RAJSHAHI UNIVERSITY OF ENGINEERING & TECHNOLOGY

Online Lab Final

COURSE NAME: SESSIONAL BASED ON CSE-2201

COURSE CODE: CSE-2102

SUBMITTED TO-

DR. MD. ALI HOSSAIN
Associate Professor
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology

BIPRODIP PAL
Assistant Professor
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology

SUBMITTED BY-

SRABONTI DEB
Roll- 1803163
Section-C
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology

SUBMISSION DATE – 9 August,2021

Problem 1: Sort a given set of elements using the Quicksort method and determine the time required to sort the elements in millisecond. Repeat the experiment for 3 different values of n (i.e, n should be > 1000 for each case) which the number of elements in the list to be sorted should be read create a bar graph of the time taken versus n. The number of element n to be sorted should be read from a file "quick1.txt", "quick2.txt", "quick3.txt"" and generate the lists of numbers randomly.

Algorithm:

```
Quicksort (arr, left, right)
if left < right
then pi ⟵ partition (arr, left, right)
        Quicksort (arr, left, pi-1)
        Quicksort (arr, right pi+1, right)
```

(To sort an entire array arr, we initially call is Quicksort (arr, 1, length[arr]))

```
Partition (arr, left, right)
pi ⟵ arr [right]
i ⟵ left - 1
for j ⟵ left to right - 1
```

do if arr[j] ≤ pi
then i ← i+1.
exchange arr[i] ↔ arr[j]
exchange arr[i+1] ↔ A[right]
return (i+1).

Code:

```
#include<bits/stdc++.h>
using namespace std;
using namespace std::chrono.
void swap(int *a, int *b)
{
  int t;
  t= *a;
  *a = *b;
  * b = t;
}
int partition (int arr[], int left, int right)
{
  int i = left-1;
  int j;
  int pi = arr[right];
  for(j = left; j <=right-1; j++)
  { if(arr[j] < pi)
    {  i++;
       swap(&arr[i], &arr[j]);
    }
```

```
}
swap(&arr[i+1], &arr[right]);
return(i+1);
void QuickSort(int arr[], int left, int right)
{ if (left<right)
  { int pi = partition(arr, left, right);
    QuickSort(arr, left, pi-1);
    QuickSort(arr, pi+1, right);
  }
}
void printarray(int arr[], int size)
{ int i;
  for(i=0; i<size; i++)
  { cout<<arr[i]<<" ";
  }
    cout<<endl;
}
int main()
{ int n, i=0, a[100000];
  ifstream file;
  file.open("quick1.txt");
  if (file)
```

```
    {
      while (file >> n)
       {
          a[i++] = n;
       }
    }
   else
     cout << "file can't open" << endl;
     file.close();
     auto count1 = chrono::steady_clock::now();
     QuickSort (a, 0, i);
     auto count2 = chrono::steady_clock::now();
     double d = double (chrono::duration_cast
                  <chrono::nanoseconds>(count2 - count1).count())
```

cout << "the time required to sort the elements of 'quick1.txt': " << (double) (d * 1.0 / 1000000) << "millisecond" << endl;

N.B.[similarly we will do same steps for taking data from "quick2.txt" and "quick3.txt".]

}

Code for file generation:

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
  str.srand (time (NULL));
```

```
ofstream fout;
long i=0, n=0;
int num[10000];
fout.open("quick1.txt");
cout<<"Give how many number you want
to generate in "quick1.txt:";
cin>>n;
for(i=0; i<n;++i)
{
  num[i]=rand();
  fout<<num[i]<<endl;
}
fout.close();
```

N.B: [We will do these steps for generating file for "quick2.txt" and "quick3.txt"]

}

Input:

quick1.txt - Notepad

File Edit Format View Help

20626	8868	12947	30857	27853	1852	13611
19163	9671	4541	9626	14581	20353	8626
4751	20187	24001	21596	31381	19961	5394
423	9530	12728	19724	1372	6001	25166
10197	10392	10038	2522	25320	16253	29296
16304	9493	20263	15643	32241	28105	20450
31398	5472	2448	24474	1690	4976	12248
30867	20605	20130	11768	90	32502	3341
17265	17147	20018	31038	4979	11892	11653
22851	20387	25194	24661	15363	27	26651
9780	4305	3250	9876	11852	23872	27518
26403	23040	1336	6296	32272	4816	22857
15135	27528	14205	24384	7040	2266	25710
2296	20212	5606	6995	19827	15434	15057
3592	22569	24038	20990	4284	17056	25866
3642	30546	3133	8788	20791	25095	17498
22408	17980	15702	6809	22160	5402	31001
18807	4545	16738	15088	3876	352	2517
4070	10386	613	23580	24914	715	4734
12474	12540	5769	3415	25406	24705	8704

Ln 1, Col 1 100% Windows (CRLF) UTF-8

Output:

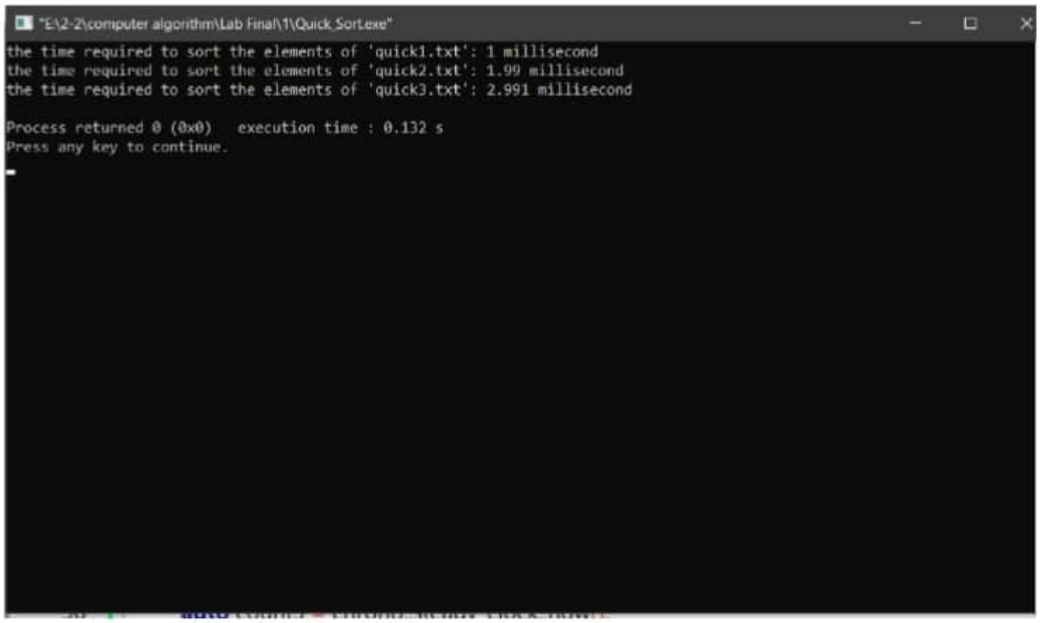
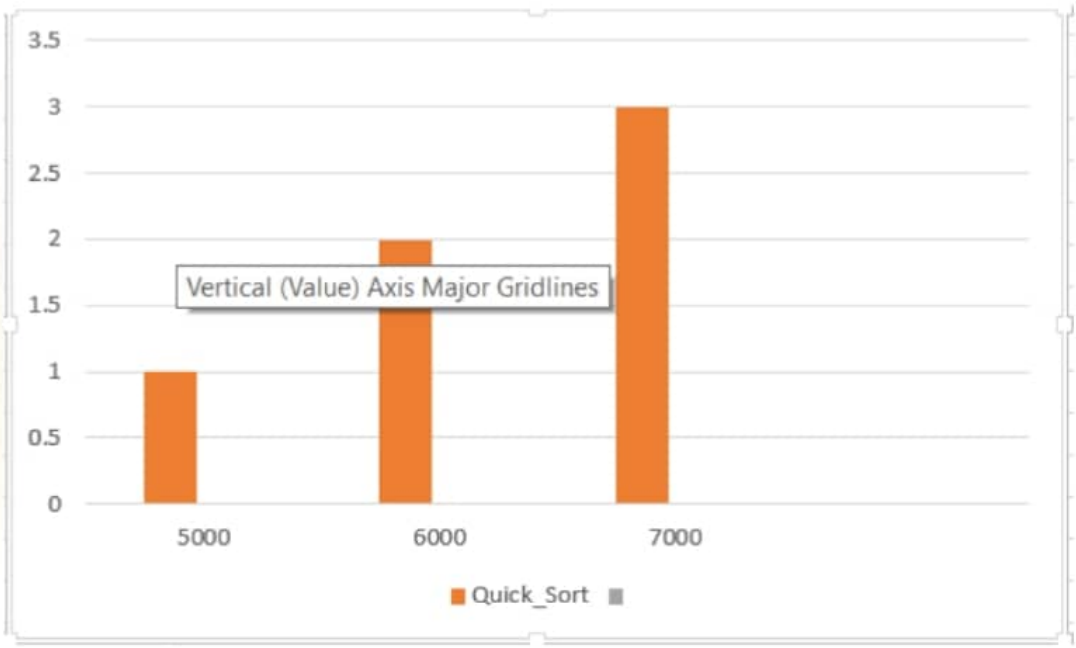


Table:

number of inputs	Quick_Sort
5000	1
6000	1.99
7000	2.991

Graph:



Problem 2: Solve the experiment defined at Q1 using Merge Sort algorithm. Create a bar graph for same problem and compare the required time of merge sort with the Quicksort algorithm.

Algorithm: Mergesort(l,h)

```
{
  if (l<h)
  {
    m=(l+h)/2;
    Mergesort(arr, l, m);
    Mergesort(arr, m+1, h);
    merge(arr, l, m, h);
  }
}
Merge(l, m, h)
i←l, j←m+1, k←l.
while((i<=m) and (j<=h))
{
  if(arr[i]<=arr[j])
  {
    b[k++]:=arr[i++];
  }
  else
  {
    b[k++]:=arr[j++];
  }
```

```
if ( i > m)
 {
   while (j ≤ h)
       {
         b [k++] := arr[j++];
       }
  }
else
  {
    while ( i ≤ m)
     {
       b[k++] := arr[i++];
     }
  }
for p ← l to h
    arr[p] := b[p]
```

Code:

```
#include <bits/stdc++.h>
using namespace std;
int arr[100000];
void merge (int arr[], int l, int m, int h)
{ int b[h], i = l, j = m + 1, k = l;
  while (i <= m && j <= h)
   { if (arr[i] <= arr[j])
```

```
    }
      b[k++]=arr[i++];
    }
    else
    {
      b[k++]=arr[j++];
    }
  }
  if(i>m)
  {
   while(j<=h)
    {
      b[k++]=arr[j++];
    }
  }
  else
   {
    while(i<=m)
    {
      b[k++]=arr[i++];
    }
  }
  for(int p=l;p<=h;p++)
   {
    arr[p]=b[p];
   }
}
```

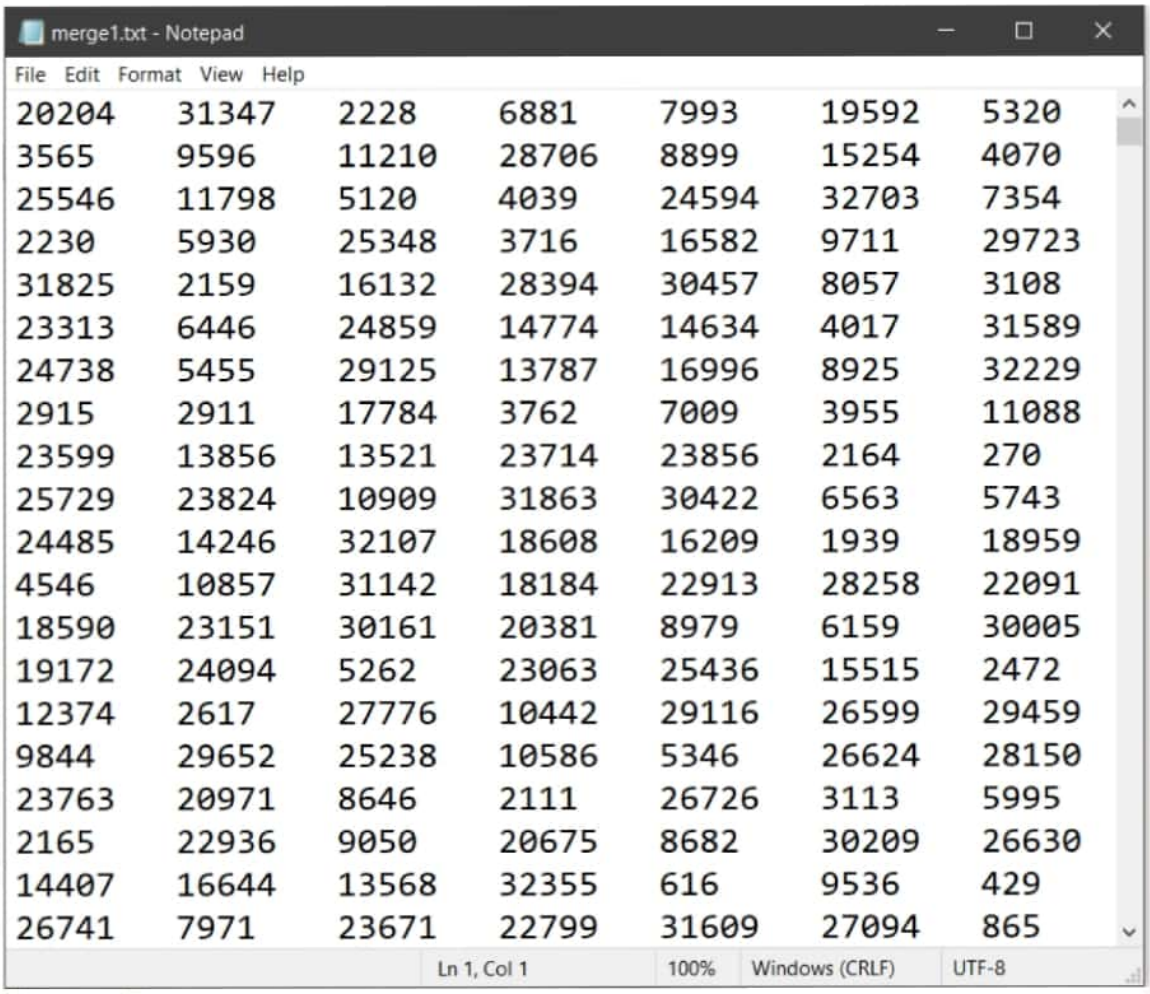
```
void Merge_sort (int arr[], int l, int h)
{
    int m;
    if(l<h)
    {
     m = (l+h)/2;
     Merge_sort (arr, l, m);
     Merge_sort(arr, m+1, h);
     merge (arr, l, m, h);
    }
}
int main()
{
    int i;
    int j=0;
    ifstream file;
    file.open ("merge1.txt");
    while (!file.eof())
    {
     file >> arr[j];
     j++;
    }
    file.close();
 auto count1 = chrono::steady_clock::now();
Merge_sort(arr, 0, j-1);
auto count2 = chrono::steady_clock::now();
double d = double(chrono::duration_cast
  <chrono::nanoseconds> (count2-coun1).coint());
```

cout << " the time required to sort the element of 'merge1.txt': " << (double) (d * 1.0/1000000) << " millisecond " << endl;

j=0;

[N.B: Similarly we will do same steps for taking data from 'merge2.txt' and 'merge3.txt']

Input:



Output:

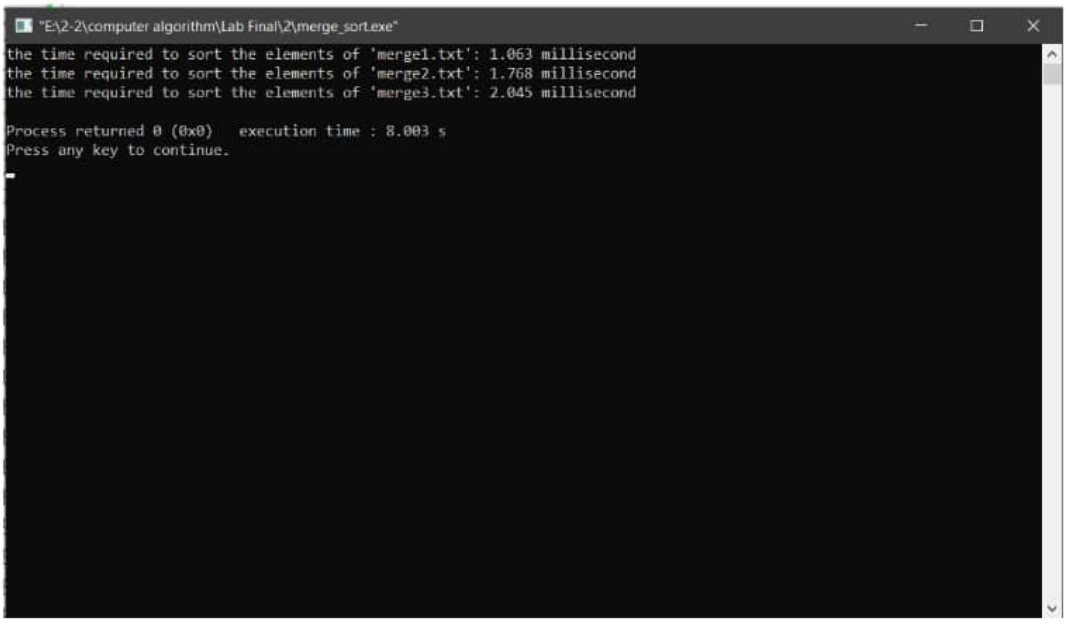
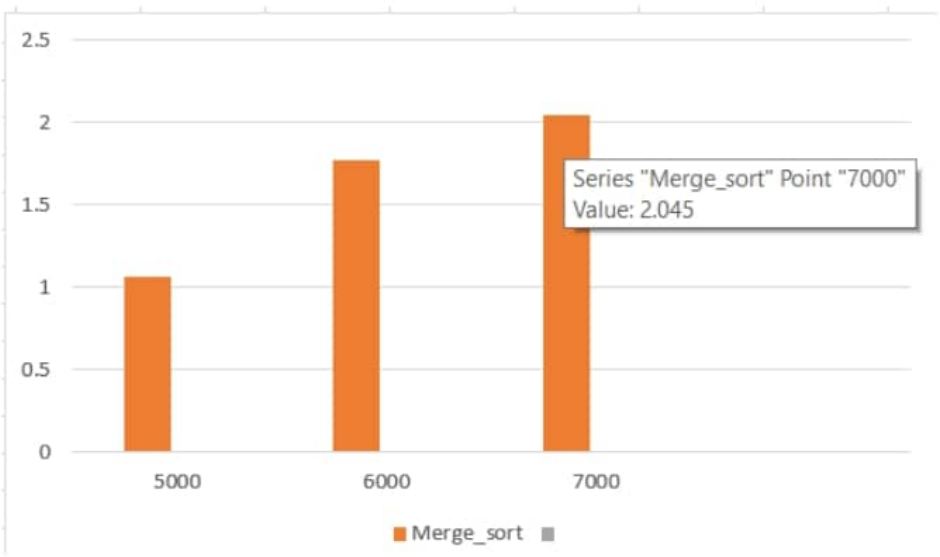


Table:

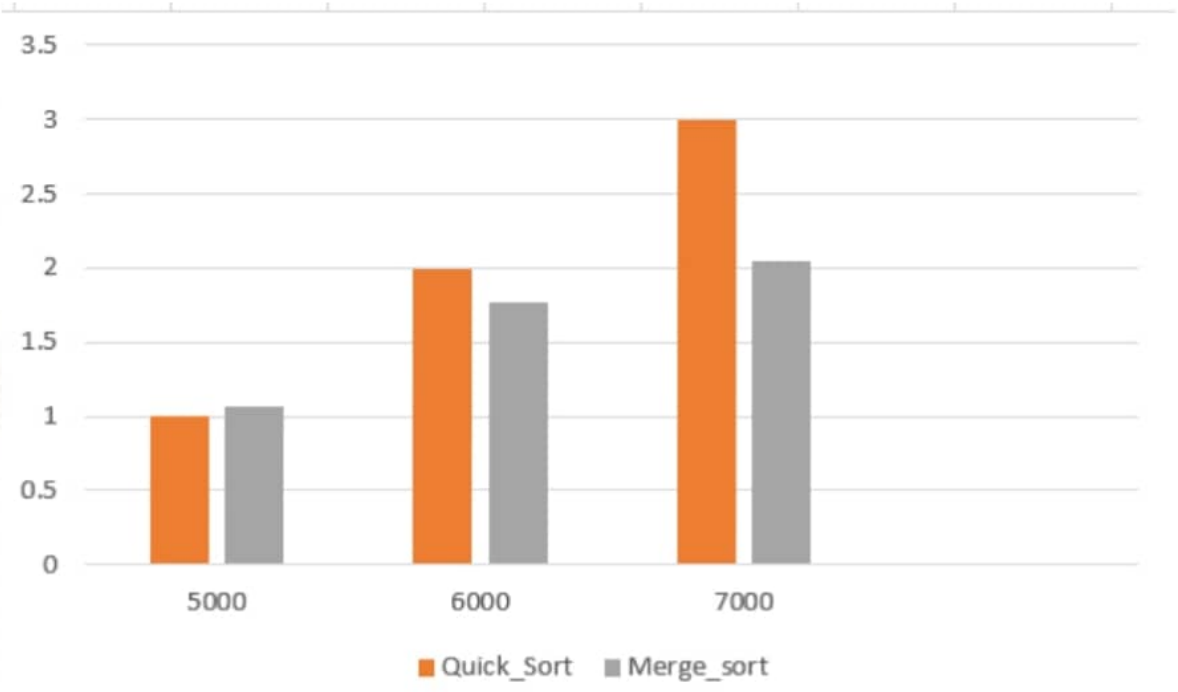
	A	B
1	number of inputs	Merge_sort
2	5000	1.063
3	6000	1.768
4	7000	2.045
5		

Graph:



A bar graph for comparing the required time of merge sort with the quick sort algorithm:

number of inputs	Quick_Sort	Merge_sort
5000	1	1.063
6000	1.99	1.768
7000	2.991	2.045



Problem 3) Implement 0/1 and fractional knapsack problem using Dynamic and Greedy approach programming. Consider the following dataset for the above problems: A knapsack of size 11kg, there are 5 items need to be chosen for the given task and their weight and values are as follows: weights $(w_1, w_2, w_3, w_4, w_5)$ $=(1,2,5,6,7)$ and values $(v_1, v_2, v_3, v_4, v_5)=(1,6,18,22,28)$. Using the 0/1 and fractional knapsack approach fill up the knapsact to maximize the values. Print out the optimal solution for each of the knapsack problem and compare the results.

Algorithm: (0/1 knapsack)

```
DKP ( arr[ ], obj-no, c)
{
 #dynamic allocation of obj-no. and c.
for ( i=01 to obj-no)
{ for(j=0 to c)
 {
    if (j-obj-arr[i].w<0)
       arr[i][j ] := arr[i-1][j];
```

```
        else
          arr[i][j] := max[arr[i-1][j], arr[i-1][j-obj-
    else
      arr[i][j] := max[arr[i-1][j], arr[i-1][j-obj-arr[i].w]
                        + obj-arr[i].p)
      # here w is the weight and p is the
        profit.
    }
  }
}
```

Code:

```
#include<bits/stdc++.h>
using namespace std;
class obj
{
public:
        int p;
        int w;
        bool f;
  };
class knapsack
  {
    int arr[100][100];
    int obj_no;
```

```
    int c;
  public:
      knapsack()
      {
       for(int i=0;i<100;i++)
         for(int j=0;j<100;j++)
            arr[i][j]=0;
       }
void print()
{
   cout<<"\n Maximum profit: "<<arr[obj_no][c]
                              <<endl;
 }
void sort(obj obj_arr[],int size)
 {for(int i=1;i<size;i++)
     for(int j=i+1;j<=size;j++)
      if(obj_arr[j].w<obj_arr[i].w)
      {
        obj_tmp=obj_arr[j];
        obj_arr[j]=obj_arr[i];
        obj_arr[i]=tmp;
       }
 }
```

```
void Dkp(obj obj_arr[], int obj_no, int c)
{
   this->obj_no = obj_no;
   this->c = c;
  for(int i=1; i<=obj_no; i++)
   {
    for(int j=0; j<=c; j++)
    {
      if(j - obj_arr[i].w<0)
         arr[i][j] = arr[i-1][j];
      else
        arr[i][j] = max(arr[i-1][j],
                    arr[i-1][j-obj_arr[i].w]
                     +obj_arr[i].p);
    }
   }
  }
};
int main()
{
  int n, k;
  obj arr[101];
  knapsack ob;
  cout << " Total item: ";
  cin >>n
  cout << " knapsack capacity: ";
```

```
cin >> k;
cout << endl;
cout << " profit:  weight: " << endl;
arr[0].p = 0;
arr[0].w = 0;
for(int i = 1; i <= n; i++)
{
   cin >> arr[i].p >> arr[i].w;
    arr[i].f = false;
 }
ob.sort(arr, n);
ob.Dkp(arr, n, k);
ob.prnt();
return 0;
}
```

Input & Output:

```
Total item: 5
knapsack capacity: 11
profit  weight
1 1
6 2
18 5
22 6
28 7
Maximum profit: 40.
```

Output:

Maximum profit: 40.

Algorithm (Fractional knapsack):

```
knapsack(arr[], size, c)
{
    t ← c;
    for( i = 0 to size)
    {
        if (arr[i].w > t)
        {
            arr[i].d := t / arr[i].w;
            break;
        }
        arr[i].d := 1;
        t! = t - arr[i].w;
    }
}
```

Code:

```
#include <bits/stdc++.h>
using namespace std;
struct obj {
        int w;
        int p;
        float d;
};
```

```
void knapsack(obj arr[], int size, int c){
    int t=c;
  for(int i=0;i<size;i++){
      if(arr[i].w>t)
      {
        arr[i].d=(float)t/arr[i].w;
        break;
      }
      arr[i].d=1;
      t=t-arr[i].w;
    }
}
int main()
{
  obj arr[100];
  int n,m;
  float maxp=0.0;
  cout<<"Total item:";
  cin>>n;
  cout<<"knapsack capacity:";
  cin>>m;
  cout<<endl;
  cout<<"profit weight"<<endl;
  for(int i=0;i<n;i++){
   cin>>arr[i].p;
   cin>>arr[i].w;
   arr[i].d=0;
}
```

```
for (int i =0; i<n-1; i++)
for (int j=i+1; j <n; j++){
  if((float)arr[j ]. p/arr[j]. w >(float)
                        (arr[i]. p/ arr[i]. w)
  {
     obj temp=arr[i];
     arr [i]=arr[j];
     arr[j] = temp;
    }
}
knapsack (arr, n, m);
for(int i =0; i<n; i++){
      if(arr[i]. d !=0.0){
       maxp+=arr[i]. p * arr [i]. d;
     }
  }
cout<< endl;
cout <<"Maximum profit: " <<maxp<<endl;
return 0;
}.
```

Input:

Total item: 5
knapsack capacity: 11
profit weight
1 1
6 2
18 5
22 6
28 7

Maximum profit: 42.6667.

Discussion: We can see from the output of 0/1 knapsack and fractional knapsack that, for maximum profit of fractional knapsack is more than 0/1 knapsack because in fractional knapsack, the ratio of weight and $ value has taken.

Problem 4) Sorting in ~~to~~ linear time: Counting Sort

(i) Generate N random integers within range 0 to 10000 in a file named input.txt.

(ii) Implement bubble sort to sort the numbers (from input.txt) and count the line

(iii) Implement counting sort to sort the numbers (from input.txt) and count the time.

(iv) Increase the value of N (upto 100000) and plot the performance

(v) Write your comments on the findings.

Algorithm: (Bubble sort)

```
Bubblesort(arr)
    for all the elements of array arr
        if arr[i] > arr[i+1]
            swap( list[i], list[i+1])
        end if
    end for
    return arr
. end Bubblesort.
```

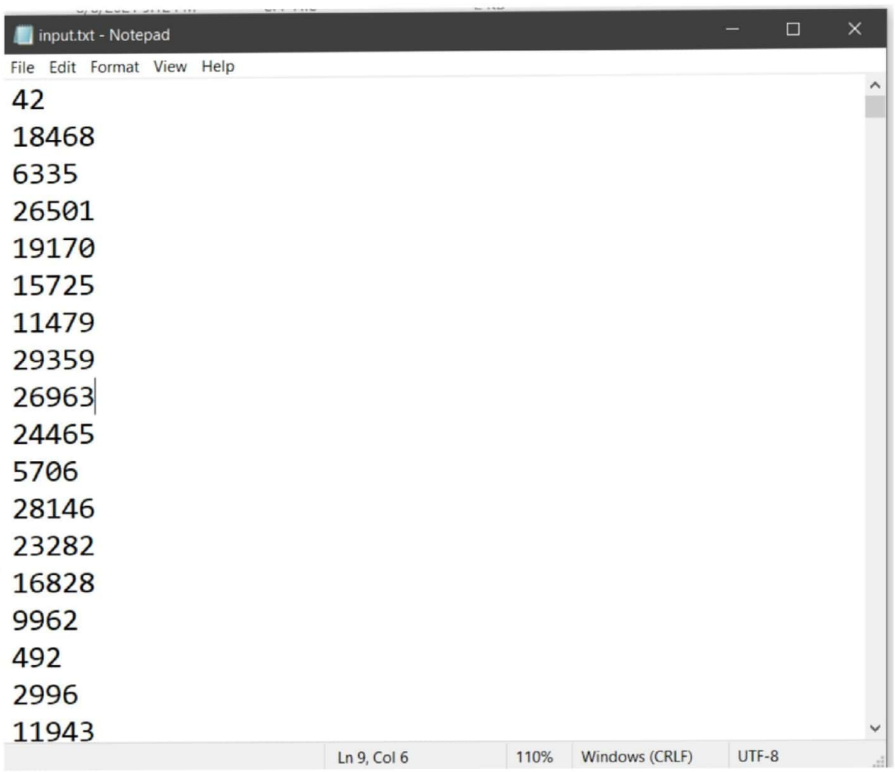
Code:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
 int cnt, n, i, j, temp;
 long long ar[40000], num;
 ofstream input;
 input.open ("input.txt");
 for(cnt=1; cnt<=30000; cnt++)
 {
   n = rand()% 30000+1;
   input<<n<<endl;
 }
 input.close();
 n=0;
 ifstream file;
 file.open ("input.txt");
 if(file)
 {
  while(file >>num)
  {
    ar[n++] = (num);
  }
 }
else
  cout<<"file can't open<<endl;
```

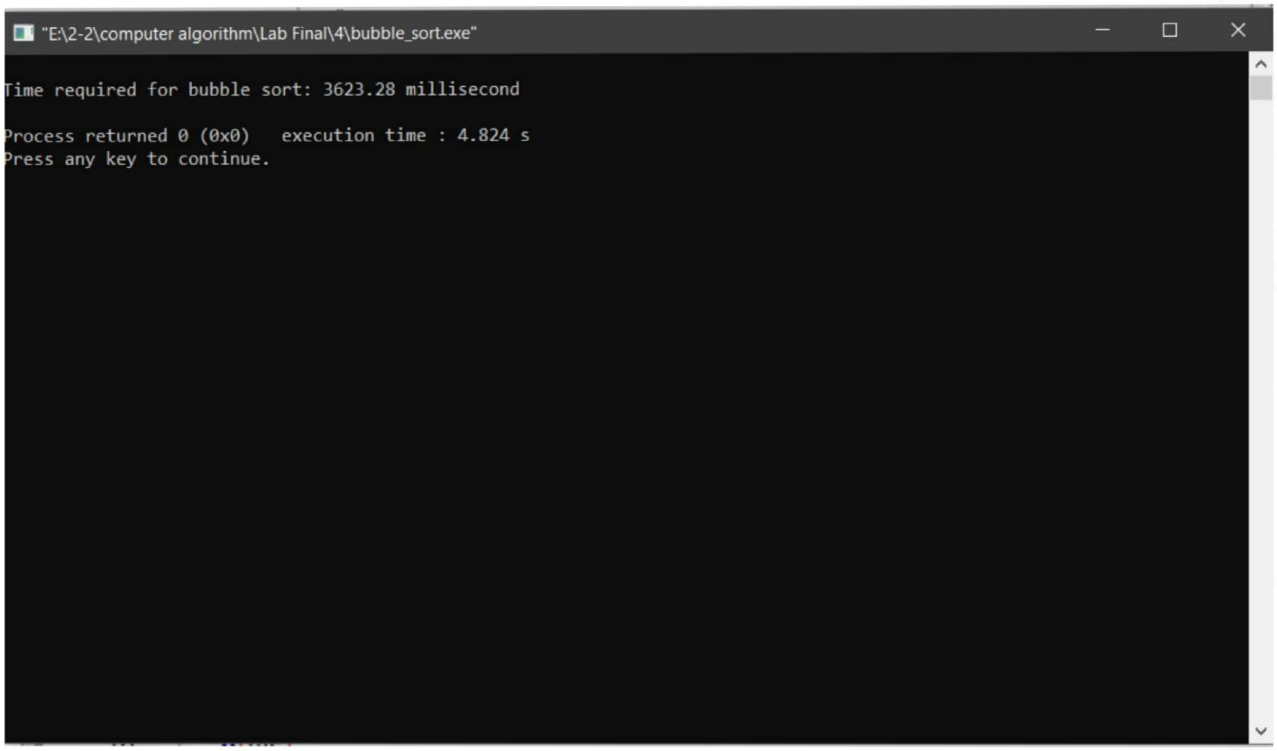
```
auto count1 = chrono:: steady_clock::now();
for(i=0; i<n; i++)
{
  for(j=1; j<n-i; j++)
  {
     if(ar[j]>ar[j+1])
     {
       temp=ar[j];
       ar[j] = ar[j+1];
       ar[j+1] =temp;
     }
  }
}
auto count2 =chrono:: steady_clock::now();
cout<<endl;
double difference = double(chrono::duration_cast
          <chrono:: nanoseconds>(count2-count1).count());

cout<<" Time required for bubble sort: " <<
        double(difference * 1.0/1000000)<<
          " millisecond" <<endl;
```

Input:



Output:



Algorithm: (Counting Sort)

```
CountingSort(arr[], n, b[])
{
  for i=0 to n
      b[arr[i]++;
  for i=1 to maximum
      b[i]: = b[i-1]+b[i];
  for i=n-1 to 0
    arr1[b[arr[i]]-1]: = arr[i];
    b[arr[i]--;
}
```

Code:

```
#include<bits/stdc++.h>
#include<chrono>
#include<ctime>
using namespace std;
int main()
{
  int cnt,n,i;
  long long arr[7000], arr1[7000]={0};num;
  ofstream input;
  input.open("input.txt");
```

```
for(cnt=1; cnt<=6000; cnt++)
{
   n=rand()%3000+1
   input<<n<<endl;
}
input.close();
n=0;
ifstream file;
file.open("input.txt");
if(file)
{
 while (file>>num)
  {
    ar[n]=(num);
    n++;
  }
}
else.
   cout<<"file can't open"<<endl;
auto count1 = chrono::steady_clock:: now();
long long maximum=*max_element(ar,ar+n),
                        b[maximum+5]={0};
for( i=0; i<n; i++)
{
   b[ar[i]]++;
}
```

~~1Title~~: for(i = 1; i <= maximum; i++)
{
b[i] = b[i-1] + b[i];
}

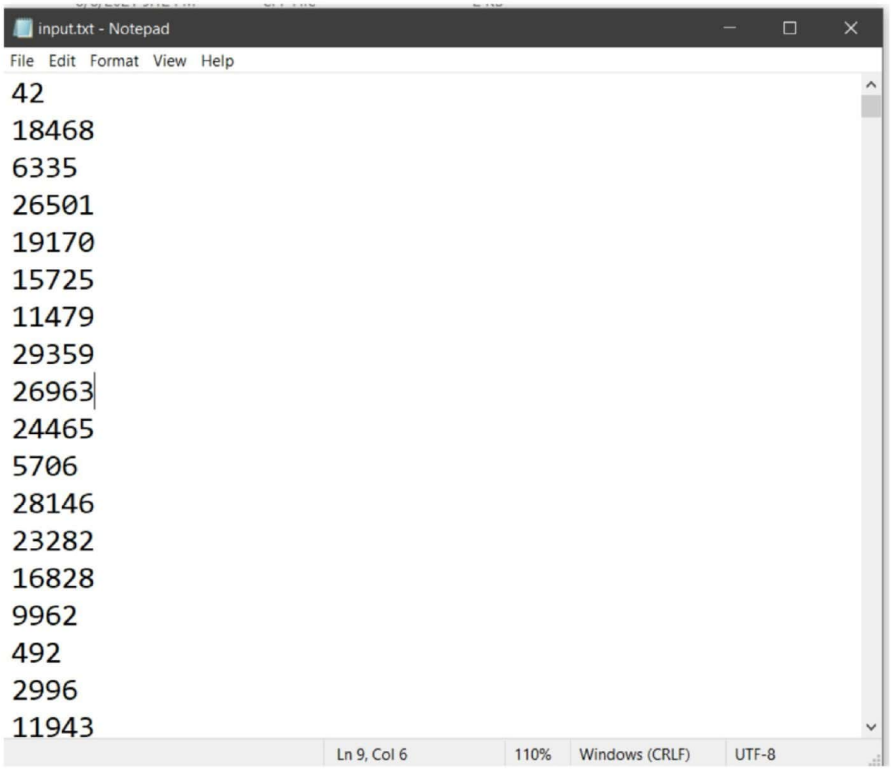
for (i=n-1; i>=0; i--)
{
ar1[b[ar[i]] - 1] = ar[i];
b[ar[i]]--;
}

auto count2 = chrono:: steady_clock::now();
double difference = (double) (chrono:: duration_
cast<chrono:: nanoseconds>
(count2 - count1). count());

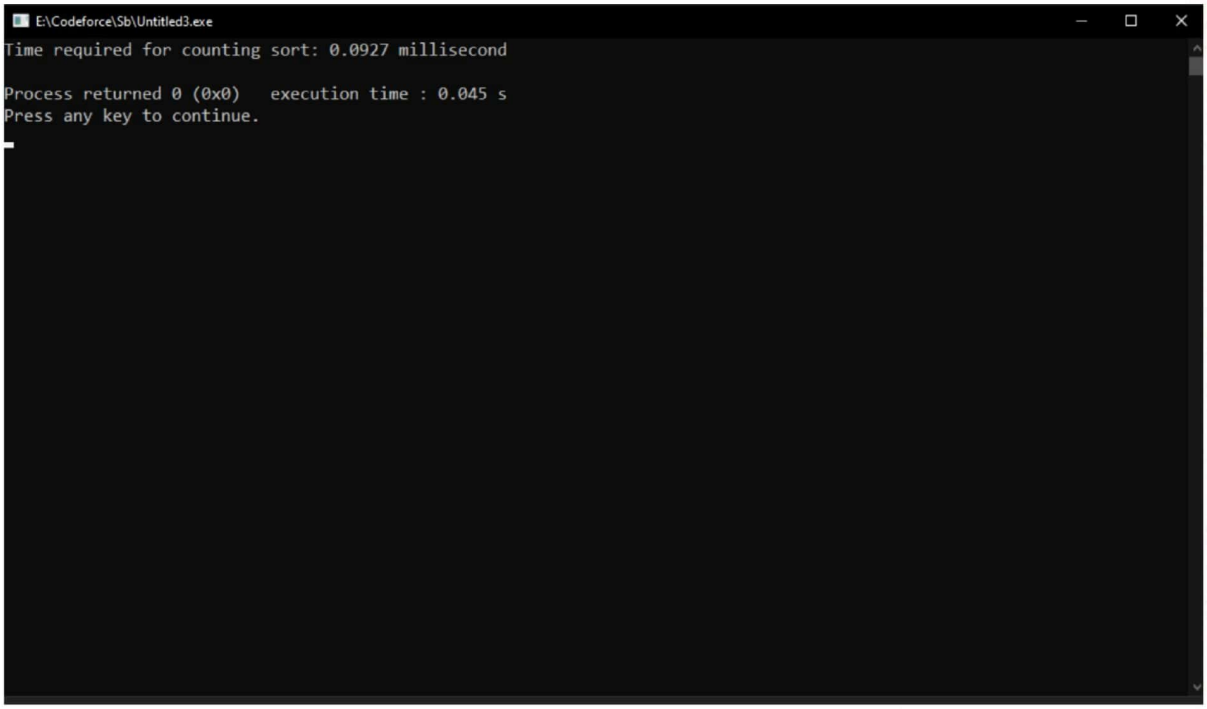
cout<<" Time required for counting sort:"
<<(double) (difference * 1.0/1000000) <<
"millisecond" <<endl;

}

Input:

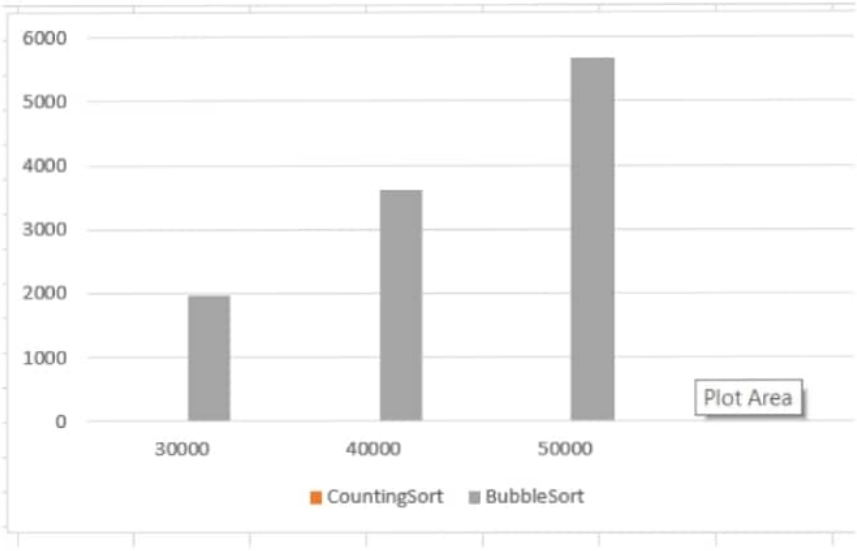


Output:



For various input (up to 100000), the performace graph of bubble sort versus counting sort:

number of inputs	CountingSort	BubbleSort
30000	0.6593	1972.57
40000	0.8875	3623.28
50000	1.0754	5673.95



Discussion: We know that, counting sort is a stable sort with a space complexity of O(k+n). It is a ~~be~~ very efficient algorithm. On the other hand, the complexity of bubble sort algorithm is $O(n^2)$. where n is the number of items. So, it is clear that the counting time of bubble sort is many more than counting sort.

Problem 5) Print all the nodes reachable from a given starting node in a digraph using BFS method.

Algorithm: Bfs(a)

```
{
  for i=1 to n
  {
    if (arra[i] && !r[i])
        q[++w]:=i;
  }
  if (p<=w)
  {
    r[q[p]]:=1;
    Bfs(q[p++]);
  }
}
```

Code:

```
#include<bits/stdc++.h>
using namespace std;
int arr[100][100],q[100],n, w=-1;
int i, j, p=0
bool r[100];
void Bfs(int a)
{
```

```
for(i = 1;i<=n;i++)
{
  if(arr[a][i] && !r[i]){
     q[++w] = i;
    }
 }
  if (p<=w)
  {
   r[q][p] = 1;
   Bfs(q[p++]);
  }
 }
int main(){
  int t;
 cout<<"Enter the number of vertices";
 cin>>n;
 cout<<endl;
 cout<<"Enter graph data in matrix
                          form: "<<endl;
for(int i = 1; i<=n;i++)
{
  for(int j= 1;j<=n;j++)
    {
      cin>>arr[i][j];
    }
 }
cout<<"Enter the starting vertex: ";
```

```
        cin >>t;
        Bfs(t);
        cout<<"The node which are reachable are:"
                                        <<endl;
      for(int i=1; i<=n; i++)
      {
          if(r[i])
            cout<<i<<"\t";
         else
          cout<<"-";
      }
}
```

Input & Output:

Enter the number of vertices: 4

Enter the graph data in matrix form:

```
0  1  1  1
0  0  0  1
0  0  0  0
0  0  1  0
```

Enter the starting vertex: 1

The node which are reachable are:

```
2  3  4
```

Problem 6) Implement N Queen's problem using Back Tracking. N should be read from a file queen.txt and N should not be <4. Determine at least two solutions and print the number of iteration needed for each solution

Algorithm:

```
n_queen()
{
for i=0 to n
bool var = false;
r.pushback(i);
for j=k to j>0
    if(var) break;
    if(arr[j][i])
      var:=true;
for j=k, l=i to j>=0
    if(var) break;
    if(arr[j][l])
      var:=true;
for j=k, l=i. to j>=0
    if(var) break;
    if(arr[j][l])
      var:=true;
}
```

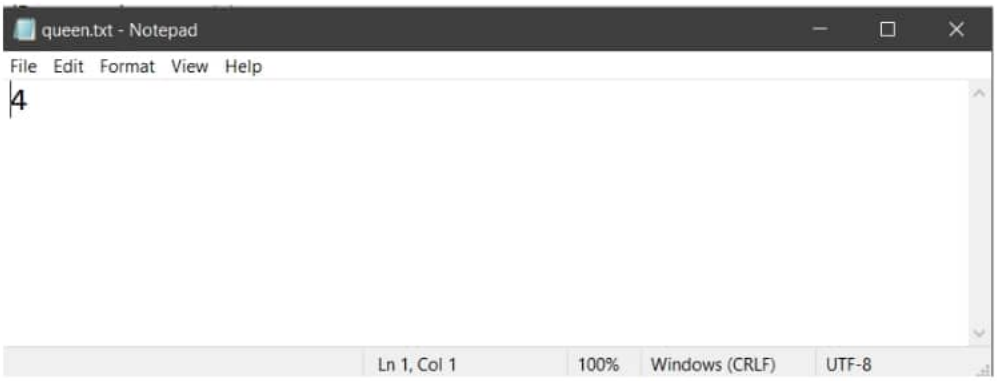
Code:

```
#include<bits/stdc++.h>
using namespace std;
int arr[1000][1000], k=0, n, iteration=0, cnt;
vector<int>r;
void n_queen(){
  for(int i=0;i<n;i++)
  {
      iteration++;
      bool var=false;
      r.push_back(i);
      for(int j=k;j>=0;j--)
      {
        if(var)
          break;
        if(arr[j][i])
          var=true;
      }
      for(int j=k,l=i;j>=0;j--,l--)
      {
        if(var)
          break;
        if(arr[j][l])
          var=true;
      }
      for(int j=k,l=i; j>=0;j--,l++)
      {
        if(var)
          break;
```

```
        if(arr[j][l])
            var=true;
    }
    arr[k][i]=1;
    if(var)
    {
      arr[k][i]=0;
      r.pop_back();
      continue;
    }
    if(r.size()==n)
    {
      cout<<"iteration:"<<iteration<<endl;
      cnt++;
      cout<<"solution: #"<<cnt<<endl;
      for(int x=0;x<n;x++)
      {
        for(int y=0;y<n;y++)
        {
            if(arr[x][y])
                cout<<"Q ";
            else
                cout<<"* ";
        }
       cout<<endl;
    }
    r.pop_back();
    arr[k][i]=0;
    return;
```

```
    }
    k++;
    n_queen();
    n.pop_back();
    k--;
    arr[k][i]=0;
  }
  return;
}
```

Input:



Output: