VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Design and Analysis of Algorithms Lab Manual					
Course Code:	Year and Semester: III-I	L	T	P	C
Prerequisites: Prior knowledge of programming language(s) and basic Data Structures and Algorithms		0	0	3	1.5

Course Objectives:

1. To learn fundamental algorithmic problems.
2. To understand methods of designing and analysing algorithms.
3. To know various designing paradigms of algorithms for solving real world problems.

Course Outcomes:

At the end of the course student will be able to:

CO1: Identify and apply the suitable algorithm for the given problem.

CO2: Design and implement efficient algorithms for a specified application.

List of experiments:

1. Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.
2. Write a program to find the optimal profit of a Knapsack using Greedy method.
3. Write a program for Optimal Merge Patterns problem using Greedy Method.
4. Write a program for Single Source Shortest Path for General Weights using Dynamic Programming.
5. Write a program to find all pair shortest path from any node to any other node within a graph.
6. Write a program to find the non-attacking positions of Queens in a given chess board using backtracking.
7. Find a subset of a given set S = $\{S_1, S_2, ..., S_n\}$ of n positive integers, whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1, 2, 6} and {1, 8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.
8. Write a program to color the nodes in a given graph such that no two adjacent can have the same color using backtracking.

9. Design and implement to find all Hamiltonian Cycles in a connected undirected Graph G of n vertices using Backtracking principle.

Eperiment 1:

Write a program to find the maximum and minimum element from the collection of elements using divide and conquer technique.

Procedure:

A divide-and-conquer algorithm for this problem would proceed as follows:

Let P = (n,a[i],….,a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list a[i],….,a[j] and we are interested in finding the maximum and minimum of this list.

Let small(P) be true when n ≤ 2.

If n = 1, the maximum and minimum are a[i].

If n = 2, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances.

For example, we might divide P into the two instances

P1 = (n/2,a[1],….,a[n/2]) and P2 = (n - n/2,a[n/2 + 1],….,a[n]).

After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

We can combine the Solutions for P1 and P2 to obtain the solution for P as follows.

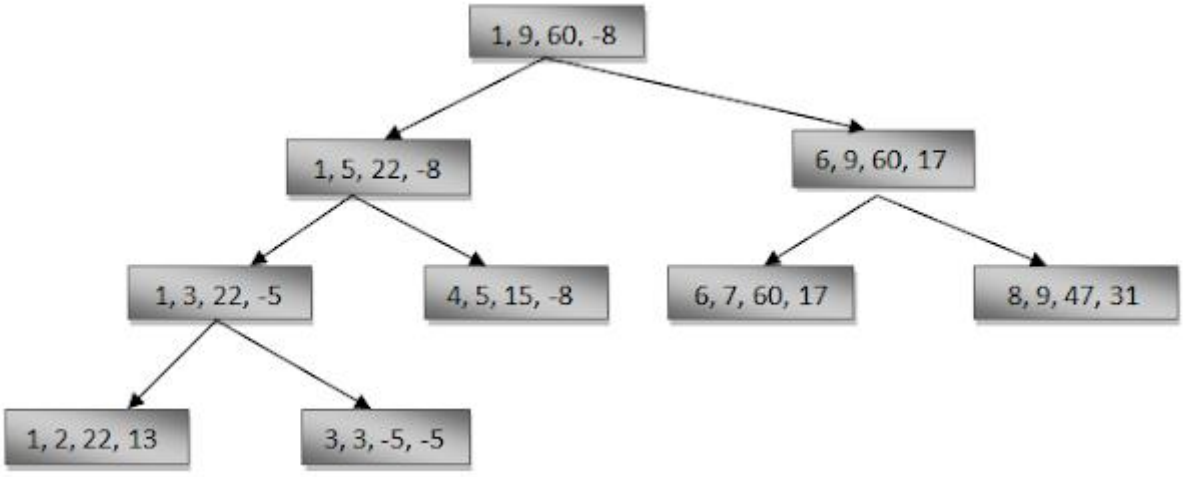
If MAX(P) and MIN(P) are the maximum and minimum of the elements of P, then MAX(P) is the larger of MAX(P1) and MAX(P2) also MIN(P) is the smaller of MIN(P1) and MIN(P2).

MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i),a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately. For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When the maxima and minima of these sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

Suppose we simulate MaxMin on the following nine elements:

Ex: a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. On the array a[ ] above, the following tree is produced.



We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call).

Algorithm:

```
Alogrithm MaxMin(i, j, max, min)
// a[1:n] is a global array. Parameters i and j are integers,
  // 1≤i≤j≤n. The effect is to set max and min to the largest and     // smallest values
in a[i:j].
{
        if (i=j) then max := min := a[i]; //Small(P)
        else if (i=j-1) then // Another case of Small(P)
                {
                        if (a[i] < a[j]) then max := a[j]; min := a[i];
                        else max := a[i]; min := a[j];
                }
        else
        {
                 // if P is not small, divide P into sub-problems.
                 // Find where to split the set.
                 mid := ( i + j )/2;
                 // Solve the sub-problems.
                 MaxMin( i, mid, max, min );
                 MaxMin( mid+1, j, max1, min1 );
                 // Combine the solutions.
                 if (max < max1) then max := max1;
                 if (min > min1) then min := min1;
        }
```

```
}
```

Sample implementation:

```
#include<stdio.h>
int max,min;
int a[100];
void maxmin(int i,int j)
{
 int max1,min1,mid;
 if(i==j)
 {
  max=min=a[i];
 }
 else
 {
  if(i==j-1)
  {
   if(a[i]<a[j])
   {
    max=a[j];
    min=a[i];
   }
   else
   {
    max= a[i];
    min=a[j];
```

```
   }
  }
  else
  {
   mid=(i+j)/2;
   maxmin(i,mid);
   max1=max; min1=min;
   maxmin(mid+1,j);
   if(max<max1)
    max=max1;
   if(min>min1)
    min=min1;
  }
 }
}
void main ()
{
 int i,s;
 printf("\n Enter size of input....");
 scanf("%d",&s);
 printf("Enter the numbers : \n");
 for(i=1;i<=s;i++)
  scanf("%d",&a[i]);
 max =-1;
```

```
 min =9999;

 maxmin(1,s);

 printf ("maximum element is: %d\n",max);

 printf ("minimum element is: %d\n",min);

 }
```

Output:

```
 Enter size of input....5
Enter the numbers :
12
56
2
45
67
maximum element is: 67
minimum element is: 2
```

Experiment 2:

Write a program to find the optimal profit of a Knapsack using Greedy method.

Procedure:

Greedy Method:

The greedy method is the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

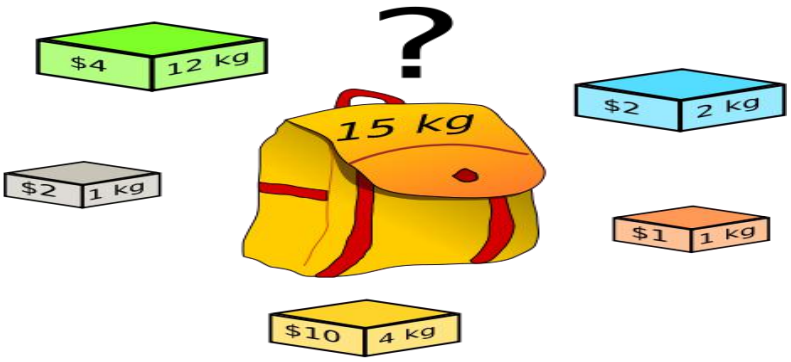
Feasible solution:- Most problems have n inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

Optimal solution: To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Knapsack problem

The knapsack problem or rucksack (bag) problem is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible



There are two versions of the problems

1. 0/1 knapsack problem
2. Fractional Knapsack problem
    a. Bounded Knapsack problem.
    b. Unbounded Knapsack problem.

Solutions to knapsack problems

Brute-force approach:-Solve the problem with a straight forward algorithm

Greedy Algorithm:- Keep taking most valuable items until maximum weight is reached or taking the largest value of each item by calculating v_i=$value_i$/$Size_i$

Fractional knapsack problem is solved using greedy method as follows

1. For each item, compute its value / weight ratio.
2. Arrange all the items in decreasing order of their value / weight ratio.
3. Start putting the items into the knapsack beginning from the item with the highest ratio.

   Put as many items as you can into the knapsack.

Example

Consider the following instance of the knapsack problem: n = 3, m = 20, (p1, p2, p3) = (25, 24, 15) and (w1, w2, w3) = (18, 15, 10).

1. First, we try to fill the knapsack by selecting the objects in some order:

x1	x2	x3	wi xi	pi xi
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4 = 24.25

2. Select the object with the maximum profit first (p = 25). So, x1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x2 = 2/15

x1	x2	x3	wi xi	pi xi
1	2/15	0	18 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 2/15 = 28.2

3. Considering the objects in the order of non-decreasing weights wi.

x1	x2	x3	wi xi	pi xi
0	2/3	1	15 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 1 = 31

4. Considered the objects in the order of the ratio pi / wi .

p1/w1	**p2/w2**	**p3/w3**
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio pi / xi. Select the object with the maximum pi / xi ratio, so, x2 = 1 and profit earned is 24. Now, only 5 units of space is left, select the object with next largest pi / xi ratio, so x3 = ½ and the profit earned is 7.5.

x1	x2	x3	wi xi	pi xi
0	1	1/2	15 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

optimal solution.

Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
for i := 1 to n do x[i] := 0.0 // initialize x U := m;
for i := 1 to n do
{
if (w(i) > U) then break;
x [i] := 1.0; U := U – w[i];
}
if (i < n) then x[i] := U / w[i];
}
```

Sample Implementation:

```
# include<stdio.h>
void knapsack(int n,float weight[],float profit[],float capacity)
{
   float x[20],pr=0;
   int i,j,u;
   u=capacity;
   for(i=0;i<n;i++)
         x[i]=0.0;
   for(i=0;i<n;i++)
   {
         if(weight[i]>u)
              break;
         else
          {
              x[i]=1.0;
              pr=pr+profit[i];
              u=u-weight[i];
          }
}
```

```
if(i<n)
      x[i]=u/weight[i];
pr=pr+(x[i]*profit[i]);
  printf("\n Result array x is...");
for(i=0;i<n;i++)
  printf("%f\t",x[i]);
printf("\n Profit is..... %f",pr);
}

void main()
{
      float weight[20],profit[20],capacity;
      int n,i,j;
      float ratio[20],t;
      printf("\n enter the capacityacity of knapsack...");
      scanf("%f",&capacity);
      printf("\n Toal no of objects...");
      scanf("%d",&n);
      printf("\n Enter the weights and profits of each object...");
      for (i=0;i<n;i++)
      {
         scanf("%f %f",&weight[i],&profit[i]);
      }
  for (i=0;i<n;i++)
    {
       ratio[i]=profit[i]/weight[i];
      }
for (i=0;i<n;i++)
{
      for (j=i + 1;j<n;j++)
        {
           if (ratio[i]<ratio[j])
             {
                t=ratio[j];
                ratio[j]=ratio[i];
                ratio[i]=t;
                t=weight[j];
                weight[j]=weight[i];
                weight[i]=t;
                t=profit[j];
```

```
            profit[j]=profit[i];
            profit[i]=t;
        }
    }
}
knapsack(n,weight,profit,capacity);
}
```

Output:

```
 enter the capacityacity of knapsack...20

 Toal no of objects...3

 Enter the weights and profits of each object...18
25
15 24
10 15

 Result array x is...1.000000   0.500000       0.000000
 Profit is..... 31.500000
```