int i=0;i<n;i++)
  {
      cin>>array[i];
  }
  cout<<"Enter the Target
element"<<endl;
  cin>>x;
  int result = binarySearch(array,
x, 0, n-1);
  if (result == -1)
    cout<<"Not found"<<endl;
  else
    cout<<"Element is found at
position: "<<result+1<<endl;
return 0;
}
```

Output:



Sorting Algorithm

Insertion Sort

Insertion Sort

Use of the Algorithm :- To sort the element from a list of array.

Input of the Algorithm :- An Array of n elements.

output of the Algorithm :- A sorted array.

Complexity of the Algorithm :- $O(N^2)$.

Example :-

(I) Best case :- $A = \{1,2,3,4\}$, $O(N)$.

(II) Worst case :- $A = \{4,3,2,1\}$, $O(N^2)$.

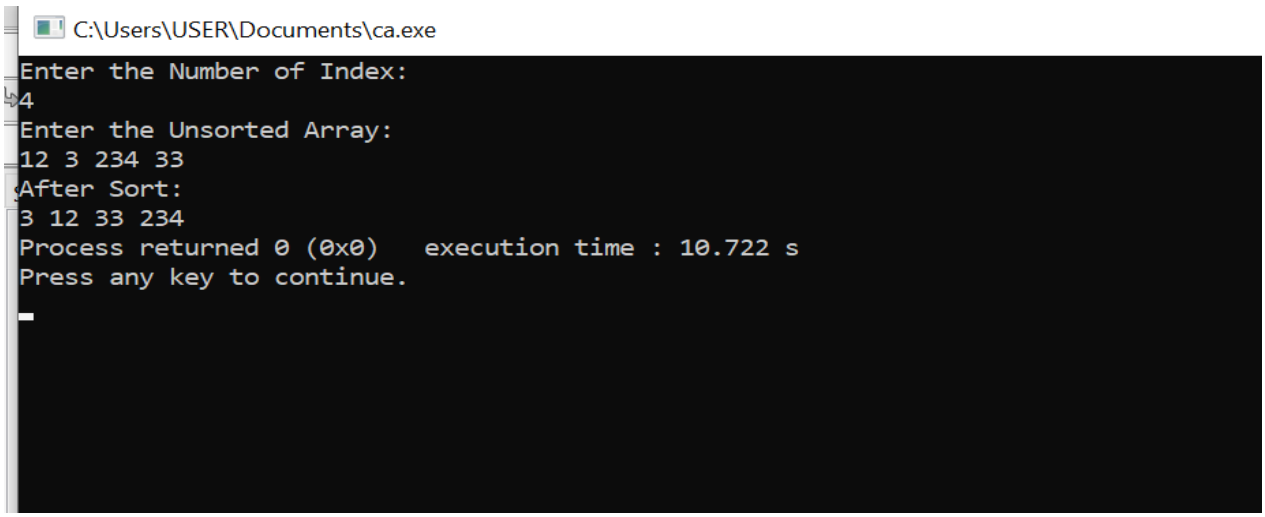
(III) Avarage case :- $A = \{2,1,3,4\}$, $O(N^2)$.

Basic workflow of the Algorithm :- Insertion sort is a simple sorting algorithm that works similar to how we play a hand of cards. Arrays are partially divided into a sorted and a unsorted array. The value is piked from the array and placed in the correct position in the array.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
      int ar[1001],n,i,j,k;
      cout<<"Enter the Number of
Index:"<<endl;
      cin>>n;
      cout<<"Enter the Unsorted
Array:"<<endl;
    for(i=0;i<n;i++)
    {
        cin>>ar[i];
    }
      for (i=1;i<n;i++)
      {
            k=ar[i];
            j = i - 1;
            while (j >= 0 && ar[j]>k)
            {
                  ar[j + 1] = ar[j];
                  j = j - 1;
            }
            ar[j + 1]=k;
      }
      cout<<"After Sort:"<<endl;
      for (int i = 0; i < n; i++)
   {
       cout << ar[i] << " ";
   }
      return 0;
}
```

Output:



Bubble Sort

Bubble Sort

Use of the Algorithm :- To sort element from a list of Array.

Input of the Algorithm :- An array of n elements.

Output of the Algorithm :- A sorted Array.

Complexity :- $O(N^2)$

Example :-

Best Case :- $A = \{1, 2, 3, 4\}$, $O(N)$

Avarage Case :- $A = \{4, 3, 3, 2, 1\}$, $O(N^2)$

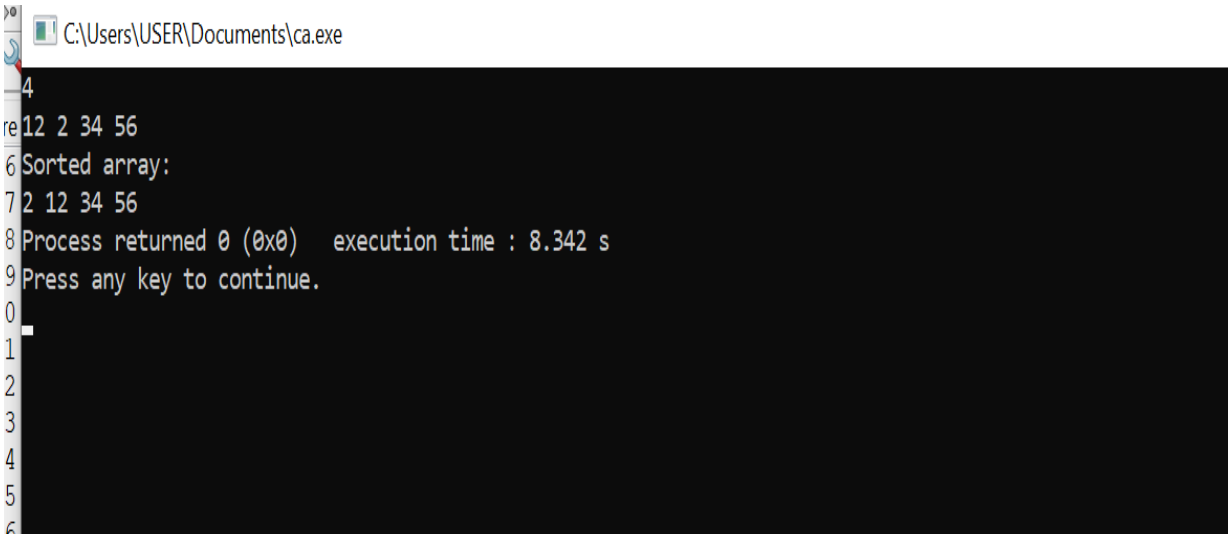
Worst case :- $A = \{1, 3, 2, 1\}$, $O(N^2)$

Basic Workflow of the Algorithm :- It works by repeatedly swapping neighboring elements if they are in the wrong order. This algorithm is not suitable for large data sets because it's worst and avarage case time complexity is High.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
      int a[1001],i,j,n;
      cin>>n;
      for (i=0;i<n;i++)
      {
            cin>>a[i];
      }
      for (i = 0; i < n - 1; i++){
        for (j = 0; j < n - i - 1;
j++){
            if (a[j] > a[j + 1]){
                swap(a[j],a[j + 1]);
                  }
        }
      }
      cout << "Sorted array: "<<endl;
      for (i=0;i<n;i++)
      {
            cout<<a[i]<<" ";
      }
      return 0;
}
```

Output:



Selection Sort

Selection sort

Use of the Algorithm :- To sort the element from list of array.

Input of the Algorithm :- An array of n elements

Output of the Algorithm :- A sorted Array.

Complexity :- $O(N^2)$.

Example :-

(I) Best Case :- $O(N^2)$, $A = \{1,2,3,4\}$

(II) Worst case :- $O(N^2)$, $A = \{4,3,3,1\}$

(III) Avarage Case :- $O(N^2)$ $A = \{2,1,3,3,4\}$

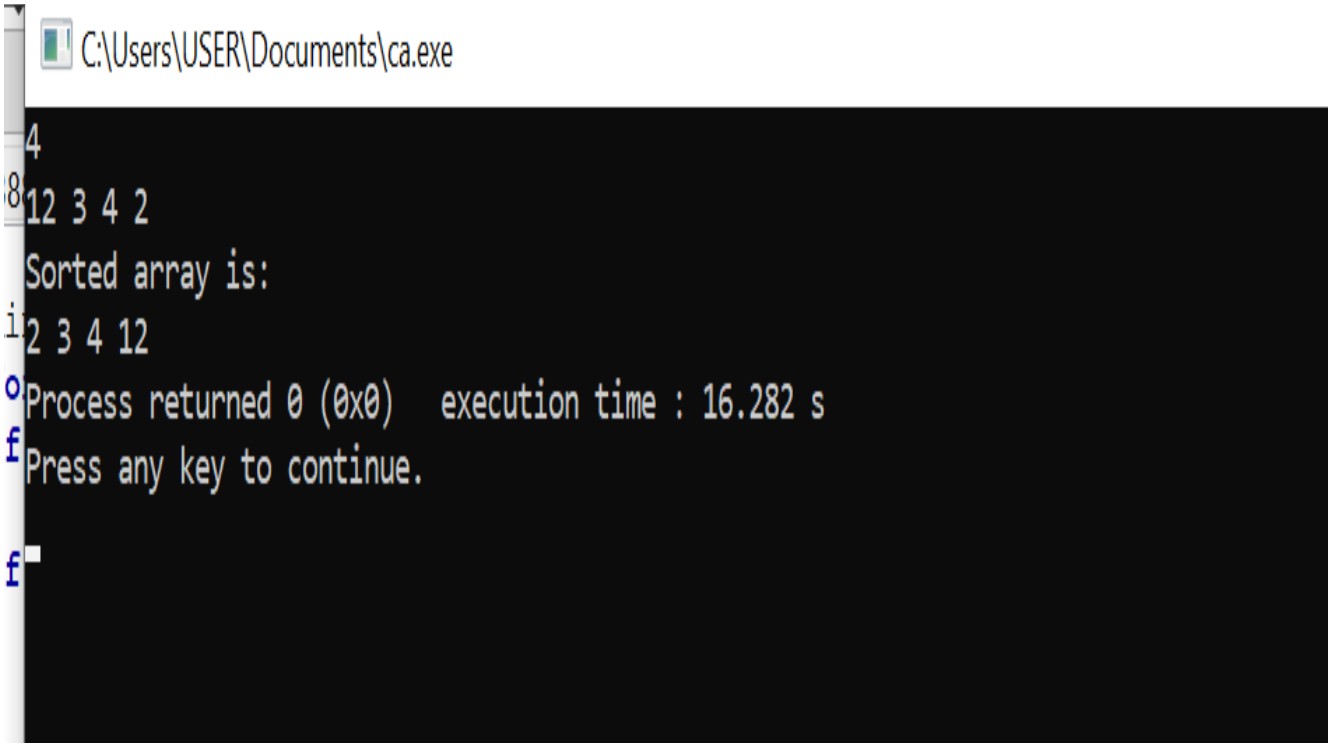
Basic Work flow of the Algorithm :- It is an in-place comparison-based algorithm that divide the list into two parts, a left shorted part and a right unsort part. Initially, the sorted selection is empty, and unsorted contains full list. It is use to sort a short list.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;
void s_sort(int a[],int n)
{
      int i, j,min_idx;
      for(i=0;i<n-1;i++)
      {
            min_idx = i;
            for (j = i+1; j < n; j++)
            if (a[j] < a[min_idx])
                  min_idx = j;
            if(min_idx!=i)

swap(a[min_idx],a[i]);
      }
}
```

```
int main()
{
      int a[1001],n,i;
      cin>>n;
      for(i=0;i<n;i++)
    {
        cin>>a[i];
    }
      s_sort(a,n);
      cout << "Sorted array is: \n";
      for (i=0;i<n;i++)
    {
        cout <<a[i]<< " ";
    }
      return 0;
}
```

Output:



Merge Sort

Merge sort

Use of the Algorithm :- To sort the elements from a list of Array.

Input of the algorithm :- An Array of n elements.

Output of the Algorithm :- A sorted Array.

complexity :- $O(N \log N)$

Example :-

(I) Best Case :- $O(N \log N)$, $A = \{1, 4, 8, 10\}$

(II) Worst case :- $O(N \log N)$, $A = \{4, 3, 2, 1\}$

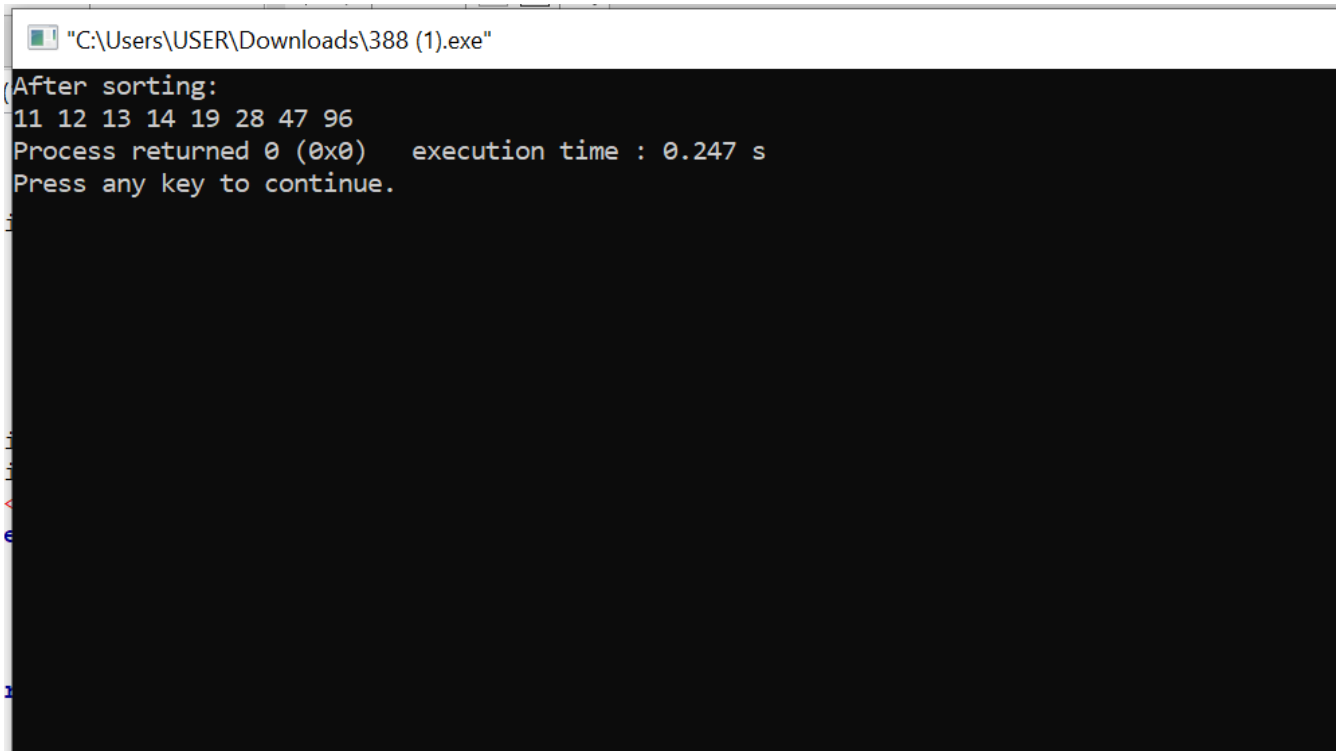
(III) Avarage case :- $O(N \log N)$, $A = \{2, 1, 1, 3, 3\}$

Basic Workflow of the Algorithm :- We divide a array in sub array untill reach a stage where we try to perform Mergeform on a sub array. After that the merge funtion comes into play and combine the sorted arrays into larger arrays untill the whole array is merge.

Source Code:

```
#include<bits/stdc++.h>
using namespace std;
int ar[10] = {11, 19, 13, 96, 14,
12, 28, 47};
vector<int>v;
void Partition(int l, int r)
{
        if(l==r)return;

        int mid = (l+r)/2;

        Partition(l,mid);
        Partition(mid+1,r);

        v.clear();

        int i=l, j=mid+1;

        while(i<=mid || j<=r)
        {
                if(i<=mid &&
j<=r)
                {

if(ar[i]<ar[j])
                        {

v.push_back(ar[i]);

i++;
                        }
                        else
                        {

v.push_back(ar[j]);

j++;
                        }
                }

                else if(i<=mid)
                {

v.push_back(ar[i]);
                        i++;
                }

                else
                {

v.push_back(ar[j]);
                        j++;
                }
        }
        for(i=0; i<v.size();i++)
        {

ar[i+l]=v[i];
        }
}

                        i++;
                }

                else
                {

v.push_back(ar[j]);
                        j++;
                }
        }
        for(i=0; i<v.size();i++)
        {

ar[i+l]=v[i];
        }
}
int main()
{
        int i = 0;
        Partition(0,7);
        cout<<"After sorting:
"<<endl;
        while(i<8)
        {
                cout << ar[i] <<
" ";
                i++;
  }    return 0;  }
```

Output:


"C:\Users\USER\Downloads\388 (1).exe"
After sorting:
11 12 13 14 19 28 47 96
Process returned 0 (0x0)   execution time : 0.247 s
Press any key to continue.

Quick Sort

Quick Sort

Use of the Algorithm :- To sort the element from a list.

Input of the Algorithm :- An Array of n elements.

Output of the Algorithm :- A sorted Array.

complexity :- O (N log N).

Example :-

(I) Best case :- A = {4,3,2,1} ; O(N log N).

(II) Worst case :- A = {1,2,3,4} , O (N log N).

(III) Avarag case :- A = {2,1,1,3,3,1} O (N²).

Basic Work flow of the Algorithm :-

(I) An array is divided into subarray by selecting a pivot element.

(II) The left and right subarrays are also divded into the same approach, utill it contains a single element.

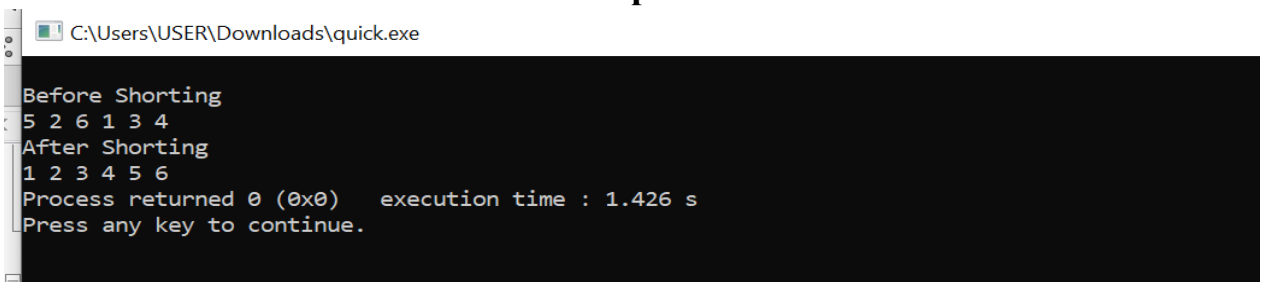
(III) The approch is the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

(IV) At this point, the elements are already sorted.

Source Code:

```
#include <bits/stdc++.h>
#include<iostream>
using namespace std;
int a[]={5,2,6,1,3,4};
int par(int l, int r)
    {
        int pivotindex=l;
        while(l!=r)
        {
            if(pivotindex==l)
            {

if(a[pivotindex]<a[r])
                {
                    r--;
                }
                else
                {

swap(a[pivotindex],a[r]);
                    pivotindex=r;
                }
            }
        else
            {
            if(a[l]<a[pivotindex])
              {
                l++;
              }
            else
              {

swap(a[pivotindex],a[l]);
                pivotindex=l;
               }
            }
      }
return l;
    }
    void qs(int l, int r)
    {
        if(l>=r) return;
        int mid=par(l,r);
        qs(l,mid-1);
        qs(mid+1,r);
    }
int main()
{
     cout<<endl<<"Before
Shorting"<<endl;
     for(int i=0;i<6;i++)
     {
     cout<<a[i]<<" ";
     }
     cout<<endl<<"After
Shorting"<<endl;
     qs(0,5);
     for(int i=0;i<6;i++)
     {
     cout<<a[i]<<" ";
     }
    return 0;
}
```

Output:



Heap Sort

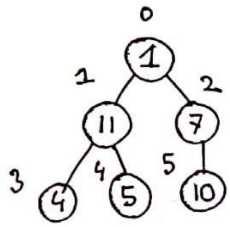
Use Of the Algorithm :- To sort the elements from a tree nodes.

Input of the Algorithm :- An array of n elements.

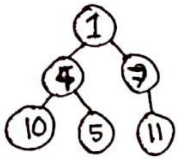
Output of the Algorithm :- An array of n elements.

Complexity :- O (N log N)

Example :-



1	11	7	4	5	10



Min heap

1	4	5	7	10	11

Basic Workflow Of the Algorithm :-

(I) Construct min heap and the min item stored in the root.

(II) Remove the rootelement and put at the end of the array, put the last item

of the tree at the vacant palace.

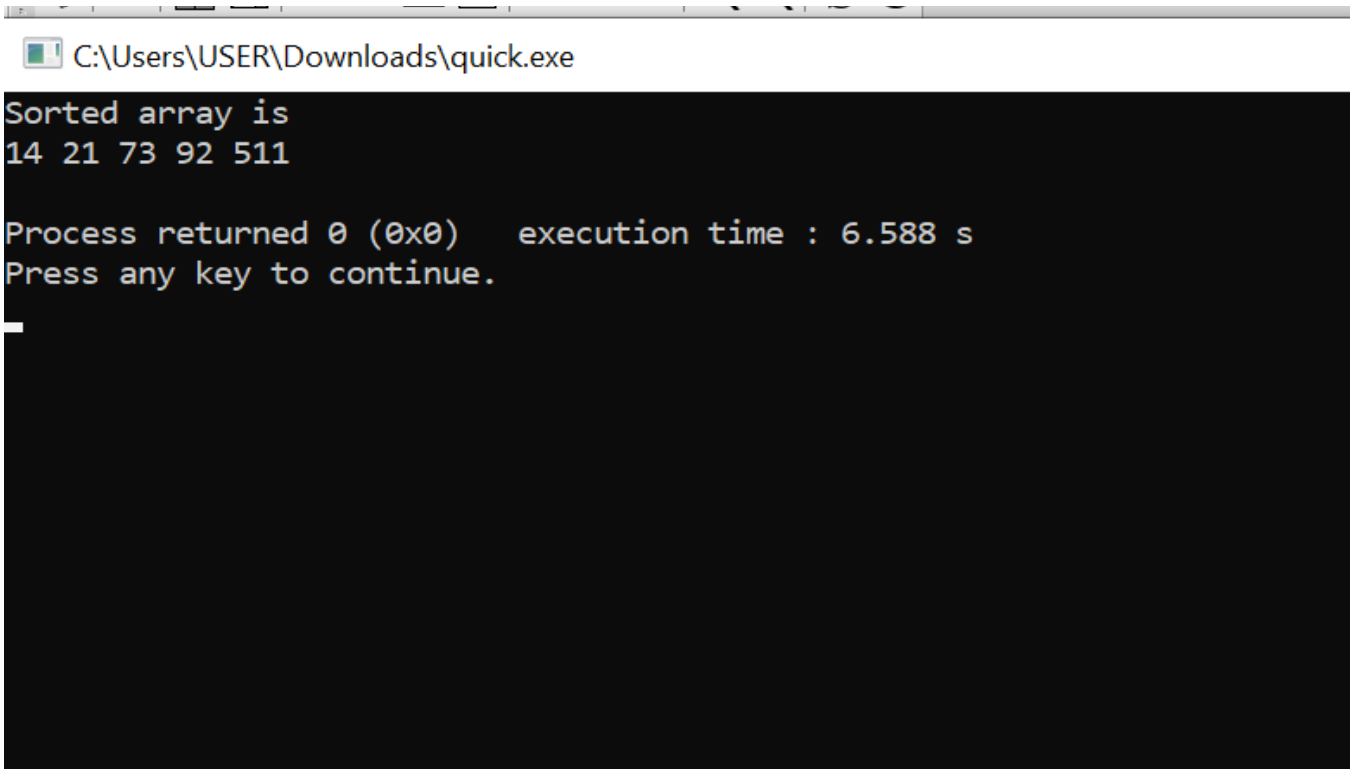
(iii) Reduce the size of heap by 1. Heapify the root element again so that we have lowest element at root. The process is repeated untill all the items of the list are sorted.

Source Code:

```
#include <iostream>
using namespace std;
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] >
arr[largest])
        largest = l;
    if (r < n && arr[r] >
arr[largest])
        largest = r;
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

        heapify(arr, i, 0);
    }
}
```

```
  void heapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0;
i--)
        heapify(arr, n, i);
    for (int i = n - 1; i >= 0; i--)
    {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
int main()
{
    int arr[] = {511,14,73,92,21};
    int n = sizeof(arr) /
sizeof(arr[0]);
    heapSort(arr, n);
    cout << "Sorted array is \n";
    for (int i = 0; i < n; ++i)
    cout << arr[i] << " ";
    cout << "\n";
}
```

Output:


C:\Users\USER\Downloads\quick.exe
Sorted array is
14 21 73 92 511
Process returned 0 (0x0) execution time : 6.588 s
Press any key to continue.

Greedy Algorithms

Activity Selection Problem

Activity Selection problem

Use of the algorithm :-

It is use to find the number of maximum non overlapping activities that can be perform. It is also known as Interval partitioning. We can find largest compatible set of works by this.

Input of the algorithm :-

The Input of the Algorithm is a array which contain starting time and ending time.

Output of the Algorithm :-

largest number of compatible work result in array.

Complexity :- $O(N \log N)$

Basic work flow :

(I) Take a structure which contains Start & Finish time.

(II) Short the Activity According to their finish time.

(III) The shroted first element of the array will print. After get the first Activity of the next activities start time is greter or equal to the finish time of the priviow selected activity then print the value of it.

(IV) Now the iterative proces continue untill reach the destination

Example:

Activities → 8

1 3
0 4
1 2
4 6
2 9
5 8
3 5
4 5

Output:-

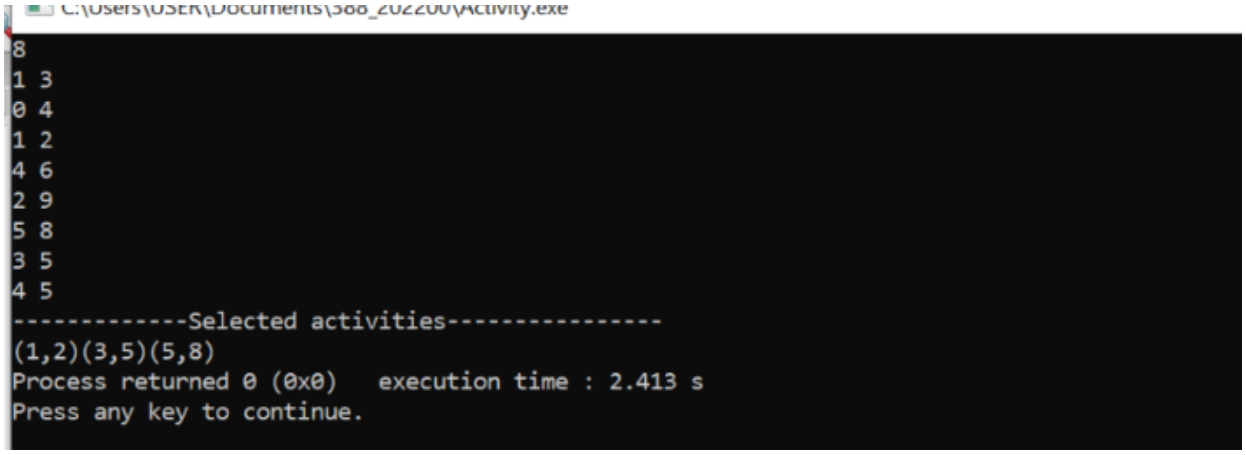
(1,2), (3,5), (5,8)

Source Code:

```
#include<bits/stdc++.h>
using namespace std;
struct Activity_selection{
int begain_;
int finish ;
Activity_selection()
{
}
Activity_selection(int initia_, int
end_)
{
begain_ = initia_;
finish = end_;
}
};
Activity_selection A[101];
bool com(Activity_selection a,
Activity_selection b)
{
if(a.finish<b.finish)
return true;
return false;
}
```

```
int main()
{
int N;
scanf("%d", &N);
for(int i=0; i<N; i++)
{
scanf("%d%d", &A[i].begain_
,&A[i].finish);
}
sort(A,A+N,com);
cout << "-------------Selected
activities----------------\n";
int i = 0;
  cout << "(" <<
A[i].begain_<< "," <<
A[i].finish << ")";
for(int j=1; j<N; j++)
{
if(A[j].begain_>=A[i].finish)
{
cout << "(" <<
A[j].begain_<< "," <<
A[j].finish << ")";
i = j;
}
}
return 0;
}
```

Output:



Optimal Cashing

Optimal Cashing

Use of the Algorithm :- Cash is the fast, small memory which hold a certain amount. It is use for reduces the number of cach miss to a minimum possiable number. Rather than it is use for minimize the number of each miss.

Input of the Algorithm :- Two Strings.

Output of the Algorithm :- Total number of cach miss and total number of cache hit.

complexity :-

$O(n \cdot m)$

Example :-

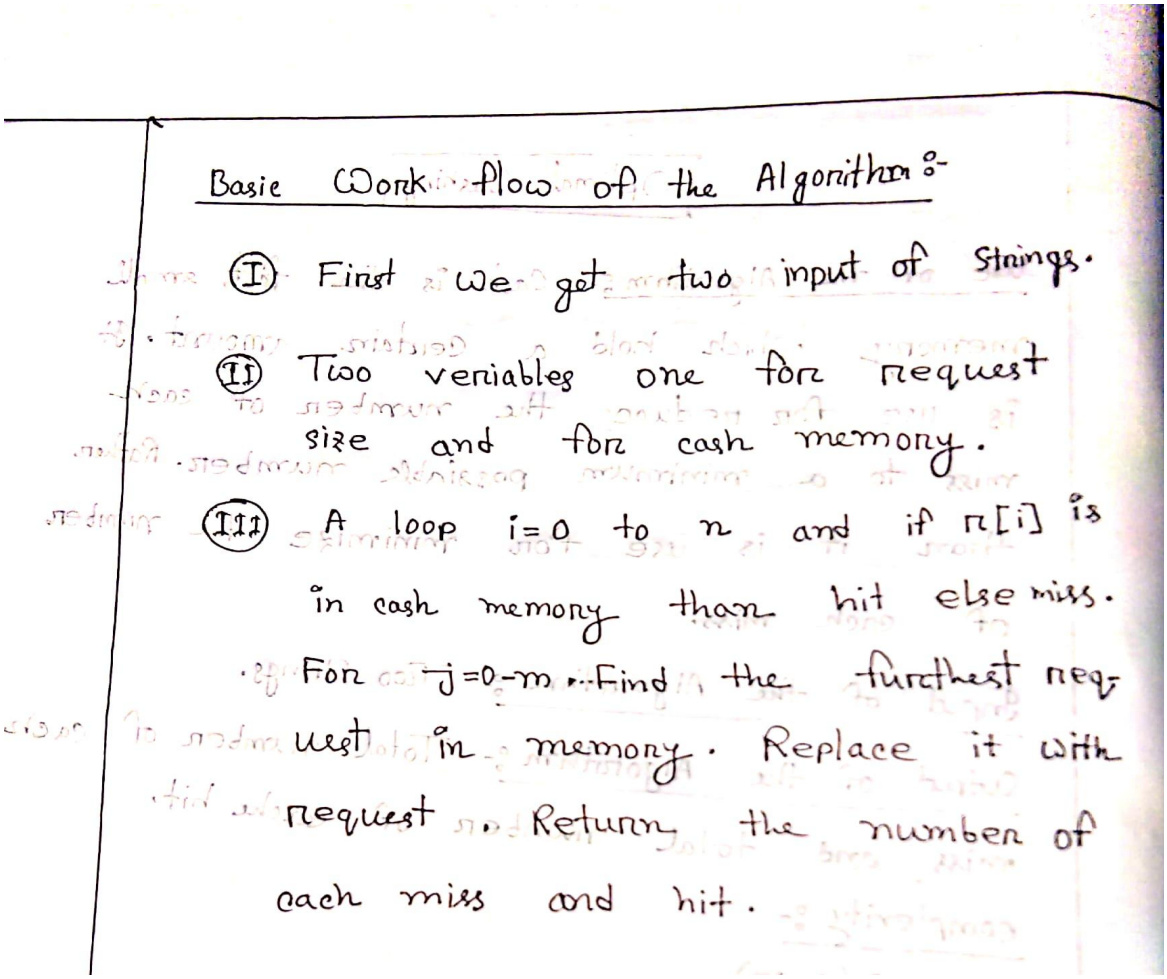
$K = 2$

Initial cash = ab

Request = a, b, c, b, c, a, a, b

Ans.

2 cashe miss



Source Code:

```
#include<bits/stdc++.h>
using namespace std;
string F,R;
int main()
{
int FS;
scanf("%d",&FS);
cout<<"Frame:";
cin>>F;
cout<<"Request:";
cin>>R;
int count=0;
int f_s=F.size();
```

```
else{
cout<<"\t miss \t\t ";
for(int j=0;j<f_s;j++)
{
int dis=INT_MAX;
for(int

k=i+1;k<r_s;k++){

if(F[j]==R[k]){
dis=k-i;
break;
}
```

```
int r_s=R.size();
cout<<"\nRequest \t hit/miss \t
replaced by \t present cache\n";
for(int i=0;i<r_s;i++){
cout<<R[i]<<" \t";
bool hit=false;
for(int j=0;j<f_s;j++){
if(R[i]==F[j]){
hit=true;
break;
}
}
int repl_ind,mx_dist=0;
if(hit)
{
cout<<"\t hit \t\t none

\t\t";
}
```

```
  }
if(mx_dist<dis)
{
mx_dist=dis;
repl_ind=j;
}
}
}
if(!hit)
{
count++;
cout<<F[repl_ind]<<"\t\t";
F[repl_ind]=R[i];
}
for(int
j=0;j<f_s;j++)cout<<F[j];
cout<<"\n";
}
cout<<"Minimum miss number:
"<<count<<"\n";
return 0;
}
```

Output:

Minimize Maximum Lateness

Minimizing Maximum Lateness

* Minimum Latness Algonithm is used for to find the maximum lateness.
* The input of the Algonithm is structure [user define data type].
* The output is integer number.
* Complexity is NlogN.

Workflow

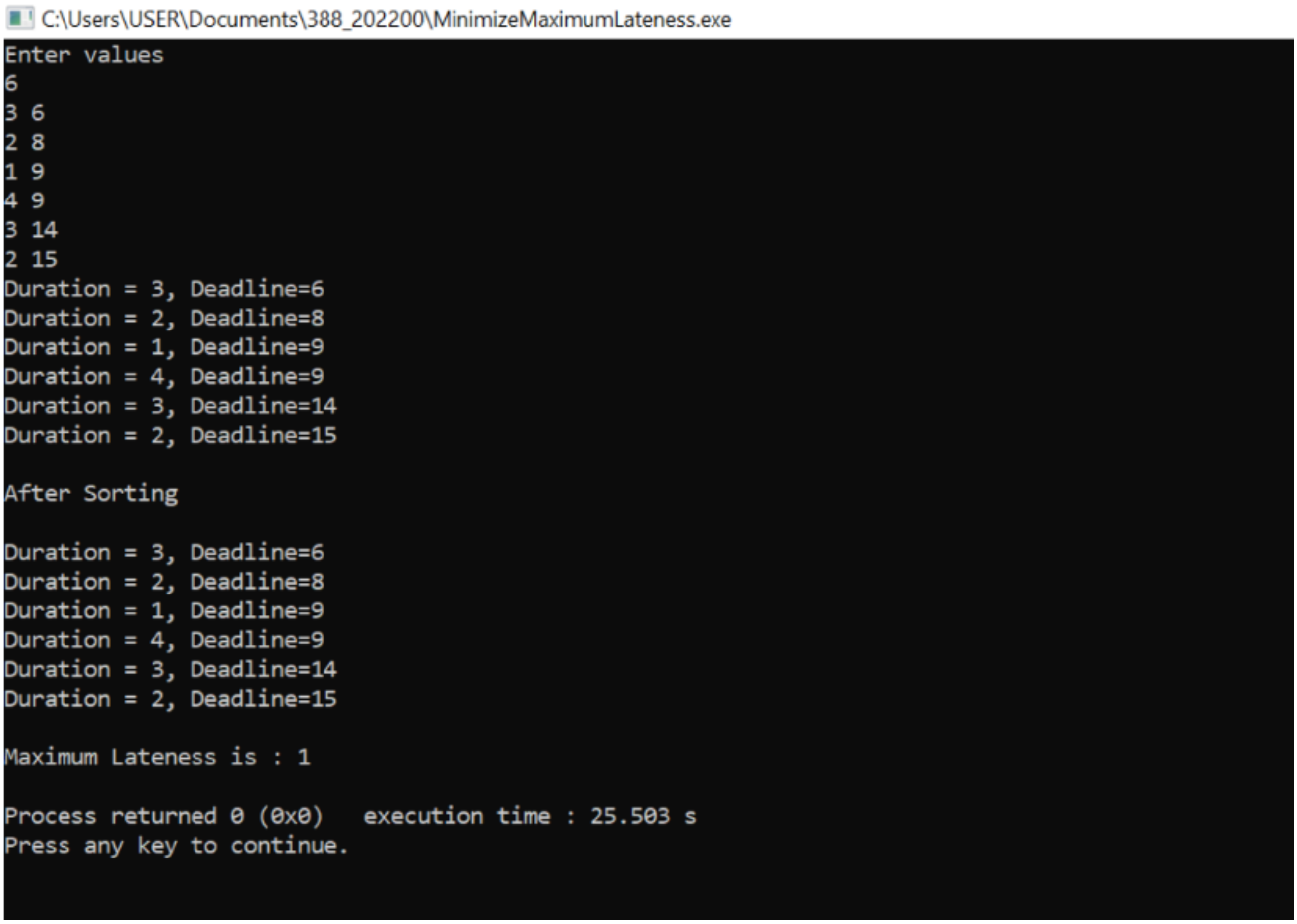
(I) User define data type Event to get duration & deadline
(II) Short the event.
(III) lateness = max(0, k-d)
(IV) Short the latenness.
(V) Output is the final index of lateness.

Source Code:

```
#include<bits/stdc++.h>
#include<bits/stdc++.h>
using namespace std;
struct Event
{
int duration;
int deadline;
Event(){}
Event(int du, int de)
{
duration =du;
deadline =de;
}
void print()
{
printf("Duration = %d,
Deadline=%d\n",duration,deadline);
}
};
Event E[100];
bool com(Event a, Event b)
{
if(a.deadline<b.deadline)
return true;
return false;
}
int main()
{
int N;
cout<<"Enter values\n";
cin>>N;
for(int i=0;i<N;i++)
{
scanf("%d%d",&E[i].duration,&E[i].dea
dline);
}
for(int i=0;i<N;i++)
{
E[i].print();
}
```

```
 sort(E,E+N,com);

cout<<"\n";

cout<<"After Sorting\n\n";

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

{


E[i].print();

}

int l[1001];

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

{

int k;

k=k+E[i].duration;

int d=E[i].deadline;

l[i]=max(0,k-d);

}

sort(l,l+N);

cout<<"\n";

cout<<"Maximum Lateness is :
"<<l[N-1];

cout<<"\n";

return 0;

}
```

Output:


C:\Users\USER\Documents\388_202200\MinimizeMaximumLateness.exe
Enter values
6
3 6
2 8
1 9
4 9
3 14
2 15
Duration = 3, Deadline=6
Duration = 2, Deadline=8
Duration = 1, Deadline=9
Duration = 4, Deadline=9
Duration = 3, Deadline=14
Duration = 2, Deadline=15
After Sorting
Duration = 3, Deadline=6
Duration = 2, Deadline=8
Duration = 1, Deadline=9
Duration = 4, Deadline=9
Duration = 3, Deadline=14
Duration = 2, Deadline=15
Maximum Lateness is : 1
Process returned 0 (0x0) execution time : 25.503 s
Press any key to continue.

Fractional Knapsack

Fractional Knapsack

Use of the Algorithm :-

When a set of items each with a weight and a value, a subset of items has to choose such that total weight of the items is less then or equal to the size of the knapsack and the profit is maximum. It is use to maximize the total value of knapsack by breaking items.

Inputs of the Algorithm :-

An annay contain weight and profit.
The knasack size also.

Output : Output is maximum profit of the choosen items.

Complexity :- O(NlogN)

Example :

Profit = {12, 32, 40, 30, 50}
Weight = {4, 8, 2, 6, 13}

Maximum profit = 124

Basic work flow :-

(I) Need a user difine data type to contain profit and weight.

(II) Then the ratio of value and weight Calculation for each item.

(III) Short all the items in decreasing oders From the ration we get.

(IV) gf weight of current item is less or equal to remaining capacity the sum the value into the result or break the item add the profit or the final value.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;
struct item
{
int value,weight;
item(){}
item(int v, int w)
  finalvalue += arr[i].value;

}
else {
finalvalue+=
arr[i].value*((double)W /
(double)arr[i].weight);
```

```
{
value = v;
weight = w;
}
};
item arr[101];
bool cmp(item a,item b)
{
double f1 = (double)a.value /
(double)a.weight;
double f2 = (double)b.value /
(double)b.weight;
return f1 > f2;
}

double frac_Knap(int W,item arr[],
int N)
{
sort(arr, arr + N, cmp);
double finalvalue = 0.0;
for (int i = 0; i < N; i++)
{
if (arr[i].weight <= W)
{
W -= arr[i].weight;
```

```
break;
}

}
return finalvalue;
}
int main()
{
int W,N;
cout<<"Enter the number of
items\n";
cin>>N;
cout<<"Enter the weight of
Knapsack\n";
cin>>W;
cout<<"Enter the item
Values:\n";
for(int i=0;i<N;i++)
{
cin>>arr[i].value;
cin>>arr[i].weight;
}
cout << "Maximum value =
"

<< frac_Knap(W, arr,

N);
return 0;
}
```

Output:

```
C:\Users\USER\Documents\388_202200\FructionalKnapsac.exe
Enter the number of items
5
Enter the weight of Knapsack
10
Enter the item Values:
12 4
32 8
40 2
30 6
50 1
Maximum value  = 124
Process returned 0 (0x0)   execution time : 35.024 s
Press any key to continue.
```

Huffman Coding

Huffman Coding

Use of the Algorithm:-

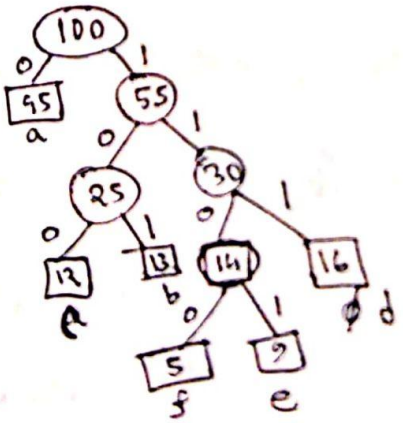
Huffman codding is a method of data compression that is indendent of data type. It reduce the overall size of file.

Input:- Pair of Intiger and String.

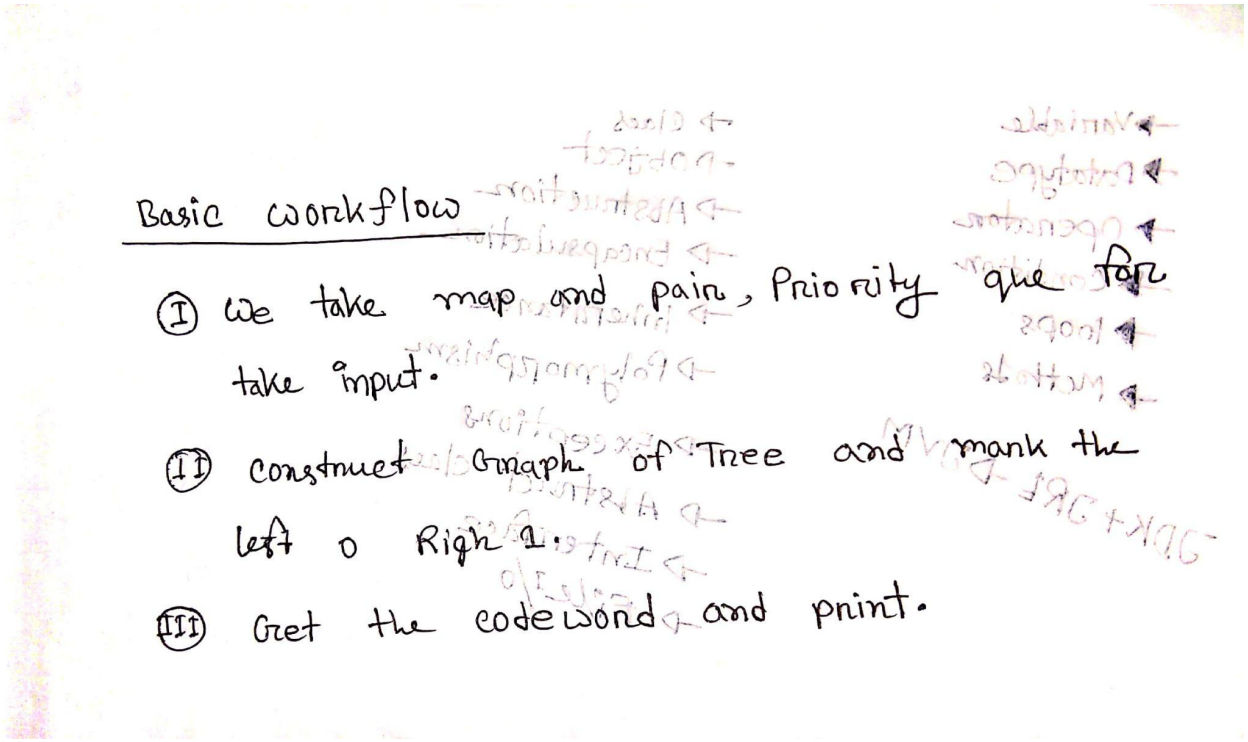
Output:- Lenth and Codward.

Complexity:- $O(n \log n)$.

Example:-



codeword	frequency
a	0
b	101
c	100
d	111
e	1101
f	1100



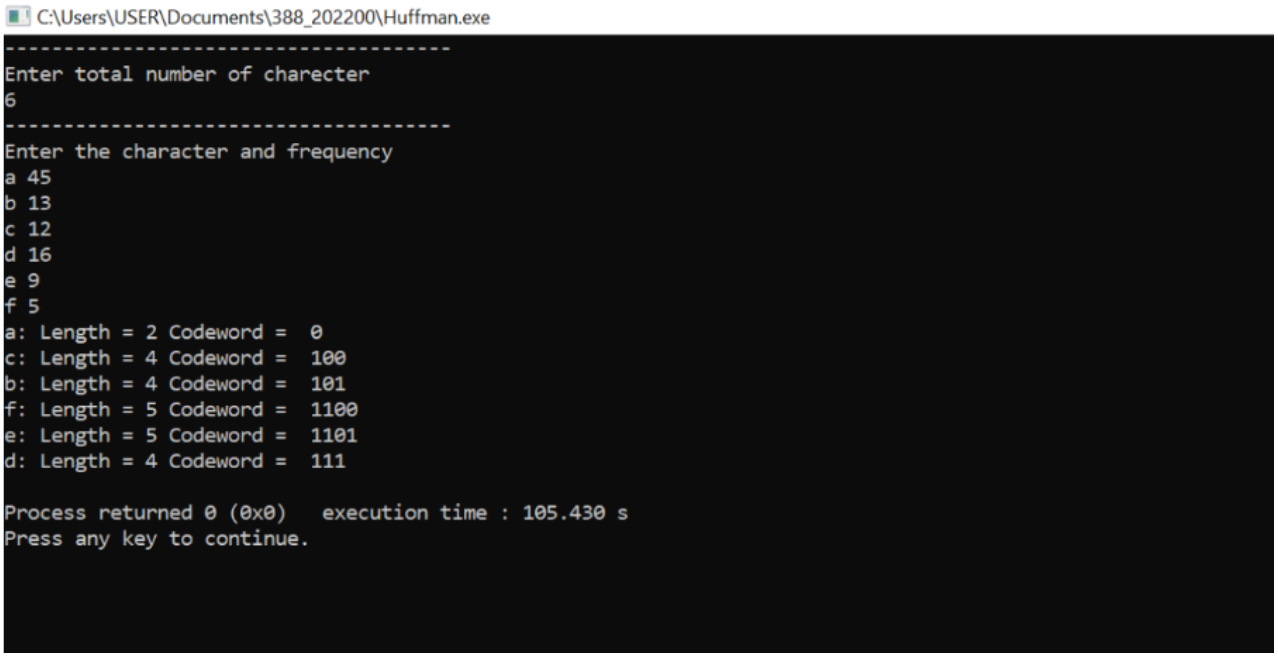
Source Code:

```
#include <bits/stdc++.h>
using namespace std;
#define N '\n'
#define ll long long
map<pair<int,string>,int>M;
pair<int,string>node[100];
vector<int>G[100];
priority_queue<pair<int,string>,vecto
r<pa
ir<int,string>>,greater<pair<int,stri
ng>>
>PQ;
void cpy(int n,string code)
{
if(G[n].size()>1)
{
cpy(G[n][0],code+"0");
cpy(G[n][1],code+"1");
}
else
{
cout<<node[n].second<<": Length =
"<<code.size()<<" Codeword =
"<<code<<N;
  while(PQ.size()>1)
{
pair<int,string>a,b,c;
a=PQ.top();
PQ.pop();
b=PQ.top();
PQ.pop();
c.first=a.first+b.first;
c.second=a.second+b.second;
PQ.push(c);
cnt++;
M[c]=cnt;
node[cnt]=c;
int x,y,z;
x=M[a];
y=M[b];
z=M[c];
G[z].push_back(x);
G[z].push_back(y);
}
int p=M[PQ.top()];
cpy(p," ");
}
int main()
```

```
}
}
void huffman(int number)
{
ll cnt=0;
for(ll i=0; i<number; i++)
{
string s;
int k;
cin>>s;
cin>>k;
pair<int,string>
A=make_pair(k,s);
M[A]=i+1;
node[i+1]=A;
PQ.push(A);
cnt++;
}
```

```
{
cout<<"-------------------------
-------------"<<N;
cout<<"Enter total number of
charecter"<<N;
ll number;
cin>>number;
cout<<"-------------------------
-------------"<<N;
cout<<"Enter the character
and frequency"<<N;
huffman(number);
return 0;
}
```

Output:



Dynamic programming

Longest Increasing Subsequence

Longest Increasing Subsequence

Use of the Algorithm :- To Find the length of long increasing subsequence of a given sequence.

Input of the Algorithm :- An Array.

Output of the Algorithm :- Lenth of the subsequence.

complexity :- $O(N^3)$.

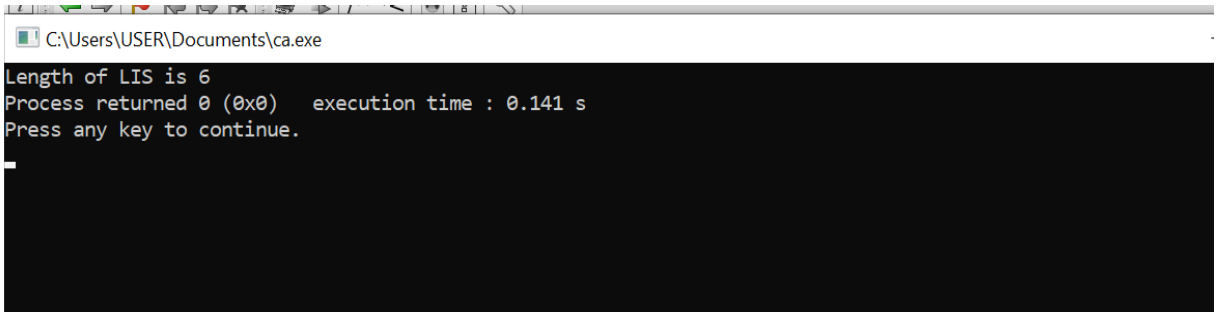
Basic Workflow of the Algorithm :- By using Dynamic programming this type of problem can be solve. Here we can get overlapping sub problems. By Dymic programming technique we can memonize this and apply on need. By Tabulan format techniq we can memonize it. It is longest increasing of an array. We can take the input and performing dynamic programming technique on it.

Source Code:

```
#include <iostream>
using namespace std;
int _lis(int arr[], int n, int*
max_ref)
{
      if (n == 1)
            return 1;
      int res, max_ending_here = 1;
      for (int i = 1; i < n; i++) {
            res = _lis(arr, i,
max_ref);
            if (arr[i - 1] < arr[n -
1]
                  && res + 1 >
max_ending_here)
                  max_ending_here =
res + 1;
      }
      if (*max_ref < max_ending_here)
            *max_ref =
max_ending_here;
      return max_ending_here;
}
```

```
  #include <iostream>
using namespace std;
int _lis(int arr[], int n, int*
max_ref)
{
      if (n == 1)
            return 1;
      int res, max_ending_here = 1;
      for (int i = 1; i < n; i++) {
            res = _lis(arr, i,
max_ref);
            if (arr[i - 1] < arr[n -
1]
                  && res + 1 >
max_ending_here)
                  max_ending_here =
res + 1;
      }
      if (*max_ref <
max_ending_here)
            *max_ref =
max_ending_here;
      return max_ending_here;
}
```

Output:



Longest Common Subsequence

Use of the Algorithm :- Find the longest common subsequence between two strings.

Input of the algorithm :- Two string.

Output of the Algorithm :- Longest common subsequence and it's Lenth.

Complexity :- $O(N^2)$.

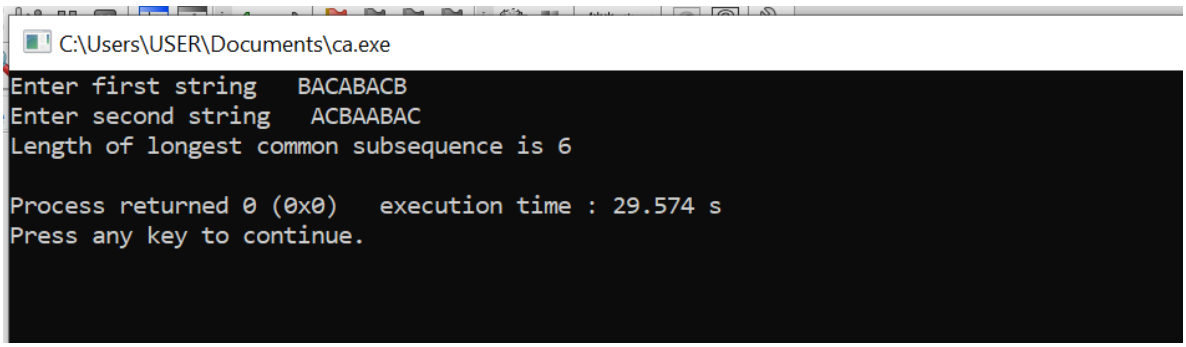
Basic workflow :- By doing tabular format this kinds of problem can be solved. If s1 and s2 are two given ~~subs~~ sequences then z is the common subsequence of both s1 & s2. The method of dynamic programming reduces the number of funtion calls. It stores the result of each funtion call so that it can be used in future call.

Source Code:

```
#include<bits/stdc++.h>
using namespace std;
int lcs(string str1, string str2, int
len1, int len2)
{
    int i, j;
    int LCS[len1+1][len2+1];
    for(i=0; i<=len1; i++)
        LCS[i][0]=0;
    for(j=0; j<=len2; j++)
        LCS[0][j]=0;
    for(i=1; i<=len1; i++)
    {
        for(j=1; j<=len2; j++)
        {
            if(str1[i-1]==str2[j-1])
            {

LCS[i][j]=1+LCS[i-1][j-1];
            }
            else
            {

LCS[i][j]=max(LCS[i-1][j],LCS[i][j-1]
);
            }
        }
    }
    return LCS[len1][len2];
}
```

```
 int main()
{
    string str1,str2;
    cout<<"Enter first string   ";
    getline(cin, str1);
    cout<<"Enter second string   ";
    getline(cin, str2);
    int len1=str1.length();
    int len2=str2.length();
    cout<<"Length of longest common
subsequence is
"<<lcs(str1,str2,len1,len2);
    cout<<endl;
    return 0;
}
```

Output:



0/1 knapsack Problem

0/1 Knapsack

Use of the algorithm :- By filling the knapsack of size N, we have to maximize the profit.

Input of the algorithm :- Weight of the items, Values of the item, size of the knapsack.

Output of the Algorithm :- Maximum profit.

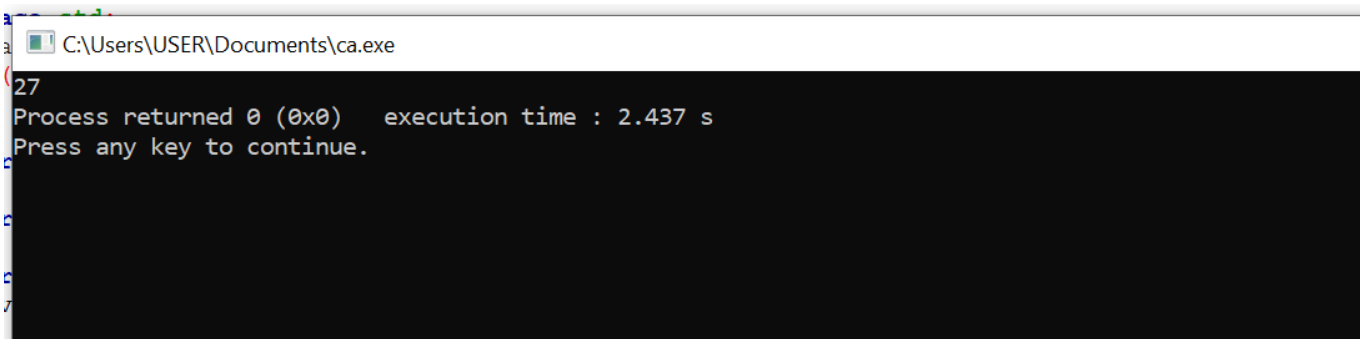
Complexity : O(N*W).

Basic Workflow :- By DP[][] table let's consider all the posible weights from 1 to "W" as the colourns and weights can be kept in the rows. The state DP[i][j] will denote maximum value of j weight considering all values from 1 to ith. By using tabular format technique we can solve this kind of problem. By sequencially Do this we can get the maximum profit.

Source Code:

```
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b)
{
      return (a > b) ? a : b;
}
int knapSack(int W, int wt[], int
val[], int n)
{
      int i, w;
      vector<vector<int>> K(n + 1,
vector<int>(W + 1));
      for(i = 0; i <= n; i++)
      {
            for(w = 0; w <= W; w++)
            {
                  if (i == 0 || w ==
0)
                        K[i][w] = 0;
                  else if (wt[i - 1]
<= w)
                        K[i][w] =
max(val[i - 1] +

            K[i - 1][w - wt[i - 1]],

            K[i - 1][w]);
                  else
                        K[i][w] = K[i
- 1][w];
            }
      }
      return K[n][W];
}

int main()
{
      int val[] = {12,10,9,6};
      int wt[] = {2,4,3,1};
      int W =6;
      int n = sizeof(val) /
sizeof(val[0]);

      cout << knapSack(W, wt, val,
n);

      return 0;
}
```

Output:



Graph Algorithms

Breadth First search (BFS)

BFS

Use of the Algorithm :- To visit all the node of the graph.

Input of the graph :- Vertices and Edges of Graph.

Output of the Algorithm :- All visited node of the graph.

complexity :- $O(V+E)$

Basic Work flow of the Algorithm :-

(I) Declear a queue and insert the starting vertex.

(II) Initialize a visited array and mark the starting vertex as visited.

(III) By following below process untill the queue becomes empty :-

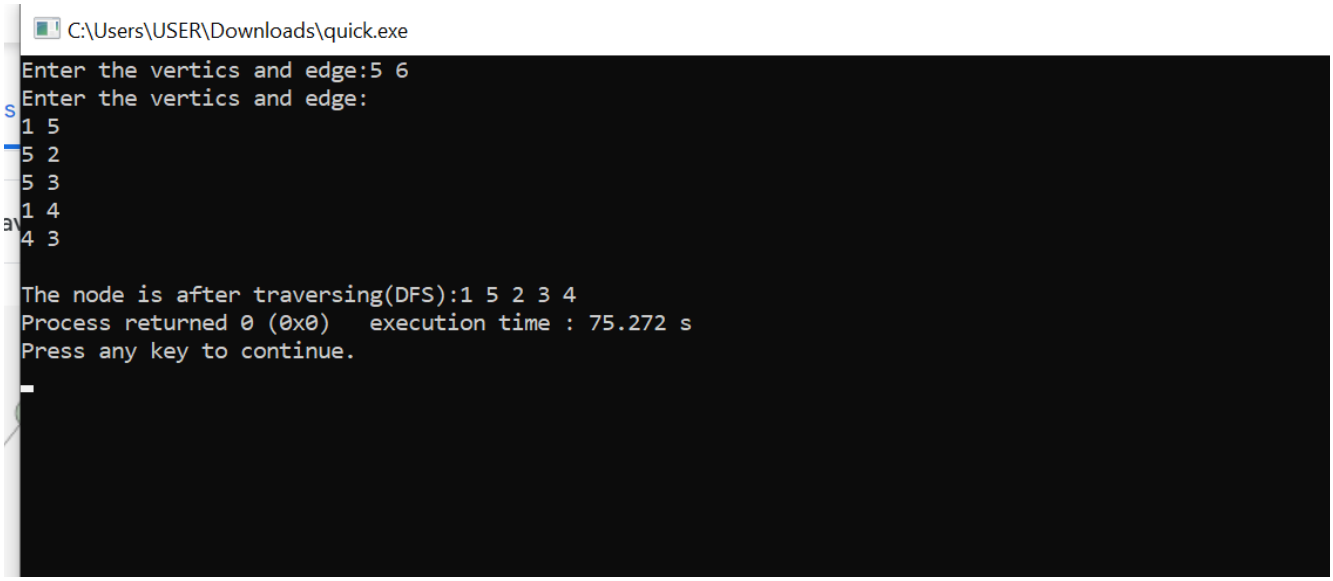
1. Remove the first vertex of the queue.
2. Mark that vertex as visited.
3. Insert all the unvisited nighbour of the vertex into the queue.

Source Code:

```
#include<bits/stdc++.h>
using namespace std;
vector<int>G[1000];
bool visited[100];
void dfs(int src)
{
  cout<<src<<" ";
  visited[src]=true;
  for(auto it:G[src])
  {
  if(!visited[it])
  {
      visited[it]=true;
      dfs(it);
  }
  }
}
```

```
int main()
{
    int edge,vertex,i,v,u;
    cout<<"Enter the vertics and
edge:";
    cin>>edge>>vertex;
    cout<<"Enter the edge:\n";
    for(i=0;i<edge;i++)
    {
        cin>>u>>v;
        G[u].push_back(v);
        G[v].push_back(u);
    }
    cout<<"\nThe node is after
traversing(DFS):";
    dfs(1);
    return 0;
}
```

Output:



Depth First search (DFS)

DFS

Use of the algorithm :- To traverse all the nodes of a graph.

Input of the Algorithm :- Vertices and edges of a Graph.

Output of the Algorithm :- All visited vertex of the graph.

complexity :- $O(V+E)$

Basic Workflow :-

(I) First put any of the node of the graph at the top of a stack.

(II) Takout the top most item of the stack and add it to visited list.

(III) Create a list of that vertex adjacent nodes. Add the one's which aren't in the visited list to the top of the Stack.

(IV) Keep Repeating step 2 & 3 untill the Stack is empty.

Source Code:

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;
vector<int>G[100];
bool visited[1000];
int main()
{
    int vertex,edge,i,u,v;
    cout<<"Enter the vertics and
edge:";
    cin>>vertex>>edge;
    cout<<"Enter the Edges:\n";
    for(i=0;i<edge;i++)
    {
        cin>>u>>v;
        G[u].push_back(v);
        G[v].push_back(u);
    }
```

```
  queue<int>q;
    q.push(0);
    visited[0]=true;
    while(!q.empty())
    {
        int x = q.front();
        q.pop();
        cout<<x<<" ";
        for(auto it:G[x])
        {
            if(!visited[it])
            {
                q.push(it);
                visited[it]=true;
            }
        }
    }
    return 0;
}
```

Output: