design analysis and algorithm in Java

LAB

KANDUKURI JASWANTH

Table of Contents

Task-1

i) Linear search in java **1**

ii) Binary search in java 6

Task-2: write a java program to implement the merge sort...12

Task-3: write a java program to implement the quick sort...19

Task-4: Write a java program to implement greedy algorithm for job sequencing with deadlines...........25

Task-5: Implement in Java, the 0/1 Knapsack problem using greedy approach...34

Task-6: Write a java program to implement Dijkstra's algorithm for the Single source shortest path problem. 43

Task-7: Write a java program that implements Prim's algorithm to generate minimum cost spanning tree. 47

Task-8: Write a java program that implements Kruskal's algorithm to generate minimum cost spanning tree..52

Task-9: Implement All-Pairs Shortest Paths problem using Floyd's algorithm 59

Task-10: Write a java program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem. 63

Task- 11: Implement Travelling Sales Person problem using Dynamic programming. 66

Task-12: Write a java program to implement the backtracking algorithm for the sum of subsets problem of a given set S = {Sl, S2,.....,Sn} 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}. Display a suitable message, if the given problem instance doesn't have a solution. 77

Task-13: Write a java program to implement the backtracking algorithm for the Hamiltonian Circuits problem..80

TASK-1

Write a java program to implement a linear search.

Aim: To write a java program to implement a linear search.

Description:

Linear search is a very simple search algorithm. In this type of search, a sequential search is done for all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
In other words, if you have N items in your collection, the worst-case scenario to find a topic is N iterations. In Big O Notation it is O(N). The speed of search grows linearly with the number of items within your collection. Unlike the Binary Search algorithm, Linear searches don't require the collection to be sorted.

Algorithm:

```
Algorithm l (A, n, x)
{
    key=0;
     for (i=0; i<n; i++)
       {
          if(key==a[i]) when
          Element is found
          Key=key+1;
        }
          if(key==0)
            Element is not found
 }
```

Eg: Consider the array elements 1, 2, 5, 7, 8, 10, 12

i) Find the searching element 7
ii) Find the searching element 6

Sol:

i)

k	a[i]	k? a[i]
7	1	7≠1
7	2	7≠2
7	5	7≠5
7	7	7=7

Element is found

ii)

k	a[i]	k? a[i]
6	1	6≠1
6	2	6≠2
6	5	6≠5
6	7	6≠7
6	8	6≠8
6	10	6≠10
6	12	6≠12

Element is not found

Program:

```
import java.util.Scanner;

public class LinearSearchExample

{

  public static void main(String args[])

  {

    int counter, num, item, array[];

    //To capture user input

    Scanner input = new Scanner(System.in);

    System.out.println("Enter number of elements:");

    num = input.nextInt();

    //Creating array to store the all the numbers

    array = new int[num];

    System.out.println("Enter " + num + " integers");

    //Loop to store each numbers in array
    for (counter = 0; counter < num; counter++)
      array[counter] = input.nextInt();


    System.out.println("Enter the search value:");
    item = input.nextInt();


    for (counter = 0; counter < num; counter++)
    {
       if (array[counter] == item)
       {
         System.out.println(item+" is present at location "+(counter+1));
         /*Item is found so to stop the search and to come out of the
          * loop use break statement.*/
         break;
       }
    }
    if (counter == num)
      System.out.println(item + " doesn't exist in array.");
  }
}
```

Output:

```
Enter number of elements:

7

Enter 7 integers

1 2 5 7 8 10 12

Enter the search value:

7

7 is present at location 4
```

Result: Thus , in the above program successfully executed without errors using linear search.

2) Write a java program to implement binary search.

Aim: To write a java program to implement binary search.

Description:

A binary search in Java is a technique that is used to search for a targeted value or key in a collection. It is a technique that uses the "divide and conquers" technique to search for a key.

The collection on which Binary search is to be applied to search for a key needs to be sorted in ascending order.

Usually, most of the programming languages support Linear search, Binary search, and Hash techniques that are used to search for data in the collection.

Binary Search In Java

Linear search is a basic technique. In this technique, the array is traversed sequentially and each element is compared to the key until the key is found or the end of the array is reached.

Linear search is used rarely in practical applications. Binary search is the most frequently used technique as it is much faster than a linear search.

Java provides three ways to perform a binary search:

1. Using the iterative approach
2. Using a recursive approach
3. Using Arrays.binarySearch () method.

Algorithm:

```
Algorithm BS (A, n, x)
{
    low= l;
    high= n;
    while(low<high)
    {
        mid= low + high/2;
        if(x<A[mid]) then
            high=  mid-1;
        else
            if(x>A[mid]) then
                low= mid+1;
            else
                return mid;
    }
    return 0;
}
```

Eg: consider the array elements 1,2,5,6,7,10,24,56,84,100,115,120,131,150.

i) Find searching element x=150.

ii) Find searching element x=9.

Sol:

i)

low	high	mid	A[mid]	x? A[mid]
1	14	7	24	150>24
8	14	11	115	150>115
12	14	13	131	150>131
14	14	14	150	150=150

Element is found

ii)

low	high	mid	A[mid]	x? A[mid]
1	14	7	24	9<24
1	6	3	5	9>5
4	6	5	7	9>7
6	6	6	10	9<10
6	5	-	-	-

Element is not found.

Time complexity of binary search:

Let the recurrence relation

T(n)= T(n/2)+1 ----1

Put n=n/2 in equation 1

T(n/2)=t(n/4)+1---2

Substitute T(n/2) in equation 1 we get

$$T(n) = t\left(\frac{n}{2^2}\right) + 1 + 1$$

$$T(n) = t\left(\frac{n}{2^2}\right) + 2$$

$T(n) = t\left(\frac{n}{2^k}\right) + 2$----3

Put n= 2^k

k= $\log_2$n in equation ---3

$T(n) = T(2^k/2^k) + \log_2 n$

$= T(1) + \log_2 n$ [since T(1)=1 for simple solution]

$O(\log_2 n)$

$T(n) = O(\log n)$

Program:

```
import java.util.Scanner;
            public class binary search
              {
                 public  static void main(String args[])
                 {
                    Scanner scan=  new scanner(System.in);
                     System.out.println("Enter the number of elements");
                     int n=s.nextInt()  low=0 high=n-1, mid, key=0;
                       a=new int[n];
                       for (int i=0; i<n; i++)
                        {
                             a[i]= s.nextInt();
                       }
                            System.out.println("key element");
                          int  k= s.nextInt();
                          while(low<=high)
                            {
                              mid= (low+high)/2;
                             if(k<a[mid])
                              high=mid-1;
                            else
                                if(k> a[mid])
                                 low=mid+1;
                                 else
                                   {
                                       System.out.println("The element" +a[mid]+ "is
                                                              found at postion"+mid);
                                             break;
                               }
                           }
                                if(key==0)
                                  System.out.println("The element is not found");
                        }
                 }
```

Output:

```
         Enter the number of elements:
             4
          Enter the elements:
          1  2  3    5
```

```
Enter the key element:
    4
 The element is not found.
```

Result: Thus , in the above program successfully executed without errors using binary search.

`

Task-2

1) Write a java program to implement the merge sort.

Aim: To write a java program to implement the merge sort algorithm by using divide and conquer approach.

Description:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. It divides the given list into two halves, calls itself for the two halves, and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs are merged into the four-element lists, and so on until we get the sorted list.

Here mainly concentrate on three steps:

i) Divide
ii) Conque
iii) Combine

i)*Divide*:

Divide the array into sub-arrays that is s1 and s2 with

$$mid = \frac{low + high}{2}$$

ii)*Conque*:

Recurrisally sort s1 and s2 sub-arrays

iii*)Combine*:

Combine is sub-arrays s1 and s2

Algorithm:

```
mergesort(low, high)
{
   if(n=1)then
   return;
    else
       {
          if(low<high) then
             $mid = \frac{low+high}{2}$ ;
               mergesort(low, high);
               mergesort(mid+1, high);
                 combine(low, mid, high);
       }
}
```

Time complexity of merge sort:

The time following the merge operation is proportional to n. Then the computing time for merge sort is described in recurrence relation.

$$T(n) = \begin{cases} a & n=1 \\ 2T(n/2)+cn & n>1 \end{cases}$$

Here a is constant

Now

Let T(n)=2T(n/2)+cn-1

Put n=$\frac{n}{2}$ in equation 1 we get

T($\frac{n}{2}$) =2T($\frac{n}{2^2}$)+c($\frac{n}{2}$)

Sub T($\frac{n}{2}$) in equation 2 we get

T(n)= 2[2T($\frac{n}{2^2}$)+ c($\frac{n}{2}$)]+cn

=2^2T($\frac{n}{2^2}$)+2c($\frac{n}{2}$)]+cn

T(n)=2^2T($\frac{n}{2^2}$)+2cn

: :

: :

T(n)= 2^3T($\frac{n}{2^3}$)+3cn

: :

: :

$\boxed{\text{T(n)= } 2^k\text{T}(\frac{n}{2^k})\text{+kcn}}$

Now put n=2^k i.e k= log_2^n

T(n)=nT($\frac{2^k}{2^k}$)+ log_2^n (cn)

=n+T(1)+ log_2^n (cn)

= n+cn(log_2^n)

$\boxed{\text{T(n)=n[1+}(log_2^n)]}$

O(n[1+clog_2^n])

O(n log n)

Time complexity of merge sort in =O(n log n)

Program:

```
public class Merge
{
void merge(int a[], int beg, int mid, int end)
{
   int i, j, k;
   int n1 = mid - beg + 1;
   int n2 = end - mid;
      int LeftArray[] = new int[n1];
      int RightArray[] = new int[n2];
   for (i = 0; i < n1; i++)
   LeftArray[i] = a[beg + i];
   for (j = 0; j < n2; j++)
   RightArray[j] = a[mid + 1 + j];

   i = 0; /* initial index of first sub-array */
   j = 0; /* initial index of second sub-array */
   k = beg;  /* initial index of merged sub-array */
   while (i < n1 && j < n2)
```

```
{
    if(LeftArray[i] <= RightArray[j])
    {
        a[k] = LeftArray[i];
        i++;
    }
    else
    {
        a[k] = RightArray[j];
        j++;
    }
    k++;
}
while (i<n1)
{
    a[k] = LeftArray[i];
    i++;
    k++;
}

while (j<n2)
{
        a[k] = RightArray[j];

        j++;

        k++;

    }

}

void mergeSort(int a[], int beg, int end)

{

    if (beg < end)

    {

        int mid = (beg + end) / 2;

        mergeSort(a, beg, mid);

        mergeSort(a, mid + 1, end);

        merge(a, beg, mid, end);

    }

}

void printArray(int a[], int n)

{

    int i;

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

        System.out.print(a[i] + " ");

}
```

```
public static void main(String args[])

{

   int a[] = { 11, 30, 24, 7, 31, 16, 39, 41 };

   int n = a.length;

   Merge m1 = new Merge();

   System.out.println("\nBefore sorting array elements are - ");

   m1.printArray(a, n);

   m1.mergeSort(a, 0, n - 1);

   System.out.println("\nAfter sorting array elements are - ");

   m1.printArray(a, n);

   System.out.println("");

}


  }
```

Output:

```
Before sorting array elements are -

11 30 24 7 31 16 39 41

After sorting array elements are -

7 11 16 24 30 31 39 41
```

Result: Thus , in the above program successfully executed without errors using merge sort.

Task-3

Write a java program to implement Quick sort.

Aim: To write a java program to implement the quick sort

Description:

Sorting is a way of arranging items systematically. Quick sort is the widely used sorting algorithm that makes **n log n** comparisons in an average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquers approach. Divide and conquer is a technique of breaking down the algorithms into sub-problems, then solving the sub-problems, and combining the results back together to solve the original problem.

Here mainly concentrate on three steps:

i) Divide
ii) Conque
iii) Combine

i)Divide: Re-arrange the elements and slip the arrays into two sub-arrays each element in the left sub-array is less than or equal to the middle element and each element in the right sub-array is greater than the mid element.

ii) Conquer: Recursively, sort two sub-arrays with quick sort.

iv) **Combine**: Combine the already sorted array.

Algorithm:

```
Quicksort(a,low,high)
{
  pivot=a[low];
  lb=low;
  ub=high;
  while(lb$\leq$ $ub$) do
   {
     while(lb≤pivot)
     lb=lb+1;
    while(a[ub]≥pivot)
     ub:=ub-1;
     if(lb<ub) then
    swap(a,low,high)
```

```
        }
          a[low]=a[ub];
           a[ub]:=pivot;
         return ub;
      }
```

Time complexity:

1) Worst case

$$T(n)=\begin{cases} aT & n=1 \\ T(n-1)+n & n>1 \end{cases}$$

Now T(n)=T(n-1)+n-------1

Put N=n-1 in equation 1 we get

T(n-1)=T(n-1-1)+n-1

T(n-1)=(n-2)+n-1

Sub T(n-1) in equation1

T(n)=T(n-2)+(n-1)+n

: :

: :

T(n)=T(n-3)+T(n-2)+(n-1)+n

:

:

:

T(n)=$\frac{n(n+1)}{2}$

=$\frac{n^2+n}{2}$

$\boxed{T(n)= \frac{n^2+n}{2}}$

O($\frac{n^2+n}{2}$)

$\boxed{T(n)=O(n^2)}$

Best case and average case:

Now the recurrence relation is

$$T(n) = \begin{cases} a & n=0 \\ 2^{T}\left(\frac{n}{2}\right) + n & n>1 \end{cases}$$

By using master method

Compare with general term:

T(n)=aT $\left(\frac{n}{b}\right) + f(n)$

a=2

b=2

d=power of n=1

a=b^d

2=2^1 (condition is True)

$\Theta(n^d \log n)$

$\Theta(n^1 \log n)$

$\Theta(n\log n)$

Program:

```
public class Quick
{
   int partition (int a[], int start, int end)
  {
    int pivot = a[end]; // pivot element
    int i = (start - 1);


    for (int j = start; j <= end - 1; j++)
    {
        if (a[j] < pivot)
        {
            i++; // increment index of smaller element
            int t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
     int t = a[i+1];
     a[i+1] = a[end];
     a[end] = t;
     return (i + 1);
}
  void quick(int a[], int start, int end)
  {
     if (start < end)
     {
        int p = partition(a, start, end);
        quick(a, start, p - 1);
        quick(a, p + 1, end);
     }
  }
   void printArr(int a[], int n)
    {
      int i;
        for (i = 0; i < n; i++)
        System.out.print(a[i] + " ");
    }
      public static void main(String[] args)
      {
        int a[] = { 13, 18, 27, 2, 19, 25 };
        int n = a.length;
         System.out.println("\nBefore sorting array elements are - ");
          Quick q1 = new Quick();
         q1.printArr(a, n);
           q1.quick(a, 0, n - 1);
          System.out.println("\nAfter sorting array elements are - ");
            q1.printArr(a, n);
          System.out.println();
    }
}
```

Output:

```
Before sorting array elements are -

13 18 27 2 19 25

After sorting array elements are -

2 13 18 19 25 27
```

Result: Thus , in the above program successfully executed without errors using quick sort.

Task-4:

Write a java program to implement greedy algorithm for job sequencing with deadlines

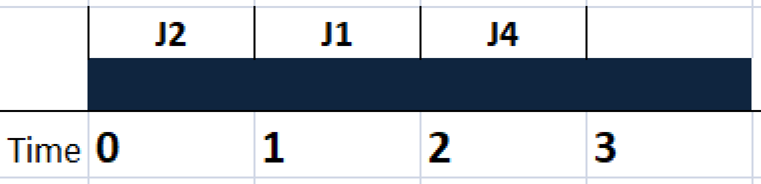
Aim: To Write a java program to implement greedy algorithm for job sequencing with deadlines.

Description:

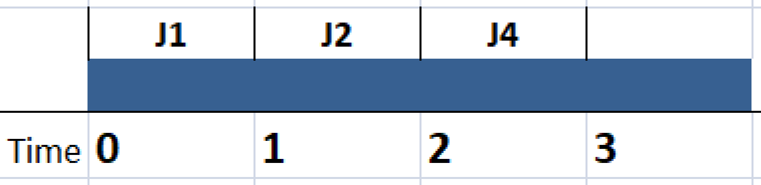
Concept of Job Sequencing Problem in Java

Job-Sequence problem consists of certain finite jobs associated with their deadline and profits.
Our goal is to earn maximum profit.
We will assume that each job takes 1 unit time to traverse, So the minimum deadline for each job is 1.
We will earn profit from a particular job only when it is completed before or on time.

Jobs	J1	J2	J3	j4	j5
Profit	20	15	10	5	1
Deadlines	2	2	1	3	3



OR



Step 1:Look for the maximum profit (J1:20) and it is ready to wait for 2 units of time. 0->1->2(put J1 in place of 1->2),Insertion is done from back.
Step 2:Look for the second maximum profit(J2:15) and it is also ready to wait for 2 units of time.0->1->2(Put J2 in place of 0->1)
J3 can't be adjusted because it only can wait for the time (0->1), and 0-> is already filled with J1.
Step 3:Repeat step 1 &2

Final Sequence : J2—> J1—>j4
or, J1—>J2—>J4
Total Profit : 20 + 15 + 5 = 40

Constraints taken while writing the algorithm:

Step-1:Start

Step-2:Each Job takes 1 unit of time.

Step-3:Arrange profit in decending order.

Step-4:Let's suppose, each Job need 1 hr for completion & J1 is ready to wait for 1 hrs, J4 is ready to wait for 3 hrs.

Step-5:Nobody is ready to wait beyong 3 hrs.

Step-6:J4 job is done in 1 hrs, but he is ready to wait for 3 hrs, But we have to look for the for the profit(Max. Profit comes first).

Step-7: Stop

Program:

```
import java.util.*;

public class job

{

  public static void main(String args[])

  {

    Scanner sc=new Scanner(System.in);

     System.out.println("Enter the number of Jobs");

       int n=sc.nextInt();
  String a[]=new String[n];
      int b[]=new int[n];
          int c[]=new int[n];
            for(int i=0;i<n;i++)
{
   System.out.println("Enter the Jobs");
     a[i]=sc.next();
       System.out.println("Enter the Profit");
          b[i]=sc.nextInt();
            System.out.println("Enter the DeadLine");
               c[i]=sc.nextInt();
 }
       System.out.println("--Arranged Order--");
         System.out.print("Jobs:    ");
            for(int i=0;i<n;i++)
             {
                  System.out.print(a[i]+" ");
             }
                  System.out.println();
                    System.out.print("Profit:  ");
                         for(int i=0;i<n;i++)
              {
                  System.out.print(b[i]+" ");
              }
```

```
System.out.println();
     System.out.print("DeadLine:");
         for(int i=0;i<n;i++)
          {
               System.out.print(c[i]+" ");
          }
           for(int i=0;i<n-1;i++)
            {
                for(int j=i+1;j<n;j++)
                  {
                     if(b[i]<b[j])
                     {
                            int temp=b[i];
                                b[i]=b[j];
                               b[j]=temp;
                                temp=c[i];
                               c[i]=c[j];
                               c[j]=temp;
                               String temp1=a[i];
                                a[i]=a[j];
                               a[j]=temp1;
                      }
                   }
           }
```

```
System.out.println();
 System.out.println("--Sorted Order--");
  System.out.print("Jobs:    ");
    for(int i=0;i<n;i++)
 {
  System.out.print(a[i]+" ");
 }
  System.out.println();
   System.out.print("Profit:  ");
     for(int i=0;i<n;i++)
  {
    System.out.print(b[i]+" ");
  }
        System.out.println();
       System.out.print("DeadLine:");
        for(int i=0;i<n;i++)
    {
     System.out.print(c[i]+" ");
    }
      System.out.println();
     int max=c[0];
     for(int i=0;i<n;i++)
    {
     if(c[i]>max)
 {
    max=c[i];
 }
}
 String x[]=new String[max];
   int xx[]=new int[max];
     int profit=0;
      for(int i=0;i<n;i++)
 {
   int pp=c[i];
    pp=pp-1;
    if(x[pp]==null )
   {
      x[pp]=a[i];
       profit+=b[i];
   }
      else
    {
      while(pp!=-1)
     {
        if(x[pp]==null)
        {
            x[pp]=a[i];
            profit+=b[i];
```

```
                                        break;
                                    }
                                      pp=pp-1;
                                  }
                               }
                             }
                              for(int i=0;i<max;i++)
                              {
                                System.out.print("-->"+x[i]);
                              }
                               System.out.println();
                                System.out.print("Profit Earned"+profit);
    }
}
```

Output:

```
Enter the number of Jobs

5

Enter the Jobs

J1

Enter the Profit

20

Enter the DeadLine

2

Enter the Jobs
```

```
J2

Enter the Profit

15

Enter the DeadLine

2

Enter the Jobs

J3

Enter the Profit

10

Enter the DeadLine

1

Enter the Jobs

J4

Enter the Profit

5

Enter the DeadLine

3

Enter the Jobs

J5

Enter the Profit

1

Enter the DeadLine

3

--Arranged Order--
```

```
Jobs:   J1 J2 J3 J4 J5

Profit:  20 15 10 5 1

DeadLine:2 2 1 3 3

--Sorted Order--

Jobs:   J1 J2 J3 J4 J5

Profit:  20 15 10 5 1

DeadLine:2 2 1 3 3

-->J2-->J1-->J4

Profit Earned40
```
job sequencing with deadlines.

Result: Thus , in the above program successfully executed without errors using greedy algorithm for job sequencing with deadlines.

Task-5:

Implement in Java, the 0/1 Knapsack problem using greedy approach.

Aim: To implement in Java, the 0/1 Knapsack problem using greedy approach.

Description:

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights of 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick the 1kg item from the 2kg item (the item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by dynamic programming.

Algorithm:

```
    profit[0:n-1]  // contains the profit of item
    weight[0:n-1]  // contains the weight of item
     int capacity, i,n;
      solution:=0;
        // sort weight in ascending order
     // sort   profit  in ascending order
   // sort profit/weight in ascending order
       while i<n do
       if weight[i] <= capacity then
       solution +=profit[i];
        capacity-=weight[i];
   end
   i++
 End
```

Or

Step-1: Find profit/weight for each object.

Step-2: Since we have to select whether we have to completely select the object or partially select it.

So we sort the profit/weight in descending order.

Step-3:The object with the highest profit/weight is selected first.

Step- 4:Mark the object with 1 if it's completely selected or the fraction part if it is not selected completely.

Step- 5:While we select a particular object, Deduct the knapsack size by its particular object size.

Step 6:-Repeat steps 4 & 5.

Step-7:Note the final fraction part and count that object in the Knapsack(Remaining weight/Total weight).

Step- 8:Find the total weight (Summesion of weights*(Selected objects weight)).

Program:

```
import java.util.Scanner;

public class Knapsack

{

  public static void main(String[] args)

    {

      Scanner sc=new Scanner(System.in);

        int object,m;

         System.out.println("Enter the Total Objects");

           object=sc.nextInt();

            int weight[]=new int[object];

             int profit[]=new int[object];

            for(int i=0;i<object;i++)
```

```
{
   System.out.println("Enter the Profit");
     profit[i]=sc.nextInt();
      System.out.println("Enter the weight");
     weight[i]=sc.nextInt();
}
 System.out.println("Enter the Knapsack capacity");
  m=sc.nextInt();
   double p_w[]=new double[object];
        for(int i=0;i<object;i++)
        {
          p_w[i]=(double)profit[i]/(double)weight[i];
        }
          System.out.println("");
           System.out.println("-------------------");
              System.out.println("-----Data-Set------");
                 System.out.print("-------------------");
                    System.out.println("");
                       System.out.print("Objects");
                          for(int i=1;i<=object;i++)
                           {
                             System.out.print(i+" ");
                           }
                             System.out.println();
System.out.print("Profit ");
  for(int i=0;i<object;i++)
{
  System.out.print(profit[i]+"  ");
}
    System.out.println();
      System.out.print("Weight ");
         for(int i=0;i<object;i++)
          {
             System.out.print(weight[i]+" ");
          }
            System.out.println();
             System.out.print("P/W");
            for(int i=0;i<object;i++)
             {
               System.out.print(p_w[i]+"  ");
              }
                for(int i=0;i<object-1;i++)
                  {
                    for(int j=i+1;j<object;j++)
                    {
                     if(p_w[i]<p_w[j])
                      {
                        double temp=p_w[j];
```

```
        p_w[j]=p_w[i];
          p_w[i]=temp;
        int temp1=profit[j];
          profit[j]=profit[i];
               profit[i]=temp1;
               int temp2=weight[j];
                   weight[j]=weight[i];
                    weight[i]=temp2;
    }
  }
}
 System.out.println("");
   System.out.println("-------------------");
    System.out.println("--After Arranging--");
     System.out.print("-------------------");
        System.out.println("");
         System.out.print("Objects");
   for(int i=1;i<=object;i++)
   {
      System.out.print(i+"    ");
   }
    System.out.println();
     System.out.print("Profit ");
        for(int i=0;i<object;i++)
```

```
{
  System.out.print(profit[i]+" ");
}
  System.out.println();
     System.out.print("Weight ");
       for(int i=0;i<object;i++)
  {
     System.out.print(weight[i]+" ");
  }
    System.out.println();
    System.out.print("P/W    ");
     for(int i=0;i<object;i++)
       {
         System.out.print(p_w[i]+"  ");
       }
           int k=0;
              double sum=0;
               System.out.println();
                  while(m>0)
             {
              if(weight[k]<m)
             {
               sum+=1*profit[k];
                m=m-weight[k];
```

```
                                                      }
                                                       else
                                                        {
                                                          double x4=m*profit[k];
                                                          double x5=weight[k];
                                                          double x6=x4/x5;
                                                           sum=sum+x6;
                                                               m=0;
                                                        }
                                                        k++;
                                                     }
                                                     System.out.println("Final Profit is="+sum);
    }
}
```

Output:

```
Enter the Total Objects
7
Enter the Profit
10
Enter the weight
2
Enter the Profit
5
Enter the weight
```

```
3

Enter the Profit

15

Enter the weight

5

Enter the Profit

7

Enter the weight

7

Enter the Profit

6

Enter the weight

1

Enter the Profit

18

Enter the weight

4

Enter the Profit

3

Enter the weight

1

Enter the Knapsack capacity

15

-------------------
```

```
-----Data-Set------
-------------------
Objects1   2   3   4   5   6   7
Profit 10   5   15   7   6   18   3
Weight 2   3   5   7   1   4   1
P/W   5.0  1.6666666666666667  3.0  1.0  6.0  4.5  3.0
-------------------
--After Arranging--
-------------------
Objects1   2   3   4   5   6   7
Profit 6   10   18   15   3   5   7
Weight 1   2   4   5   1   3   7
P/W   6.0  5.0  4.5  3.0  3.0  1.6666666666666667  1.0
Final Profit is=55.333333333333336
```

Result: Thus, the above program was successfully executed without errors using the Knapsack problem using the greedy approach.

Task-6:

Write a java program to implement Dijkstra’s algorithm for the single-source shortest path problem.

Aim: To write a java program to implement Dijkstra’s algorithm for the single-source shortest path problem.

Description:

Dijkstra algorithm is one of the prominent algorithms to find the shortest path from the source node to a destination node. It uses the greedy approach to find the shortest path. The concept of the Dijkstra algorithm is to find the shortest distance (path) starting from the source point and to ignore the longer distances while doing an update.

Algorithm:

Step-1: All nodes should be marked as unvisited.
Step-2: All the nodes must be initialized with the "infinite" (a big number) distance. The starting node must be initialized with zero.
Step-3: Mark starting node as the current node.
Step-4: From the current node, analyze all of its neighbors that are not visited yet, and compute their distances by adding the weight of the edge, which establishes the connection between the current node and neighbor node to the current distance of the current node.
Step-5: Now, compare the recently computed distance with the distance allotted to the neighboring node, and treat it as the current distance of the neighboring node,
Step-6: After that, the surrounding neighbors of the current node, which has not been visited, are considered, and the current nodes are marked as visited.
Step-7: When the ending node is marked as visited, then the algorithm has done its job; otherwise,
Step-8: Pick the unvisited node which has been allotted the minimum distance and treat it as the new current node. After that, start again from step4

Program:

```
import java.util.*;

import java.io.*;

import java.lang.*;

public class DijkstraExample

{
```

```
static final int totalVertex = 9;

int minimumDistance(int distance[], Boolean spSet[])

{

  // Initialize min value

   int m = Integer.MAX_VALUE, m_index = -1;

  for (int vx = 0; vx < totalVertex; vx++)

  {

    if (spSet[vx] == false && distance[vx] <= m)

    {

      m = distance[vx];

      m_index = vx;

    }

  }

  return m_index;

}

void printSolution(int distance[], int n)

{

  System.out.println("The shortest Distance from source 0th node to all other nodes are: ");

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

   System.out.println("To " + j + " the shortest distance is: " + distance[j]);

}

 void dijkstra(int graph[][], int s)

 {

   int distance[] = new int[totalVertex];
   Boolean spSet[] = new Boolean[totalVertex];
   for (int j = 0; j < totalVertex; j++)
  {
    distance[j] = Integer.MAX_VALUE;
     spSet[j] = false;
   }
    distance[s] = 0;
    for (int cnt = 0; cnt < totalVertex - 1; cnt++)
    {
      int ux = minimumDistance(distance, spSet);
        spSet[ux] = true;
         for (int vx = 0; vx < totalVertex; vx++)
         if (!spSet[vx] && graph[ux][vx] != -1 && distance[ux] != Integer.MAX_VALUE &&
distance[ux] + graph[ux][vx] < distance[vx])
        {
          distance[vx] = distance[ux] + graph[ux][vx];
        }
    }
      printSolution(distance, totalVertex);
  }
  public static void main(String argvs[])
  {
    int graph[][] = new int[][] { { -1, 3, -1, -1, -1, -1, -1, 7, -1 },
   { 3, -1, 7, -1, -1, -1, -1, 10, 4 },
   { -1, 7, -1, 6, -1, 2, -1, -1, 1 },
   { -1, -1, 6, -1, 8, 13, -1, -1, 3 },
   { -1, -1, -1, 8, -1, 9, -1, -1, -1 },
   { -1, -1, 2, 13, 9, -1, 4, -1, 5 },
   { -1, -1, -1, -1, -1, 4, -1, 2, 5 },
   { 7, 10, -1, -1, -1, -1, 2, -1, 6 },
   { -1, 4, 1, 3, -1, 5, 5, 6, -1 } };
  DijkstraExample obj = new DijkstraExample();
     obj.dijkstra(graph, 0);
  }
}
```

Output:

```
java -cp /tmp/YlMq7kldTi DijkstraExample
The shortest Distance from source 0th node to all other nodes are:
To 0 the shortest distance is: 0To 1 the shortest distance is: 3To 2 the shortest distance is: 8To 3 the shortest distance is: 10To 4 the shortest distance is: 18To 5 the shortest distance is: 10
To 6 the shortest distance is: 9
To 7 the shortest distance is: 7
To 8 the shortest distance is: 7
```

Result: Thus, the above program was successfully executed without errors using implementing Dijkstra's algorithm for the single-source shortest path problem.

Task-7:

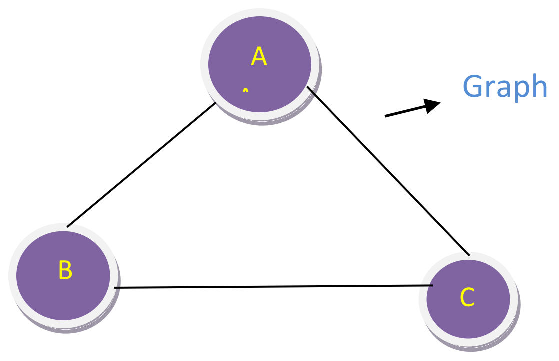
Write a java program that implements Prim's algorithm to generate minimum cost spanning tree.

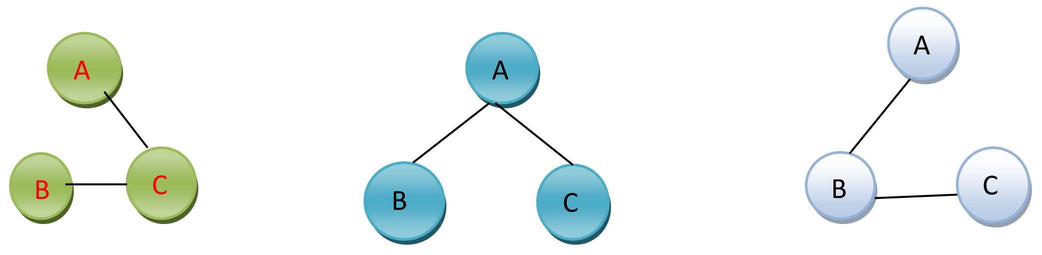
Aim: To write a java program that implements Prim's algorithm to generate minimum cost spanning tree.

Description:

Minimum Spanning Tree: A tree that contains every vertex of a connected graph g is a spanning tree.

For example



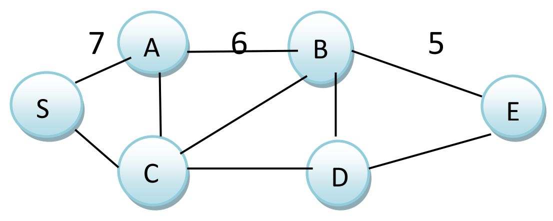


There are two types of Algorithms for MST:

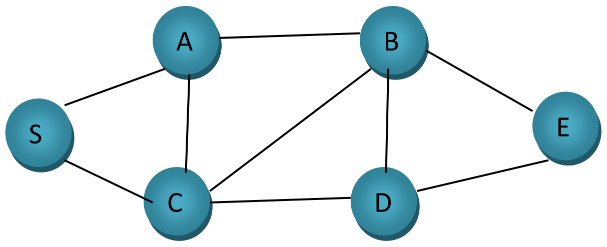
i) Prim's
ii) Kruskal's

i)Prism:

It used greedy approach to find MST.



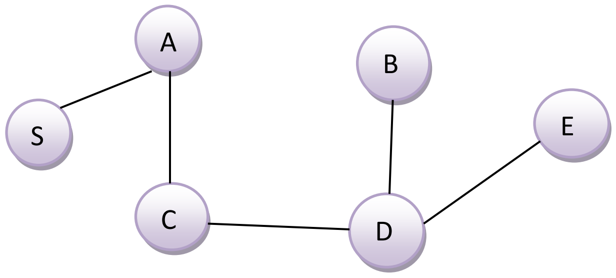
1) Remove all loops & parallel edges.



2) Choose any arbitrary node as root node

    ‘s’

3) Check outgoing edges and select one with less cost



= 7+3+3+2+2

Total weight =17

Algorithm:

```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3: Select an edge 's' connecting the tree vertex and fringe vertex that has a minimum weight
Step 4: Add the selected edge and the vertex to the minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

Program:

```
import java.util.*;

import java.lang.*;
import java.io.*;

public class MST

{

   private static final int V = 5;

   int minKey(int key[], Boolean mstSet[])

   {

      int min = Integer.MAX_VALUE, min_index = -1;

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

         if (mstSet[v] == false && key[v] < min)

         {

            min = key[v];

            min_index = v;

         }

           return min_index;

   }

     void printMST(int parent[], int graph[][])

      {

         System.out.println("Edge \tWeight");

          for (int i = 1; i < V; i++)

         System.out.println(parent[i] + " - " + i + "\t" + graph[i][parent[i]]);

     }

        void primMST(int graph[][])
```

```
{
  int parent[] = new int[V];
  int key[] = new int[V];
Boolean mstSet[] = new Boolean[V];
  for (int i = 0; i < V; i++)
  {
    key[i] = Integer.MAX_VALUE;
    mstSet[i] = false;
   }
     key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V - 1; count++)
    {
         int u = minKey(key, mstSet);
            mstSet[u] = true;
       for (int v = 0; v < V; v++)
      if (graph[u][v] != 0 && mstSet[v] == false && graph[u][v] < key[v])
      {
        parent[v] = u;
        key[v] = graph[u][v];
      }
  }
```

```
            printMST(parent, graph);
    }
      public static void main(String[] args)
       {
        MST t = new MST();
        int graph[][] = new int[][] {
                                    { 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 }

                                  };
        t.primMST(graph);
    }
}
```

Output:

Edge	Weight
0 – 1	2
1 - 2	3
0 - 3	6
1 - 4	5

Result: Thus, the above program was successfully executed without errors using Prim's algorithm to generate a minimum cost spanning tree.

Task-8:

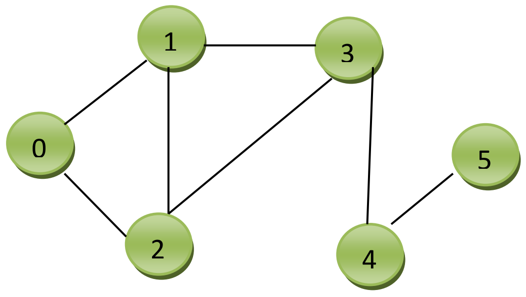
Write a java program that implements Kruskal’s algorithm to generate minimum cost spanning tree.

Aim: To write a java program that implements Kruskal’s algorithm to generate minimum cost spanning tree.

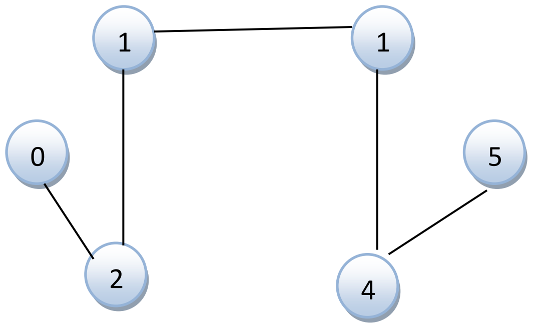
Description:

Kruskal’s algorithm for finding the Minimum Spanning Tree(MST), which finds an edge of the least possible weight that connects any two trees in the forest
It is a greedy algorithm.
It finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.
If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).
Number of edges in MST: V-1 (V – no of vertices in Graph)

[1]Example:



Undirected Graph



Minimum Spanning Tree

Algorithm:

Step-1: Sort the edges in ascending order of weights.
Step-2:Pick the edge with the least weight. Check if including this edge in the spanning tree will form a cycle is Yes then ignore it if No then add it to the spanning tree.
Step-3:Repeat step- 2 till the spanning tree has V-1 (V – no of vertices in Graph).
Step-4:The spanning tree with the least weight will be formed, called Minimum Spanning Tree.

Program:

```
import java.util.ArrayList;

import java.util.Comparator;

import java.util.PriorityQueue;

public class KrushkalMST

{

  static class Edge

  {

   int source;

   int destination;

    int weight;

    public Edge(int source, int destination, int weight) {

     this.source = source;

    this.destination = destination;

     this.weight = weight;

   }

 }

  static class Graph
```

```
{
 int vertices;
ArrayList<Edge> allEdges = new ArrayList<>();
  Graph(int vertices)
  {
   this.vertices = vertices;
  }
   public void addEgde(int source, int destination, int weight)
   {
      Edge edge = new Edge(source, destination, weight);
     allEdges.add(edge); //add to total edges
    }
     public void kruskalMST()
     {
      PriorityQueue<Edge> pq = new PriorityQueue<>(allEdges.size(),
     Comparator.comparingInt(o –> o.weight));
       //add all the edges to priority queue,
       //sort the edges on weights
        for (int i = 0; i <allEdges.size() ; i++)
       {
         pq.add(allEdges.get(i));
       }
   //create a parent []
   int [] parent = new int[vertices];
  //makeset
     makeSet(parent);
 ArrayList<Edge> mst = new ArrayList<>();
   //process vertices – 1 edges
  int index = 0;
while(index<vertices–1)
{
  Edge edge = pq.remove();
 //check if adding this edge creates a cycle
  int x_set = find(parent, edge.source);
int y_set = find(parent, edge.destination);
 if(x_set==y_set)
 {
  //ignore, will create cycle
}
  else
  {
   //add it to our final result
    mst.add(edge);
     index++;
```

```
            union(parent,x_set,y_set);
          }
        }
        //print MST
      System.out.println("Minimum Spanning Tree: ");
        printGraph(mst);
    }
    public void makeSet(int [] parent)
    {
     //Make set- creating a new element with a parent pointer to itself.
      for (int i = 0; i <vertices ; i++)
      {
       parent[i] = i;
      }
    }
       public int find(int [] parent, int vertex)
       {
         //chain of parent pointers from x upwards through the tree
               // until an element is reached whose parent is itself
         if(parent[vertex]!=vertex)
          return find(parent, parent[vertex]);;
         return vertex;
        }
            public void union(int [] parent, int x, int y)
            {
              int x_set_parent = find(parent, x);
              int y_set_parent = find(parent, y);
             //make x as parent of y
              parent[y_set_parent] = x_set_parent;
            }
             public void printGraph(ArrayList<Edge> edgeList)
             {
               for (int i = 0; i <edgeList.size() ; i++)
                {
                  Edge edge = edgeList.get(i);
                  System.out.println("Edge-" + i + " source: " + edge.source +"
destination: " + edge.destination +
                      " weight: " + edge.weight);
                }
            }
        }
         public static void main(String[] args)
         {
          int vertices = 6;
```

```
            Graph graph = new Graph(vertices);

             graph.addEgde(0, 1, 4);

              graph.addEgde(0, 2, 3);

              graph.addEgde(1, 2, 1);

              graph.addEgde(1, 3, 2);

              graph.addEgde(2, 3, 4);

             graph.addEgde(3, 4, 2);

              graph.addEgde(4, 5, 6);

              graph.kruskalMST();

        }

}
```

Output:

```
Minimum Spanning Tree:
Edge-0 source: 1 destination: 2 weight: 1
Edge-1 source: 1 destination: 3 weight: 2
Edge-2 source: 3 destination: 4 weight: 2
Edge-3 source: 0 destination: 2 weight: 3
Edge-4 source: 4 destination: 5 weight: 6
```

Result: Thus, the above program was successfully executed without errors using Kruskal’s algorithm to generate a minimum cost spanning tree.

Task-9

Implement All-Pairs Shortest Paths problem using Floyd's algorithm

Aim: To implement the All-Pairs Shortest Paths problem using Floyd's algorithms.

Description:

Floyd's Algorithm:

Floyd‟s algorithm is applicable to both directed and undirected graphs provided that they do not contain a cycle. It is convenient to record the lengths of shortest path in an n- by- n matrix D called the distance matrix. The element dij in the ith row and jth column of matrix indicates the shortest path from the ith vertex to jth vertex (1<=i, j<=n). The element in the ith row and jth column of the current matrix D(k-1) is replaced by the sum of elements in the same row i and kth column and in the same column j and the kth column if and only if the latter sum is smaller than its current value.

Complexity: The time efficiency of Floyd‟s algorithm is cubic i.e. Θ (n^3)

Algorithm:

Step 1: Initialize the shortest paths between any 2 vertices with Infinity.
Step 2: Find all pair shortest paths that use 0 intermediate vertices, then find the shortest paths that use 1 intermediate vertex, and so on.. until using all N vertices as intermediate nodes.
Step 3: Minimize the shortest paths between any 2 pairs in the previous operation.
Step 4: For any 2 vertices (i,j) , one should actually minimize the distances between this pair using the first K nodes, so the shortest path will be: min(dist[i][k]+dist[k][j],dist[i][j]). dist[i][k] represents the shortest path that only uses the first K vertices, and dist[k][j] represents the shortest path between the pair k,j. As the shortest path will be a concatenation of the shortest path from I to k, then from k to j.

Program:

```
import java.util.*;

import java.lang.*;

import java.io.*;

public class AllPairShortestPath

{

  final static int INF = 99999, V = 4;
```

```
void floydWarshall(int graph[][])
{
        int dist[][] = new int[V][V];
        int i, j, k;
        for (i = 0; i < V; i++)
                for (j = 0; j < V; j++)
                        dist[i][j] = graph[i][j];

        for (k = 0; k < V; k++)
        {
           for (i = 0; i < V; i++)
                {
                        for (j = 0; j < V; j++)
                        {
                                if (dist[i][k] + dist[k][j] < dist[i][j])
                                        dist[i][j] = dist[i][k] + dist[k][j];
                        }
                }
        }
        printSolution(dist);
}
void printSolution(int dist[][])
{
        System.out.println("The following matrix shows the shortest "+
                                                        "distances between every pair of
vertices");
                        for (int i=0; i<V; ++i)
                        {
                                for (int j=0; j<V; ++j)
                                {
                                        if (dist[i][j]==INF)
                                                System.out.print("INF ");
                                        else
                                                System.out.print(dist[i][j]+" ");
                                }
                                System.out.println();
                        }
                }
                public static void main (String[] args)
                {
                        int graph[][] = {
                                    {0, 5, INF, 10},
                                                         {INF, 0, 3, INF},
                                                        {INF, INF, 0, 1},
                                                        {INF, INF, INF, 0}
                                                        };
                                AllPairShortestPath a = new AllPairShortestPath();
                                a.floydWarshall(graph);
```

```
                    }
}
```

Output:

The following matrix shows the shortest distances between every pair of vertices

```
0 5 8 9

INF 0 3 4

INF INF 0 1

INF INF INF 0
```

Result: Thus, the above program was successfully executed without errors using the All-Pairs Shortest Paths problem using Floyd's algorithm.

Task-10

Write a java program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem.

Aim: To write a java program to implement Dynamic Programming algorithm for the 0/1 Knapsack problem.

Description:

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that the sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don't pick it (0-1 property).

Algorithm:

Step-1: Find solutions to the smallest sub-problems.

Step-2: Find out the formula (or rule) to build a solution to a sub-problem through solutions of even the smallest subproblems.

Step-3: Create a table that stores the solutions to sub-problems. Then calculate the solution of the subproblem according to the found formula and save it to the table.

Step-4:From the solved subproblems, you find the solution to the original problem.

Program:

```
public class Knapsack
{
                    static int max(int a, int b)
 {
   return (a > b) ? a : b;
 }
```

```
static int knapSack(int W, int wt[], int val[], int n)
{
        int i, w;
        int K[][] = new int[n + 1][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];
}
public static void main(String args[])
{
        int val[] = new int[] { 60, 100, 120 };
        int wt[] = new int[] { 10, 20, 30 };
```

```
                        int W = 50;

                        int n = val.length;

                        System.out.println(knapSack(W, wt, val, n));

                }

}
```

Output:

```
220
```

Result: Thus, the above program was successfully executed without errors using the implemented Dynamic Programming algorithm for the 0/1 Knapsack problem.

Task-11

Implement Travelling Sales Person problem using Dynamic programming.

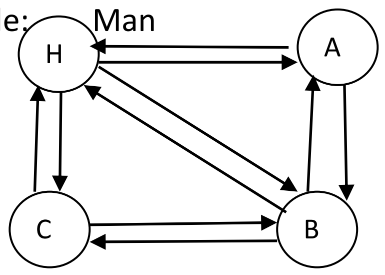
Aim: To implement Travelling Sales Person problem using Dynamic programming.

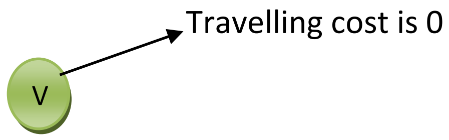
Description:

TSM problem consists of salesman and a set of cities.

The salesmen has to vist each city starting from home and returning to the same city.

Example:





The person wants to minimize the total length of the trip.

Let g(i,s) be the length of the shortest path starting at vertex i, going through all vertices in s and terminating at vertex 1.

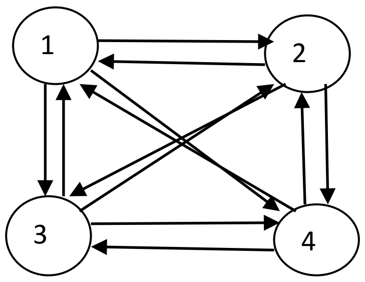
The function(1,v-{1}) is the length of the optimal sales persons tour.

g(1,v-{1})=min i<k<n

$((c_{ik} + g(k, v - \{i, k\}))$ ----1

g(i,s)=min j€s $((c_{jk} + g(k, v - \{i, k\}))$ ----2

Example:



The cost adjancency matrix

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{array}$$

let us start the tour

vertex 1:

g(1,v-{1}) = min [c_{1k}+g(k,v-{1,k}]----1

g(i,s)= min {c_{ij}+g(j,s-{j}}----2

clearly

g(i,Ø)= c_{i1} so that

g(2,Ø)= c_{21}=5

g(3,Ø)= c_{31}=6

g(4,Ø)= c_{41}=8

we have to find out the person start from vertex one he has to visit all set of the vertex two, three and four. So this we have to find out .

g(1,{2,3,4}) = min {c_{12}+g(2,{3,4}), $c_{13} + g(3,\{2,4\})$,

$c_{14} + g(4\{2,3\})\}$

So I have to find out above one.

So if I want to find out this I want to know the value of c_{12}, , c_{13} and , c_{14} and this so let us calculate individually above one.

1) g(2,{3,4})= min{c_{23}+g(3{4}),c_{24}+g(4{3})}

= min {9+g(3,{4})+10+g(4{3})

g(3{4}) = min {c_{34}+g(4,ø)}

= 12+8= 20

g(4{3})= min {c_{43}+g(3,Ø)}

= 9+6 =15

g(2,{3,4})= min{9+20,10+15}

=min{29,25}= 25

2) g(3,{2,4})= min{$c_{32} + g(2\{4\}), c_{34} + g(4\{2\})$}

g(2{4})= min {$c_{24} + g(4, Ø)$}

= min{10+8}=18

g(4{2})= min {$c_{42} + g(2, Ø)$}

= min{8+5} =13

g(3,{2,4})= min{13+18, 12+13}

=min {31,25}=25

3) $g(4\{2,3\})$=min {$c_{42} + g(2\{3\})$, $c_{43} + g(3\{2\})$}

g(2{3})=min{$c_{23} + g(3, Ø)$}

= 9+6=15

g(3{2})=min{$c_{32} + g(2, Ø)$}

= 13+5=18

$g(4\{2,3\})$=min {8+15,9+18}

=min{23,27}=23

we got the three values so just substitute those three values in the given equation

g(1,{2,3,4})= min {c_{12}+g(2,{3,4}), $c_{13}+g(3,\{2,4\}), c_{14}+g(4\{2,3\})\}$

=min {10+25, 15+25, 20+23}

= min {35, 40, 43} =35

Therefore optimal tour for the graph has length=35

Therefore optimal tour is 1,2,4,3,1

Algorithm:

```
C ({1}, 1) = 0
for s = 2 to n do
for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
C (S, 1) = ∞
for all j Є S and j ≠ 1
C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

Program:

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
public class Main
{
  private final int N, start;
  private final double[][] distance;
  private List<Integer> tour = new ArrayList<>();

  private double minTourCost = Double.POSITIVE_INFINITY;

  private boolean ranSolver = false;

  public Main(double[][] distance)

  {

   this(0, distance);

  }

  public Main(int start, double[][] distance)

  {

   N = distance.length;


   if (N <= 2) throw new IllegalStateException("N <= 2 not yet supported.");

   if (N != distance[0].length) throw new IllegalStateException("Matrix must

   be square (n x n)");

   if (start < 0 || start >= N) throw new IllegalArgumentException("Invalid start node.");

   this.start = start;

   this.distance = distance;

  }

  public List<Integer> getTour()

  {

   if (!ranSolver) solve();

   return tour;

  }
```

```
 public double getTourCost()
 {
  if (!ranSolver) solve();
  return minTourCost;
}
public void solve()
{
 if (ranSolver) return;
 final int END_STATE = (1 << N) - 1;
 Double[][] memo = new Double[N][1 << N];
 for (int end = 0; end < N; end++)
 {
   if (end == start) continue;
   memo[end][(1 << start) | (1 << end)] = distance[start][end];
 }
 for (int r = 3; r <= N; r++)
 {
   for (int subset : combinations(r, N))
   {
     if (notIn(start, subset)) continue;
     for (int next = 0; next < N; next++)
     {
       if (next == start || notIn(next, subset)) continue;
```

```
      int subsetWithoutNext = subset ^ (1 << next);
      double minDist = Double.POSITIVE_INFINITY;
      for (int end = 0; end < N; end++)
      {
        if (end == start || end == next || notIn(end, subset)) continue;
        double newDistance = memo[end][subsetWithoutNext] + distance[end][next];
        if (newDistance < minDist)
        {
          minDist = newDistance;
        }
      }
       memo[next][subset] = minDist;
    }
  }
}
for (int i = 0; i < N; i++)
{
  if (i == start) continue;
  double tourCost = memo[i][END_STATE] + distance[i][start];
  if (tourCost < minTourCost)
  {
    minTourCost = tourCost;
  }
```

```
}
int lastIndex = start;
int state = END_STATE;
tour.add(start);
for (int i = 1; i < N; i++)
{
  int index = -1;
  for (int j = 0; j < N; j++)
  {
    if (j == start || notIn(j, state)) continue;
    if (index == -1) index = j;
    double prevDist = memo[index][state] + distance[index][lastIndex];
    double newDist  = memo[j][state] + distance[j][lastIndex];
    if (newDist < prevDist)
    {
      index = j;
    }
  }
   tour.add(index);
   state = state ^ (1 << index);
   lastIndex = index;
}
tour.add(start);
```

```
  Collections.reverse(tour);
  ranSolver = true;
}
private static boolean notIn(int elem, int subset)
{
  return ((1 << elem) & subset) == 0;
}
public static List<Integer> combinations(int r, int n)
{
  List<Integer> subsets = new ArrayList<>();
  combinations(0, 0, r, n, subsets);
  return subsets;
}
private static void combinations(int set, int at, int r, int n, List<Integer> subsets)
{
  int elementsLeftToPick = n - at;
  if (elementsLeftToPick < r) return;
  if (r == 0)
  {
    subsets.add(set);
  }
  else
   {
    for (int i = at; i < n; i++)
    {
      set |= 1 << i;
     combinations(set, i + 1, r - 1, n, subsets);
     set &= ~(1 << i);
   }
  }
}
public static void main(String[] args)
{
  // Create adjacency matrix
  int n = 6;
  double[][] distanceMatrix = new double[n][n];
  for (double[] row : distanceMatrix) java.util.Arrays.fill(row, 10000);
  distanceMatrix[5][0] = 10;
  distanceMatrix[1][5] = 12;
  distanceMatrix[4][1] = 2;
  distanceMatrix[2][4] = 4;
  distanceMatrix[3][2] = 6;
  distanceMatrix[0][3] = 8;
  int startNode = 0;
  Main solver = new Main(startNode, distanceMatrix);
  System.out.println("Tour: " + solver.getTour());
```

```
    System.out.println("Tour cost: " + solver.getTourCost());
  }
}
```

Output:

```
Tour: [0, 3, 2, 4, 1, 5, 0]

Tour cost: 42.0
```

Result: Thus, the above program was successfully executed without errors using Travelling Sales Person problem using Dynamic programming.

Task-12

Write a java program to implement the backtracking algorithm for the sum of subsets problem of a given set S = {Sl, S2,.....,Sn} 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}. Display a suitable message, if the given problem instance doesn't have a solution

Aim: To write a java program to implement the backtracking algorithm for the sum of sub-sets problem of a given set.

Description:

Sum of Sub-sets

Sub-set-Sum Problem is to find a subset of a given set S= {s1, s2… sn} of n positive integers whose sum is equal to a given positive integer d. It is assumed that the set‟s elements are sorted in increasing order. The state-space tree can then be constructed as a binary tree and applying a backtracking algorithm, the solutions could be obtained. Some instances of the problem may have no solutions

Complexity: Subset sum problem solved using backtracking generates at each step maximal two new sub-trees, and the running time of the bounding functions is linear, so the running time is O(2n).

Algorithm:

step-1:Start with an empty set.

Step-2:Add to the subset, the next element from the list.

Step-3:If the subset is having sum m then stop with that subset as the solution.

Step-4:If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

Step-5:If the subset is feasible then repeat step 2.

Step-6:If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without a solution.

Program:

```
public class Main

{

   public static boolean subsetSum(int[] A, int n, int k)
```

```
{
   if (k == 0)
   {
      return true;
   }
    if (n < 0 || k < 0)
    {
       return false;
    }
    boolean include = subsetSum(A, n - 1, k - A[n]);
    boolean exclude = subsetSum(A, n - 1, k);

    return include || exclude;
 }
  public static void main(String[] args)
 {
   int[] A = {1, 2,5,6, 8};
   int k = 9;
   if (subsetSum(A, A.length - 1, k))
   {
      System.out.print("Subsequence with the given sum exists");
   }
    else
```

```
        {
          System.out.print("Subsequence with the given sum does not exist");
        }
    }
}
```

Output:

```
Subsequence with the given sum exists
```

Result: Thus, the above program was successfully executed without errors using the backtracking algorithm for the sum of sub-sets problem of a given set.

Task-13

Write a java program to implement the backtracking algorithm for the Hamiltonian Circuits problem

Aim: To write a java program to implement the backtracking algorithm for the Hamiltonian Circuits problem.

Description:

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains the Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

Algorithm:

```
while there are untried conflagrations
{
    generate the next configuration
    if ( there are edges between two consecutive vertices of this
      configuration and there is an edge from the last vertex to
    the first ).
     {
        print this configuration;
         break;
    }
}
```

Program:

```
public class HamiltonianCycle
{
```

```
final int V = 5;

int path[];

boolean isSafe(int v, int graph[][], int path[], int pos)

{

   if (graph[path[pos - 1]][v] == 0)

      return false;

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

      if (path[i] == v)

         return false;


   return true;

}

 boolean hamCycleUtil(int graph[][], int path[], int pos)

 {

   if (pos == V)

   {

      if (graph[path[pos - 1]][path[0]] == 1)

         return true;

      else

         return false;

   }

    for (int v = 1; v < V; v++)

     {
            if (isSafe(v, graph, path, pos))
            {
                path[pos] = v;
                if (hamCycleUtil(graph, path, pos + 1) == true)
                    return true;
                path[pos] = -1;
            }
        }
        return false;
    }
    int hamCycle(int graph[][])
    {
        path = new int[V];
        for (int i = 0; i < V; i++)
            path[i] = -1;
        path[0] = 0;
        if (hamCycleUtil(graph, path, 1) == false)
        {
            System.out.println("\nSolution does not exist");
            return 0;
        }

        printSolution(path);
    return 1;
}

void printSolution(int path[])
{
    System.out.println("Solution Exists: Following" +
                       " is one Hamiltonian Cycle");
    for (int i = 0; i < V; i++)
        System.out.print(" " + path[i] + " ");
    System.out.println(" " + path[0] + " ");
}

public static void main(String args[])
{
    HamiltonianCycle hamiltonian = new HamiltonianCycle();
    int graph1[][] =
    {
        {0, 1, 0, 1, 0},
        {1, 0, 1, 1, 1},
        {0, 1, 0, 0, 1},
        {1, 1, 0, 0, 1},
        {0, 1, 1, 1, 0},
    };
```

```
        hamiltonian.hamCycle(graph1);

        int graph2[][] =

        {

           {0, 1, 0, 1, 0},

           {1, 0, 1, 1, 1},

           {0, 1, 0, 0, 1},

           {1, 1, 0, 0, 0},

           {0, 1, 1, 0, 0},

        };

        hamiltonian.hamCycle(graph2);

    }

}
```

Output:

```
Solution Exists: Following is one Hamiltonian Cycle

 0  1  2  4  3  0

Solution does not exist
```

Result: Thus, the above program was successfully executed without errors using the backtracking algorithm for the Hamiltonian Circuits problem.