6. Implement in Java, the 0/1 Knapsack problem using
(a) Dynamic Programming method
(b) Greedy method.

a)
```
import java.util.Scanner;
 public class KnapSackDPDemo
{
 public static void main(String [] args)
 {
   int i;
   int n; // No of items
   int W; // Capacity of the knapSack
   int wt[] = new int[10]; // Weights of 'n' items
   int val[] = new int[10]; // Values of 'n' items

   Scanner in = new Scanner(System.in);

   System.out.println("Enter the no of items");
   n = in.nextInt();

   System.out.println("Enter the weight of the items");
   for(i=1;i<=n;i++)
   {
    wt[i] = in.nextInt();
   }

   System.out.println("Enter the value of the items");
   for(i=1;i<=n;i++)
   {
    val[i] = in.nextInt();
   }

   System.out.println("Enter the capacity of the knapsack");
   W = in.nextInt();

   System.out.println("The maximum value in knapsack of capacity " + W + " is:
  "+knapSack(W, wt, val, n));
}
// Returns the maximum value that can be put in a knapsack of capacity W
public  static int knapSack(int W, int wt[], int val[], int n)
{
        int i, j;
        int v[][] = new int[n+1][W+1];

        // Build table v[][] in bottom up manner
```

```
        for (i = 0; i <= n; i++)
        {
           // j is the temporary capacity of knapsack during table building
            for (j = 0; j <= W; j++)
            {
                if (i==0 || j==0)
                  v[i][j] = 0;
                else if (j-wt[i]<0)
                    v[i][j] = v[i-1][j];
                else
                    v[i][j] = max(v[i-1][j],val[i] + v[i-1][j-wt[i]]);
             }
        }

        return v[n][W];
  }
   public static int max(int a, int b)
   {
     return (a > b)? a : b;
   }
}
```

b)

```
import java.util.Scanner;
   public class KnapSackGreedyDemo
   {
      public static void main(String [] args)
      {
         int nItems;
         Scanner in = new Scanner(System.in);
         System.out.println("Enter the number of items: ");
         nItems = in.nextInt();
         double W; /* capacity of the knapsack */
         double weight[] = new double[nItems]; // Weights of 'n' items
         double value[] = new double[nItems]; // Values of 'n' items
         double ratio[] = new double[nItems]; // ratio or density of 'n' items

         System.out.println("Enter the weights of the item");
         for (int i = 0; i<nItems; ++i)
         weight[i] = in.nextDouble();
          System.out.println("Enter the  values/profits of the item");
         for (int i = 0; i<nItems; ++i)
         value[i] = in.nextDouble();
          for (int i = 0; i<nItems; ++i)
         ratio[i] = value[i] / weight[i];
          System.out.println("Enter the Capacity of the knapsack: ");
          W = in.nextDouble();
```

```
        System.out.println("\n The maximum value in a knapsack of capacity " + W+ " is: "+
        computeMaxValue(W,weight,value,ratio));
    }

    public static double computeMaxValue(double W,double weight[],double value[],double
    ratio[])
   {
      double cW = 0; //current weight
      double cV = 0; //current value
      System.out.print("\n Items considered are: ");
      while (cW<W)
      {
       int item = getNextItem(weight,value,ratio);   // getnext highest ratio
       if (item == -1)
       {
        //No items left
         break;
       }
       System.out.print((item+1)+" ");
       if (cW + weight[item] <= W)
       {
        cW += weight[item];
        cV += value[item];
       //mark as used for the getNext() (ratio) function
        ratio[item] = 0;
      }
      else
      {
         cV += (ratio[item] * (W - cW)); // Break the value and add
         cW += (W - cW);
         break;  //the knapsack is full
      }
}
 return cV;
}
   //Method to get the next highest ratio
   public static int getNextItem(double weight[],double value[],double ratio[])
   {
    double highest = 0;
    int index = -1;
    for (int i = 0; i<value.length; ++i)
    {
      if (ratio[i] >highest)
      {
       highest = ratio[i];
```

```
            index = i;
          }
        }
       return index;
      }
}
```