UNIVERSITY VISVESVARAYA COLLEGE OF ENGINEERING

K R Circle, Dr Ambedkar Veedhi, Bengaluru,
Karnataka 560001

Laboratory Manual
For
Algorithm Laboratory

(SUBCODE : 2K13MECS16)

Nagaraju Y

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

(1st semester M.E, IT)

1. Doubly Circular Linked List.

```
import java.util.Scanner;
class Node
{
   protected int data;
   protected Node next,prev;

   public Node()
   {
      next=null;
      prev=null;
      data=0;
   }
   public Node(int d,Node n,Node p)
   {
      data=d;
      next=n;
      prev=p;
   }
   public void setLinkNext(Node n)
   {
         next=n;
   }
   public void setLinkPrev(Node p)
   {
        prev=p;
   }
   public Node getLinkNext()
   {
       return next;
    }
   public Node getLinkPrev()
   {
      return prev;
   }
   public void setData(int d)
   {
      data=d;
   }
   public int getData()
   {
      return data;
   }
}
```

```
class linkedList
{
   protected Node start;
   protected Node end;
   public int size;
   public linkedList()
   {
       start=null;
       end=null;
       size=0;
   }
  public boolean isEmpty()
  {
     return start==null;
  }
  public int getSize()
  {
      return size;
  }
 public void insertAtStart(int val)
 {
      Node nptr=new Node(val,null,null);;
      if(start==null)
      {
             nptr.setLinkNext(nptr);
             nptr.setLinkPrev(nptr);
             start=nptr;
             end=start;
    }
    else
   {
          nptr.setLinkPrev(end);
          end.setLinkNext(nptr);
          start.setLinkPrev(nptr);
          nptr.setLinkNext(start);
          start=nptr;
  }
   size++;
 }
 public void insertAtEnd(int val)
 {
     Node nptr=new Node(val,null,null);
     if(start==null)
     {
          nptr.setLinkNext(nptr);
          nptr.setLinkPrev(nptr);
```

```
        start=nptr;
        end=start;
    }
    else
    {
        nptr.setLinkPrev(end);
        end.setLinkNext(nptr);
        start.setLinkPrev(nptr);
        nptr.setLinkNext(start);
        end=nptr;
    }
    size++;
}
public void insertAtPos(int val,int pos)
{
    Node nptr=new Node(val,null,null);
    if(pos==1)
    {
        insertAtStart(val);
        return;
    }
    Node ptr=start;
    for(int i=2;i<=size;i++)
    {
        if(i==pos)
        {
            Node tmp=ptr.getLinkNext();
            ptr.setLinkNext(nptr);
         nptr.setLinkPrev(ptr);
         nptr.setLinkNext(tmp);
         tmp.setLinkPrev(nptr);
       }
       ptr=ptr.getLinkNext();
   }
   size++;
}
public void deleteAtPos(int pos)
{
    if(pos==1)
    {
        if(size==1)
        {
         start=null;
         end=null;
         size=0;
         return;
```

```
        }
        start=start.getLinkNext();
        start.setLinkPrev(end);
        end.setLinkNext(start);
        size--;
        return;
    }
    if(pos==size)
    {
         end=end.getLinkPrev();
         end.setLinkNext(start);
         start.setLinkPrev(end);
         size--;
    }
   Node ptr=start.getLinkNext();
   for(int i=2;i<=size;i++)
   {
      if(i==pos)
      {
          Node p=ptr.getLinkPrev();
          Node n=ptr.getLinkNext();
          p.setLinkNext(n);
          n.setLinkPrev(p);
          size--;
          return;
      }
     ptr=ptr.getLinkNext();
   }
}
public void display()
{
     System.out.print("\nCircular Doubly Linked List=");
     Node ptr=start;
     if(size==0)
     {
         System.out.print("empty\n");
         return;
    }
    if(start.getLinkNext()==start)
    {
        System.out.print(start.getData()+"<->"+ptr.getData()+"\n");
        return;
    }
    System.out.print(start.getData()+"<->");
    ptr=start.getLinkNext();
    while(ptr.getLinkNext()!=start)
```

```
    {
        System.out.print(ptr.getData()+"<->");
        ptr=ptr.getLinkNext();
    }
    System.out.print(ptr.getData()+"<->");
    ptr=ptr.getLinkNext();
    System.out.print(ptr.getData()+"\n");
  }
}
public class CircularDoublyLinkedList
{
    public static void main(String[] args)
    {
        Scanner scan=new Scanner(System.in);
        linkedList list=new linkedList();
        System.out.println("Circular Doubly Linked List Test\n");
        char ch;
        do
        {
            System.out.println("\nCircular Doubly Linked List Operations\n");
            System.out.println("1.insert at beginning");
            System.out.println("2.insert at end");
            System.out.println("3.insert at position");
            System.out.println("4.delete at position");
            System.out.println("5.Check empty");
            System.out.println("6.Get size");
            int choice=scan.nextInt();
            switch(choice)
            {
              case 1:
                    System.out.println("Enter integer element to insert");
                    list.insertAtStart(scan.nextInt());
                    break;
              case 2:
                    System.out.println("Enter integer element to insert");
                    list.insertAtEnd(scan.nextInt());
                    break;
              case 3:
                    System.out.println("Enter integer element to insert");
                    int num=scan.nextInt();
                    System.out.println("Enter position");
                    int pos=scan.nextInt();
                    if(pos<1||pos>list.getSize())
                      System.out.println("Invalid position\n");
                    else
                      list.insertAtPos(num,pos);
```

```
                    break;
            case 4:
                    System.out.println("Enter position");
                    int p=scan.nextInt();
                    if(p<1||p>list.getSize())
                      System.out.println("Invalid position\n");
                    else
                       list.deleteAtPos(p);
                    break;
            case 5:
                    System.out.println("Empty status="+list.isEmpty());
                    break;
            case 6:
                    System.out.println("Size="+list.getSize()+"\n");
                    break;
            default:
                    System.out.println("Wrong Entry\n");
                    break;
       }
      list.display();
      System.out.println("\nDo you want to continue(Type y or n)\n");
      ch=scan.next().charAt(0);
   }
   while(ch=='Y'||ch=='y');
 }
}
```

output

Circular Doubly Linked List Operations

1. insert at beginning
2. insert at end
3. insert at position
4. delete at position
5. Check empty
6. Get size

Menu

1
Enter integer element to insert
30

Circular Doubly Linked List=30<->30

Do you want to continue(Type y or n)
y

Circular Doubly Linked List Operations

Menu

6
Size=1
Circular Doubly Linked List=30<->30
Do you want to continue(Type y or n)
y

Circular Doubly Linked List Operations

Menu

2
Enter integer element to insert
40
Circular Doubly Linked List=30<->40<->30
Do you want to continue(Type y or n)
y

Circular Doubly Linked List Operations

Menu

3
Enter integer element to insert
60
Enter position
1
Circular Doubly Linked List=60<->30<->40<->60
Do you want to continue(Type y or n)
y

Circular Doubly Linked List Operations

Menu

4
Enter position
4
Invalid position
Circular Doubly Linked List=60<->30<->40<->60

Do you want to continue(Type y or n)
y

Circular Doubly Linked List Operations
Menu

6
Size=3
Circular Doubly Linked List=60<->30<->40<->60

Do you want to continue(Type y or n)
y

Circular Doubly Linked List Operations

Menu

4
Enter position
3
Circular Doubly Linked List=60<->30<->60

Do you want to continue(Type y or n)
y

Circular Doubly Linked List Operations
Menu

5
Empty status=false
Circular Doubly Linked List=60<->30<->60
Do you want to continue(Type y or n)
N

2.AVL Tree

```
import java.util.Scanner;
class AVLNode
{
        AVLNode left,right;
        int data;
        int height;
        public AVLNode()
        {
                left=null;
                right=null;
                data=0;
                height=0;
        }
        public AVLNode(int n)
        {
                left=null;
                right=null;
                data=n;
                height=0;
        }
}
class AVLTree
{
        private AVLNode root;
        public AVLTree()
        {
                root=null;
        }
        public boolean isEmpty()
        {
                return root==null;
        }
        public void makeEmpty()
        {
                root=null;
        }
        public void insert(int data)
        {
                root=insert(data,root);
        }
        private int height(AVLNode t)
        {
                return t==null?-1:t.height;
        }
```

```
private int max(int lhs,int rhs)
{
        return lhs>rhs?lhs:rhs;
}
private AVLNode insert(int x,AVLNode t)
{
        if(t==null)
                t=new AVLNode(x);
        else if(x<t.data)
        {
                t.left=insert(x,t.left);
                if(height(t.left)-height(t.right)==2)
                        if(x<t.left.data)
                                t=rotateWithLeftChild(t);
                        else
                                t=doubleWithLeftChild(t);
        }
        else if(x>t.data)
        {
                t.right=insert(x,t.right);
                if(height(t.right)-height(t.left)==2)
                        if(x>t.right.data)
                                t=rotateWithRightChild(t);
                        else
                                t=doubleWithRightChild(t);
        }
        else
                ;
        t.height=max(height(t.left),height(t.right))+1;
        return t;
}
private AVLNode rotateWithLeftChild(AVLNode k2)
{
        AVLNode k1=k2.left;
        k2.left=k1.right;
        k1.right=k2;
        k2.height=max(height(k2.left),height(k2.right))+1;
        k1.height=max(height(k1.left),k2.height)+1;
        return k1;
        }
        private AVLNode rotateWithRightChild(AVLNode k1)
        {
                AVLNode k2=k1.right;
                k1.right=k2.left;
                k2.left=k1;
                k1.height=max(height(k1.left),height(k1.right))+1;
```

```
        k2.height=max(height(k2.right),k1.height)+1;
        return k2;
}
private AVLNode doubleWithLeftChild(AVLNode k3)
{
        k3.left=rotateWithRightChild(k3.left);
        return rotateWithLeftChild(k3);
}
private AVLNode doubleWithRightChild(AVLNode k1)
{
        k1.right=rotateWithLeftChild(k1.right);
        return rotateWithRightChild(k1);
}
public int countNodes()
{
        return countNodes(root);
}
private int countNodes(AVLNode r)
{
        if(r==null)
                return 0;
        else
        {
                int l=1;
                l+=countNodes(r.left);
                l+=countNodes(r.right);
                return l;
        }
}
public boolean search(int val)
{
        return search(root,val);
}
private boolean search(AVLNode r,int val)
{
        boolean found=false;
        while((r!=null) && !found)
        {
                int rval=r.data;
                if(val<rval)
                        r=r.left;
                else if(val>rval)
                        r=r.right;
                else
                {
                        found=true;
```

```
                    break;
                }
                found=search(r,val);
            }
            return found;
        }
        public void inorder()
        {
            inorder(root);
        }
        private void inorder(AVLNode r)
        {
            if(r!=null)
            {
                inorder(r.left);
                System.out.print(r.data+" ");
                inorder(r.right);
            }
        }
        public void preorder()
        {
            preorder(root);
        }
        private void preorder(AVLNode r)
        {
            if(r!=null)
            {
                System.out.println(r.data+" ");
                preorder(r.left);
                preorder(r.right);
            }
        }
        public void postorder()
        {
            postorder(root);
        }
        private void postorder(AVLNode r)
        {
            if(r!=null)
            {
                postorder(r.left);
                postorder(r.right);
                System.out.print(r.data+" ");
            }
        }
    }
```

```
public class AVLTreeTest
{
	public static void main(String args[])
	{
		Scanner scan=new Scanner(System.in);
		AVLTree avlt=new AVLTree();
		System.out.println("AVLTree Tree Test\n");
		char ch;
		do
		{
			System.out.println("\nAVLTree Operations\n");
			System.out.println("1.insert");
			System.out.println("2.search");
			System.out.println("3.count nodes");
			System.out.println("4.check empty");
			System.out.println("5.clear tree");
			int choice=scan.nextInt();
			switch(choice)
			{
			case 1: System.out.println("Enter integer element to insert");
			avlt.insert(scan.nextInt());
			break;
			case 2: System.out.println("Enter integer element to search");
			System.out.println("Search result:"+avlt.search(scan.nextInt()));
			break;
			case 3: System.out.println("Nodes="+avlt.countNodes());
			break;
			case 4: System.out.println("Empty status="+avlt.isEmpty());
			break;
			case 5: System.out.println("\nTree cleared");
			avlt.makeEmpty();
			break;
			default: System.out.println("Wrong entry\n");
			break;
			}
			System.out.print("\n Post order:");
			avlt.postorder();
			System.out.print("\n Pre order:");
			avlt.preorder();
			System.out.print("\n In order:");
			avlt.inorder();
			System.out.println("\nDo you want to continue(Type y or n)\n");
			ch=scan.next().charAt(0);
		}while(ch=='Y'||ch=='y');
	}
}
```

----

output

AVLTree Tree Test
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
1
Enter integer element to insert
30
 Post order:30
 Pre order:30
 In order:30
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
1
Enter integer element to insert
60
 Post order:60 30
 Pre order:30
60
 In order:30 60
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
1
Enter integer element to insert
70
 Post order:30 70 60
 Pre order:60
30
70
 In order:30 60 70

```
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
1
Enter integer element to insert
20
 Post order:20 30 70 60
 Pre order:60
30
20
70
 In order:20 30 60 70
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
1
Enter integer element to insert
25
 Post order:20 30 25 70 60
 Pre order:60
25
20
30
70
 In order:20 25 30 60 70
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
1
Enter integer element to insert
1
```

```
 Post order:1 20 30 70 60 25
 Pre order:25
20
1
60
30
70
 In order:1 20 25 30 60 70
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
50
Wrong entry
 Post order:1 20 30 70 60 25
 Pre order:25
20
1
60
30
70
 In order:1 20 25 30 60 70
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
1
Enter integer element to insert
50
 Post order:1 20 50 30 70 60 25
 Pre order:25
20
1
60
30
50
70
 In order:1 20 25 30 50 60 70
```

```
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
2
Enter integer element to search
30
Search result:true
 Post order:1 20 50 30 70 60 25
 Pre order:25
20
1
60
30
50
70
 In order:1 20 25 30 50 60 70
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
3
Nodes=7
 Post order:1 20 50 30 70 60 25
 Pre order:25
20
1
60
30
50
70
 In order:1 20 25 30 50 60 70
Do you want to continue(Type y or n)
y

AVLTree Operations

1.insert
```

2.search
3.count nodes
4.check empty
5.clear tree
4
Empty status=false
Post order:1 20 50 30 70 60 25
 Pre order:25
20
1
60
30
50
70
 In order:1 20 25 30 50 60 70
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
5
Tree cleared
 Post order:
 Pre order:
 In order:
Do you want to continue(Type y or n)
y
AVLTree Operations
1.insert
2.search
3.count nodes
4.check empty
5.clear tree
4
Empty status=true
 Post order:
 Pre order:
 In order:
Do you want to continue(Type y or n)
n

3.Efficiency of Heap sort and Quicksort

HEAP SORT

```
import java.io.*;
import java.util.*;
class heapsort
{	LinkedList<Integer> list1 = new LinkedList<Integer>();
	LinkedList<Integer> list2 = new LinkedList<Integer>();

	void Read(int n)
	{	int x;
		Scanner in=new Scanner(System.in);
		Random r=new Random();
		 int [] h=new int[100];
		for(int i=1;i<=n;i++)
		{
		h[i]=r.nextInt(100) + (-25)+1;
			list1.addLast(h[i]);
		}
			list1.addLast(r.nextInt(n));
		list1.addFirst(0); //index 0 in list1 is not used
	}
	public void heaps(int n)
	{	int k,v,j,i,heap;
		for(i=n/2;i>=1;i--)
		{	k=i;
			v=list1.get(k);
			heap=0;
			while((heap==0) && ((2*k)<=n))
			{	j=2*k;
				if(j<n) //checking for 2 children
					if(list1.get(j) < list1.get(j+1))
						j++;
				if(v>=list1.get(j))
					heap=1;
				else
				{	list1.set(k,list1.get(j));
					k=j;
				}
			}
			list1.set(k,v);
		}
	}
	public void heapify(int n)
	{	int t;
		if(n==1)
```

```
                {       Display(n);
                        list2.addLast(list1.get(1));
                        list1.removeLast();
                }
                else
                {       heaps(n);//construct the heap using bottom up method.
                        t=list1.get(1);
                        list1.set(1,list1.get(n));
                        list1.set(n,t);
                        list2.addLast(list1.get(n));
                        Display(n);
                        list1.removeLast();
                        heapify(n-1);//reduce the heap size by 1.
                }
        }
        void Display(int n)
        {       int x;
                System.out.println("\n--------------\n");
                for(int i=1;i<=n;i++)
                        System.out.print(list1.get(i)+ "\t");
        }
        void Display1(int n)
        {       int x;
                System.out.println("\nHeapified List is :");
                for(int i=0;i<n;i++)
                        System.out.println(list2.get(i));
        }
}
public class HeapLL
{
        public static void  main (String[] args)
        {     int n;
                heapsort h=new heapsort();
                Scanner in=new Scanner(System.in);
                System.out.println("********** HEAP SORT *********");
                System.out.println("enter the size of heap :");
                n=in.nextInt();
                h.Read(n);
                System.out.println("Heaped elements : ");
                long time = System.currentTimeMillis();
                h.heapify(n);
                long timeNow = System.currentTimeMillis();
                h.Display1(n);
                System.out.println("time: " + (timeNow - time)+"ms");
        }
}
```

----

output

********** HEAP SORT *********
enter the size of heap :
15
Heaped elements :

38	20	51	-13	9	36	48	-22	-24	-17	-19	30	-1
	24	75										

24	20	48	-13	9	36	38	-22	-24	-17	-19	30	-1
	51											

-1 20 38 -13 9 36 24 -22 -24 -17 -19 30 48

-1 20 36 -13 9 30 24 -22 -24 -17 -19 38

-19 20 30 -13 9 -1 24 -22 -24 -17 36

-17 20 24 -13 9 -1 -19 -22 -24 30

-24 20 -1 -13 9 -17 -19 -22 24

-22 9 -1 -13 -24 -17 -19 20

-19 -13 -1 -22 -24 -17 9

-19 -13 -17 -22 -24 -1

-24 -19 -17 -22 -13

-22 -19 -24 -17

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

-24    -22    -19
--------------

-24    -22
--------------

-24
Heapified List is :
75
51
48
38
36
30
24
20
9
-1
-13
-17
-19
-22
-24
time: 15ms
```

QUICK SORT

```
import java.io.*;
import java.util.*;
class Quick
{       LinkedList<Integer> list1 = new LinkedList<Integer>();
Scanner in=new Scanner(System.in);
        int i,j;
     public void Read(int n)
     {    Random r=new Random();
          int [] h=new int[100];
          for(int i=1;i<=n;i++)
         {       h[i]=r.nextInt(100)+(-25) +1;//max 100 and min 25 range i.e -25 to 100
                 list1.addLast(h[i]);
                 list1.addLast(0);
          }
```

```
        }
        public void swap(int m,int n)
        {       int t;
                t=list1.get(m);
                list1.set(m,list1.get(n));
                list1.set(n,t);
        }
        public int partition(int l,int r)
        {       int p,i,j;
                p=list1.get(l);
                i=l+1;
                j=r;

                while(i<=j)
                {       while(list1.get(i)<=p & i<=r)
                                i++;
                        while(list1.get(j)>p)
                                j--;
                        swap(i,j);
                }
                swap(i,j);
                swap(l,j);
                return j;
        }
        public void QSort(int l,int r)
        {       int s;
                if(l<r)
                {       s=partition(l,r);
                        QSort(l,s-1);
                        QSort(s+1,r);
                }
        }
        public void Display(int n)
        {       System.out.print("sorted order :");
                for(i=0;i<n;i++)
                        System.out.print(list1.get(i)+" ");
        }
}
public class QuickLL
{   public static void main(String[] args)
  {             int n;
                Scanner in=new Scanner(System.in);
                Quick q=new Quick();
                System.out.println("********** QUICK SORT *********");
                System.out.println("Enter n :");
                n=in.nextInt();
```

```
                q.Read(n);
                long time = System.currentTimeMillis();
                q.QSort(0,n-1);
                long timeNow = System.currentTimeMillis();
               q.Display(n);
                System.out.println("\ntime: " + (timeNow - time)+"ms");
   }
}
```

Output

********** QUICK SORT *********
Enter n :
20
sorted order :-20 -8 -5 0 0 0 0 0 0 0 0 0 0 2 9 10 19 29 31 42
time: 0ms

4.TSP(Dynamic Programming)

```
import java.util.*;
import java.text.*;

public class TSP
{       int a[][],visited[],n,cost;
        TSP()
        {       cost = 0;
                int i,j;
                a = new int[10][10];
                visited = new int[10];
                Scanner scan = new Scanner(System.in);
                System.out.print("Enter No. of Cities:");
                n = scan.nextInt();
                System.out.println("Enter Cost Matrix:");
                for(i=0;i<n;i++)
                {       for(j=0;j<n;j++)
                        {       if(i!=j)
                                {       System.out.print("Enter distance from "+(i+1)+" to
"+(j+1)+":=>");
                                        a[i][j]=scan.nextInt();
                                }
                        }
                        visited[i] = 0;
                }
                System.out.println();
                System.out.println("Starting node assumed to be node 1.");
                System.out.println("The Cost adjacency matrix is");
                for( i=0;i<n;i++)
                {       System.out.println();
                        for( j=0;j<n;j++)
                                System.out.print("      "+a[i][j]+"      ");
                }
                System.out.println();
        }

        void mincost(int city)
        {
                int i,ncity;
                visited[city]=1;
                System.out.print((city+1)+"->");
                ncity=least(city);
                if(ncity==999)
                {
```

```
                    ncity=0;
                    System.out.println(ncity+1);
                    cost+=a[city][ncity];
                    return;
                }
                mincost(ncity);
        }

        int least(int c)
        {
                int i,nc=999;
                int min=999,kmin=0;


for(i=0;i<n;i++)
                {       if((a[c][i]!=0)&&(visited[i]==0))
                                if(a[c][i]<min)
                                {       min=a[i][0]+a[c][i];
                                        kmin=a[c][i];
                                        nc=i;
                                }
                }
                if(min!=999)
                        cost+=kmin;
                return nc;
        }

        void put()
        {               System.out.println("Minimum cost:"+cost);
        }

        public static void main(String args[])
        {
                TSP t = new TSP();
                System.out.println("The Optimal Path is:");
                t.mincost(0);
                t.put();
        }
}
```

Output

Enter No. of Cities:5
Enter Cost Matrix:
Enter distance from 1 to 2:=>8
Enter distance from 1 to 3:=>1
Enter distance from 1 to 4:=>0

```
Enter distance from 1 to 5:=>0
Enter distance from 2 to 1:=>0
Enter distance from 2 to 3:=>0
Enter distance from 2 to 4:=>0
Enter distance from 2 to 5:=>9
Enter distance from 3 to 1:=>0
Enter distance from 3 to 2:=>0
Enter distance from 3 to 4:=>2
Enter distance from 3 to 5:=>0
Enter distance from 4 to 1:=>0
Enter distance from 4 to 2:=>0
Enter distance from 4 to 3:=>0
Enter distance from 4 to 5:=>5
Enter distance from 5 to 1:=>0
Enter distance from 5 to 2:=>0
Enter distance from 5 to 3:=>0
Enter distance from 5 to 4:=>0

Starting node assumed to be node 1.
The Cost adjacency matrix is

        0       8       1       0       0
        0       0       0       0       9
        0       0       0       2       0
        0       0       0       0       5
        0       0       0       0       0
The Optimal Path is:
1->3->4->5->1
Minimum cost:8
```

5.N-Queens Problem(Backtracking/Branch & Bound)

```
import java.io.*;

import java.util.*;

class Myqueen

{	int[] x=new int[100];
	void display(int n)
	{	char[][] chessboard=new char[20][20];
		int i,j;
		for(i=0;i<n;i++)
			for(j=0;j<n;j++)
				chessboard[i][j]='x';
		for(i=0;i<n;i++)
			chessboard[i][x[i]]='Q';
		for(i=0;i<n;i++)
		{	for(j=0;j<n;j++)
				System.out.print(chessboard[i][j]+" ");
			System.out.println("\n");
		}
		System.out.println("\n************\n");
	}
	int place(int k)
	{	for(int i=0;i<k;i++)
			if(x[i]==x[k] || (Math.abs(x[i]-x[k]) == Math.abs(i-k)))
				return 1;
		return 0;
	}

	public void queen(int n)
	{	int k=0,c=0;
		x[0]=-1;
		while(k>=0)
		{	x[k]=x[k]+1;
			while(x[k]<n & place(k)==1)
				x[k]=x[k]+1;
			if(x[k]<n)
			{	if(k==n-1)
				{	display(n);
					c++;
				}
				else
				{	k++;
					x[k]=-1;
				}
```

```
                    }
                    else
                            k--;
            }
            System.out.println("No. of possibilities for "+n+" queens is :"+c);
            if(k<0 && c==0)
                    System.out.println("\nFailure!!!!!! No solution");
    }
}
public class Nqueen
{
    public static void  main (String[] args)
    {  int n;
            Myqueen m=new Myqueen();
          Scanner in=new Scanner(System.in);
            System.out.println("Enter no of Queens :");
            n=in.nextInt();
            m.queen(n);

    }
}
```

Output

```
Enter no of Queens :
4
x Q x x

x x x Q

Q x x x

x x Q x


************

x x Q x

Q x x x

x x x Q

x Q x x


************
```

```
No. of possibilities for 4 queens is :2
-----------------------------------------------------------------------------
Enter no of Queens :
2
No. of possibilities for 2 queens is :0

Failure!!!!!! No solution
```

6.Bellman-Ford algorithm

```
import java.util.Scanner;
public class BellmanFord
{
private int distances[];
private int numberofvertices;
public static final int MAX_VALUE=999;

public BellmanFord(int numberofvertices)
{
this.numberofvertices=numberofvertices;
distances=new int[numberofvertices+1];
}
public void BellmanFordEvaluation(int source,int adjacencymatrix[][])
{
for(int node=1;node<=numberofvertices;node++)
{
distances[node]=MAX_VALUE;
}
distances[source]=0;
for(int node=1;node<=numberofvertices-1;node++)
{
for(int sourcenode=1;sourcenode<=numberofvertices;sourcenode++)
{
for(int destinationnode=1;destinationnode<=numberofvertices;destinationnode++)
{
if(adjacencymatrix[sourcenode][destinationnode]!=MAX_VALUE)
{
if(distances[destinationnode]>distances[sourcenode]+adjacencymatrix[sourcenode][destinationn
ode])
distances[destinationnode]=distances[sourcenode]+adjacencymatrix[sourcenode][destinationnod
e];
}
}
}
}
for(int sourcenode=1;sourcenode<=numberofvertices;sourcenode++)
{
for(int destinationnode=1;destinationnode<=numberofvertices;destinationnode++)
{
if(adjacencymatrix[sourcenode][destinationnode]!=MAX_VALUE)
{
if(distances[destinationnode]>distances[sourcenode]+adjacencymatrix[sourcenode][destinationn
ode])
System.out.println("The graph contains negative edge cycle");
```

```
}
}
}
for(int vertex=1;vertex<=numberofvertices;vertex++)
{
System.out.println("distance of source "+source+" to "+vertex+" is "+distances[vertex]);
}
}
public static void main(String arg[])
{
int numberofvertices=0;
int source;
Scanner scanner=new Scanner(System.in);
System.out.println("Enter the number of vertices");
numberofvertices=scanner.nextInt();
int adjacencymatrix[][]=new int[numberofvertices+1][numberofvertices+1];
System.out.println("Enter the adjacency matrix");
for(int sourcenode=1;sourcenode<=numberofvertices;sourcenode++)
{
for(int destinationnode=1;destinationnode<=numberofvertices;destinationnode++)
{
adjacencymatrix[sourcenode][destinationnode]=scanner.nextInt();
if(sourcenode==destinationnode)
{
adjacencymatrix[sourcenode][destinationnode]=0;
continue;
}
if(adjacencymatrix[sourcenode][destinationnode]==0)
{
adjacencymatrix[sourcenode][destinationnode]=MAX_VALUE;
}
}
}
System.out.println("Enter the source vertex");
source=scanner.nextInt();
BellmanFord bellmanford=new BellmanFord(numberofvertices);
bellmanford.BellmanFordEvaluation(source,adjacencymatrix);
scanner.close();
}
}
```

Output

Enter the number of vertices
5
Enter the adjacency matrix
0 8 1 0 0

```
0 0 0 0 9
0 0 0 2 0
0 0 0 0 5
0 0 0 0 0
Enter the source vertex
1
distance of source 1 to 1 is 0
distance of source 1 to 2 is 8
distance of source 1 to 3 is 1
distance of source 1 to 4 is 3
distance of source 1 to 5 is 8
```

7.Shortest Paths in a DAG

```
import  java.util.*;
/* An example class for directed graphs.  The vertex type can be specified.
   There are no edge costs/weights.  */
public class acyclic <V> {
    /* The implementation here is basically an adjacency list, but instead  of an array of lists,
        a Map is used to map each vertex to its list of   adjacent vertices.     */

   private Map<V,List<V>> neighbors = new HashMap<V,List<V>>();
       //  String representation of graph.
    public String toString ()    {
        StringBuffer s = new StringBuffer();
        for (V v: neighbors.keySet()) s.append("\n    " + v + " -> " + neighbors.get(v));
        return s.toString();
    }

    //Add a vertex to the graph.  Nothing happens if vertex is    already in graph.
    public void add (V vertex)  {
        if (neighbors.containsKey(vertex)) return;
        neighbors.put(vertex, new ArrayList<V>());
    }
    // True iff graph contains vertex.
     public boolean contains (V vertex)   {
        return neighbors.containsKey(vertex);
    }

    /* Add an edge to the graph; if either vertex does not exist, it's added.
        This implementation allows the creation of multi-edges and self-loops.  */
    public void add (V from, V to)   {
        this.add(from); this.add(to);
        neighbors.get(from).add(to);
    }

    /*  Remove an edge from the graph.  Nothing happens if no such edge.
        @throws IllegalArgumentException if either vertex doesn't exist.     */
    public void remove (V from, V to)    {
        if (!(this.contains(from) && this.contains(to)))
            throw new IllegalArgumentException("Nonexistent vertex");
        neighbors.get(from).remove(to);
    }

   //  Report (as a Map) the out-degree of each vertex.
   public Map<V,Integer> outDegree ()   {
        Map<V,Integer> result = new HashMap<V,Integer>();
        for (V v: neighbors.keySet()) result.put(v, neighbors.get(v).size());
        return result;
```

```
  }

// Report (as a Map) the in-degree of each vertex.
  public Map<V,Integer> inDegree () {
      Map<V,Integer> result = new HashMap<V,Integer>();
      for (V v: neighbors.keySet()) result.put(v, 0);      // All in-degrees are 0
      for (V from: neighbors.keySet())      {
          for (V to: neighbors.get(from))     {
              result.put(to, result.get(to) + 1);         // Increment in-degree
          }
      }
      return result;
  }

  // Report (as a List) the topological sort of the vertices; null for no such sort.
  public List<V> topSort ()     {
      Map<V, Integer> degree = inDegree();         // Determine all vertices with zero in-degree
      Stack<V> zeroVerts = new Stack<V>();        // Stack as good as any here
      for (V v: degree.keySet())      {
          if (degree.get(v) == 0) zeroVerts.push(v);
      }

  // Determine the topological order
      List<V> result = new ArrayList<V>();
      while (!zeroVerts.isEmpty())     {
          V v = zeroVerts.pop();
               // Choose a vertex with zero in-degree result.add(v);
               // Vertex v  is next in topol order
               // "Remove" vertex v by updating its neighbors
          for (V neighbor: neighbors.get(v))      {
              degree.put(neighbor, degree.get(neighbor) - 1);
               // Remember any vertices that now have zero in-degree
                if (degree.get(neighbor) == 0) zeroVerts.push(neighbor);
          }
      }
     // Check that we have used the entire graph (if not, there was a cycle)
      if (result.size() != neighbors.size()) return null;
      return result;
  }

 // True iff graph is a dag (directed acyclic graph).
  public boolean isDag ()    {
      return topSort() != null;
  }
```

```
     /*  Report (as a Map) the bfs distance to each vertex from the start vertex.  The distance is an
Integer;
       the value null is used to represent infinity  (implying that the corresponding node cannot be
reached).
    */
   public Map bfsDistance (V start)  {
      Map<V,Integer> distance = new HashMap<V,Integer>();
     // Initially, all distance are infinity, except start node
      for (V v: neighbors.keySet()) distance.put(v, null);
      distance.put(start, 0);                    // Process nodes in queue order
      Queue<V> queue = new LinkedList<V>();
      queue.offer(start);                        // Place start node in queue
      while (!queue.isEmpty())    {
         V v = queue.remove();
         int vDist = distance.get(v);
         // Update neighbors
         for (V neighbor: neighbors.get(v))       {
            if (distance.get(neighbor) != null) continue;
              // Ignore if already done
            distance.put(neighbor, vDist + 1);
            queue.offer(neighbor);
         }
      }
      return distance;
   }
                          // Main program (for testing).
 public static void main (String[] args)  {
         // Create a Graph with Integer nodes
            acyclic<Integer> graph = new acyclic<Integer>();
      graph.add(0, 1); graph.add(0, 2); graph.add(0, 3);
      graph.add(1, 2); graph.add(1, 3); graph.add(2, 3);
      graph.add(2, 4); graph.add(4, 5); graph.add(5, 6);      // Tetrahedron with tail
      System.out.println("The current graph: " + graph);
      System.out.println("In-degrees: " + graph.inDegree());
      System.out.println("Out-degrees: " + graph.outDegree());
      System.out.println("A topological sort of the vertices: " + graph.topSort());
      System.out.println("The graph " + (graph.isDag()?"is":"is not") + " a dag");
      System.out.println("BFS distances starting from " + 0 + ": " + graph.bfsDistance(0));
      System.out.println("BFS distances starting from " + 1 + ": " + graph.bfsDistance(1));
      System.out.println("BFS distances starting from " + 2 + ": " + graph.bfsDistance(2));
                                       //  graph.add(4, 1);
                                      // Create a cycle
      System.out.println("Cycle created");
      System.out.println("The current graph: " + graph);
      System.out.println("A topological sort of the vertices: " + graph.topSort());
      System.out.println("The graph " + (graph.isDag()?"is":"is not") + " a dag");
```

```
        System.out.println("BFS distances starting from " + 2 + ": " + graph.bfsDistance(2));
    }
}
```

Output

```
The current graph:
    0 -> [1, 2, 3]
    1 -> [2, 3]
    2 -> [3, 4]
    3 -> []
    4 -> [5]
    5 -> [6]
    6 -> []
In-degrees: {0=0, 1=1, 2=2, 3=3, 4=1, 5=1, 6=1}
Out-degrees: {0=3, 1=2, 2=2, 3=0, 4=1, 5=1, 6=0}
A topological sort of the vertices: null
The graph is not a dag
BFS distances starting from 0: {0=0, 1=1, 2=1, 3=1, 4=2, 5=3, 6=4}
BFS distances starting from 1: {0=null, 1=0, 2=1, 3=1, 4=2, 5=3, 6=4}
BFS distances starting from 2: {0=null, 1=null, 2=0, 3=1, 4=1, 5=2, 6=3}
Cycle created
The current graph:
    0 -> [1, 2, 3]
    1 -> [2, 3]
    2 -> [3, 4]
    3 -> []
    4 -> [5]
    5 -> [6]
    6 -> []
A topological sort of the vertices: null
The graph is not a dag
BFS distances starting from 2: {0=null, 1=null, 2=0, 3=1, 4=1, 5=2, 6=3}
```

8.Ford-Fulkerson Algorithm

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class FordFulkerson
{
   private int[] parent;
   private Queue<Integer> queue;
   private int numberOfVertices;
   private boolean[] visited;

   public FordFulkerson(int numberOfVertices)
   {
      this.numberOfVertices = numberOfVertices;
      this.queue = new LinkedList<Integer>();
      parent = new int[numberOfVertices + 1];
      visited = new boolean[numberOfVertices + 1];
   }

   public boolean bfs(int source, int goal, int graph[][])
   {
      boolean pathFound = false;
      int destination, element;

      for(int vertex = 1; vertex <= numberOfVertices; vertex++)
      {
         parent[vertex] = -1;
         visited[vertex] = false;
      }

      queue.add(source);
      parent[source] = -1;
      visited[source] = true;

      while (!queue.isEmpty())
      {
         element = queue.remove();
         destination = 1;

         while (destination <= numberOfVertices)
         {
            if (graph[element][destination] > 0 &&  !visited[destination])
            {
               parent[destination] = element;
               queue.add(destination);
```

```
                visited[destination] = true;
            }
            destination++;
        }
    }
    if(visited[goal])
    {
        pathFound = true;
    }
    return pathFound;
}

public int fordFulkerson(int graph[][], int source, int destination)
{
    int u, v;
    int maxFlow = 0;
    int pathFlow;

    int[][] residualGraph = new int[numberOfVertices + 1][numberOfVertices + 1];
    for (int sourceVertex = 1; sourceVertex <= numberOfVertices; sourceVertex++)
    {
        for (int destinationVertex = 1; destinationVertex <= numberOfVertices;
destinationVertex++)
        {
            residualGraph[sourceVertex][destinationVertex] =
graph[sourceVertex][destinationVertex];
        }
    }

    while (bfs(source ,destination, residualGraph))
    {
        pathFlow = Integer.MAX_VALUE;
        for (v = destination; v != source; v = parent[v])
        {
            u = parent[v];
            pathFlow = Math.min(pathFlow, residualGraph[u][v]);
        }
        for (v = destination; v != source; v = parent[v])
        {
            u = parent[v];
            residualGraph[u][v] -= pathFlow;
            residualGraph[v][u] += pathFlow;
        }
        maxFlow += pathFlow;
    }
```

```
        return maxFlow;
    }

    public static void main(String...arg)
    {
        int[][] graph;
        int numberOfNodes;
        int source;
        int sink;
        int maxFlow;

        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the number of nodes");
        numberOfNodes = scanner.nextInt();
        graph = new int[numberOfNodes + 1][numberOfNodes + 1];

        System.out.println("Enter the graph matrix");
        for (int sourceVertex = 1; sourceVertex <= numberOfNodes; sourceVertex++)
        {
           for (int destinationVertex = 1; destinationVertex <= numberOfNodes;
destinationVertex++)
           {
              graph[sourceVertex][destinationVertex] = scanner.nextInt();
           }
        }

        System.out.println("Enter the source of the graph");
        source= scanner.nextInt();

        System.out.println("Enter the sink of the graph");
        sink = scanner.nextInt();

        FordFulkerson fordFulkerson = new FordFulkerson(numberOfNodes);
        maxFlow = fordFulkerson.fordFulkerson(graph, source, sink);
        System.out.println("The Max Flow is " + maxFlow);
        scanner.close();
    }
}
```

Output

```
Enter the number of nodes
5
Enter the graph matrix
0 8 1 0 0
0 0 0 0 9
0 0 0 2 0
```

0 0 0 0 5
0 0 0 0 0
Enter the source of the graph
1
Enter the sink of the graph
5
The Max Flow is 9

9.Robin-Karp Algorithm

```
import java.io.*;
import java.util.*;
public class RobinKarp
{
	String text = null, pattern = null;
	int m,n,p,q;
	int flag=0;
	public void preprocessing()
	{	m= pattern.length();
		n= text.length();
		q=11;
		p=Integer.parseInt(pattern)%q ;
	}

	public void string_match()
	{
		System.out.println( "Enter the Numeric String :");
		Scanner in=new Scanner(System.in);
		text=in.nextLine();
		System.out.print("Enter the pattern to be searched:\n ");
		pattern = in.nextLine();
		preprocessing();
		int i=0,rem;
		for(int s=0;s<=n-m;s++)
		{	i=0;
			rem=Integer.parseInt(text.substring(s,s+m))%q;
			 // performs mod operation on the substring of size m
			if (p==rem)
			{	while(i<m &&text.charAt(s+i)==pattern.charAt(i))
				i++;
				if (i==m)
				{	System.out.print("\n SUCCESS!!! The pattern is found at
position " + (s+1));
					flag=1;
				}
			}
		}
		if(flag==0)
		{
		System.out.print("FAILURE!!!! \nThe pattern "+pattern+" is not found in the
text");
		}
	}
	public static void main(String args[]) throws IOException
	{	RobinKarp r = new RobinKarp();
```

```
            r.string_match();
        }
}
```

Output

Enter the Numeric String :
28122014
Enter the pattern to be searched:
 12

 SUCCESS!!! The pattern is found at position 3

10.Knuth-Morris-Pratt Algorithms

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class KnuthMorrisPratt
{
	private int[] failure;
	public KnuthMorrisPratt(String text, String pat)
	{
		failure=new int[pat.length()];
		fail(pat);
		int pos=postMatch(text,pat);
		if(pos==-1)
			System.out.println("\nNo match found");
		else
			System.out.println("\nMatch found at index "+pos);
	}
	private void fail(String pat)
	{
		int n=pat.length();
		failure[0]=-1;
		for(int j=1;j<n;j++)
		{
			int i= failure[j-1];
			while((pat.charAt(j)!=pat.charAt(i+1))&& i>=0)
				i=failure[i];
			if(pat.charAt(j)==pat.charAt(i+1))
				failure[j]=i+1;
			else
				failure[j]=-1;
		}
	}
	private int postMatch(String text,String pat)
	{
		int i=0,j=0;
		int lens=text.length();
		int lenp=pat.length();
		while(i<lens && j<lenp)
		{
			if(text.charAt(i)==pat.charAt(j))
			{
				i++;
				j++;
			}
```

```
                    else if(j==0)
                            i++;
                    else
                            j=failure[j-1]+1;
            }
            return((j==lenp)?(i-lenp):-1);
        }
        public static void main(String[] args) throws IOException
        {
            BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
            System.out.println("KnuthMorrisPratt text");
            System.out.println("Enter text\n");
            String text=br.readLine();
            System.out.println("Enter pattern");
            String pattern=br.readLine();
            KnuthMorrisPratt kmp=new KnuthMorrisPratt(text,pattern);
        }
}
```

Output

KnuthMorrisPratt text
Enter text
Ashwini c k
Enter pattern
win
Match found at index 3

11.String Matching with Finite Automata

```
import java.util.BitSet;

import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

class State<E>
{
   private static int nextStateNum = 0;
   private final int num = nextStateNum++;
   public final BitSet positions;
   public final Map<E,State<E>> transitions = new HashMap<E,State<E>>();
   public boolean finale;


   public State(BitSet bs) { positions = bs; }
   public String toString() { return Integer.toString(num); }
}

public class DFAStringSearch<E>
{
   // maps the set of string positions a state represents to the state
   private final Map<BitSet, State<E>> stateMap = new HashMap<BitSet, State<E>>();
   // list of states in order of creation
   private final List<State<E>> states = new ArrayList<State<E>>();

   public State<E> initialState;

   public DFAStringSearch(E[] pattern)
   {
      BitSet initialPos = new BitSet();
         initialPos.set(0);
         initialState = getState(initialPos);
      for (int i = 0; i < states.size(); i++)
      {
         State<E> s = states.get(i);
         for (int j = s.positions.nextSetBit(0); j >= 0; j = s.positions.nextSetBit(j+1))
         {
            if (j == pattern.length)
                  {
                     s.finale = true;
                     break;
                  }
            E cNext = pattern[j];
```

```
            if (!s.transitions.containsKey(cNext))
               fillTransitionTableEntry(pattern, s, cNext);
         }
      }
   }
   public State<E> getState(BitSet s)
   {
      if (stateMap.containsKey(s))
         return stateMap.get(s);
      else
      {
         State<E> st = new State<E>(s);
         stateMap.put(s, st);
         states.add(st);
         return st;
      }
   }
   private void fillTransitionTableEntry(E[] pattern, State<E> s, E cNext)
   {
      BitSet newPositions = new BitSet();
      newPositions.set(0);
      for (int i = s.positions.nextSetBit(0); i >= 0 && i < pattern.length; i =
s.positions.nextSetBit(i+1))
      {
         if (pattern[i].equals(cNext))
            newPositions.set(i + 1);
      }
      s.transitions.put(cNext, getState(newPositions));
      System.err.println("Adding edge " + s + " -" + cNext + "-> " + s.transitions.get(cNext));
   }
   public int search(E[] searchFor, E[] searchIn)
   {
      State<E> curState = initialState;
      int curPos;
      for (curPos = 0; curPos < searchIn.length && !curState.finale; curPos++)
      {
         curState = curState.transitions.get(searchIn[curPos]);
         if (curState == null)
            curState = initialState;
      }
      if (curState.finale)
         return curPos - searchFor.length;
      else
         return -1;
   }
   private static Character[] str2charArray(String str) {
```

```
        Character[] result = new Character[str.length()];
        for (int i = 0; i < str.length(); i++)
            result[i] = str.charAt(i);
        return result;
    }

    public static void main(String[] args)
    {
            String s1="abcd";
            String s2="hbabcdhbsjdd";
        Character[] a = str2charArray(s1), b = str2charArray(s2);
        DFAStringSearch<Character> foo = new DFAStringSearch<Character>(a);
        int result = foo.search(a, b);
        if (result == -1)
            System.out.println("No match found.");
        else
        {
            System.out.println("Matched at position " + result + ":");
            System.out.println(s2.substring(0, result) + "|" + s2.substring(result));
        }
    }

}
```

Output

```
Adding edge 0 -j-> 1
Adding edge 1 -j-> 1
Adding edge 1 -d-> 2
Adding edge 2 -j-> 1
Adding edge 2 -d-> 3
Adding edge 3 -j-> 1
Matched at position 9:
hbabcdhbs|jdd
```

12.Vertex Cover Problem

```
import java.util.HashSet;
 import java.util.Set;

public class VertexCover {
private static final char[] name_vertex = { 'A', 'B', 'C', 'D', 'E', 'F', 'G'};


/* // Input Number One
  private static final int[][] matrix = {
    //A  B  C  D  E
    { 0, 1, 0, 0, 0 },// A
    { 1, 0, 1, 0, 0 },// B
    { 0, 1, 0, 1, 1 },// C
    { 0, 0, 1, 0, 1 },// D
    { 0, 0, 1, 1, 0 },// E
  };
 */
  // Input Number Two
  // Cormen , Introduction to Algo , Chap 35.1 , Approx Algo , Pg 1109,
  private static final int[][] matrix = {
    //A  B  C  D  E  F  G
    { 0, 1, 0, 0, 0, 0, 0 },// A
    { 1, 0, 1, 0, 0, 0, 0 },// B
    { 0, 1, 0, 1, 1, 0, 0 },// C
    { 0, 0, 1, 0, 1, 1, 1 },// D
    { 0, 0, 1, 1, 0, 1, 0 },// E
    { 0, 0, 0, 1, 1, 0, 0 },// F
    { 0, 0, 0, 1, 0, 0, 0 },// G
  };

 private static final int no_vertices = matrix[0].length;

 private static final boolean arr[] = new boolean[no_vertices];

 private static void printEnabledVertices(String s) {
   for (int i = 0; i < no_vertices; i++) {
    if (arr[i] == true) { // Vertices chosen for this iteration
     System.out.print(" " + name_vertex[i]);
    }
   }
   System.out.println("");
   pickMinimum();// Written separately :)
  }

 private static void checkVertexCover() {
```

```
  int count = 0;
  for (int i = 0; i < no_vertices; i++) { // Check the graph Matrix
   for (int j = 0; j < i; j++) {
    if (matrix[i][j] == 1) { // Check this edge
     if (arr[i] || arr[j]) { // u or v or both in cover
      count++;
     } else {
      return; // case u and v don't cover an edge
     }
    }
   }
  }
  if (count > 0) {
   printEnabledVertices(null);
  }
 }

private static void calcVertexCover(int index) {
  if (index == (-1)) {
   checkVertexCover();
  } else {
   arr[index] = false;
   calcVertexCover(index - 1);
   arr[index] = true;
   calcVertexCover(index - 1);
  }
 }

public static void main(String args[]) {
  System.out.println("\n\n Vertex Covers Are");
  System.out.println("-----------------------");
  calcVertexCover(no_vertices - 1);
  printMinimum();
 }

/*CODE TO PICK MINIMUM PLEASE OPTIMIZE BY COMBINING LOOPS  */

private static int min_cover_vertices = no_vertices;

private static Set<String> min_cover = new HashSet<String>();

private static String getVertexString() {
  StringBuffer s = new StringBuffer();
  for (int i = 0; i < no_vertices; i++) {
   if (arr[i] == true) { // Vertices chosen for this iteration
    s.append(" " + name_vertex[i]);
```

```
    }
   }
  return s.toString();
  }
 private static void pickMinimum() { // This function Can be optimized
  int count = 0;
  for (int i = 0; i < no_vertices; i++) {
   if (arr[i] == true) { // Vertices chosen for this iteration
    count++;
   }
  }
  if (count > 0) {
   if (min_cover_vertices == count) {
    min_cover.add(getVertexString());
   } else if (min_cover_vertices > count) {
    min_cover_vertices = count;
    min_cover.clear();
    min_cover.add(getVertexString());
   }
  }
 }
 private static void printMinimum() {
  if (min_cover.size() > 0) {
   System.out.println("\n\n Minimum Covers Are");
   System.out.println("----------------------");
   for (String s : min_cover) {
    System.out.println(s);
   }
  }
 }
}
```

---

output

```
Vertex Covers Are
----------------------
 B D E
 A B D E
 A C D E
 B C D E
 A B C D E
 A C D F
 B C D F
 A B C D F
 B D E F
 A B D E F
 A C D E F
```

B C D E F
A B C D E F
B D E G
A B D E G
A C D E G
B C D E G
A B C D E G
A C D F G
B C D F G
A B C D F G
A C E F G
B C E F G
A B C E F G
B D E F G
A B D E F G
A C D E F G
B C D E F G
A B C D E F G

Minimum Covers Are

B D E

13.The Set Covering problem

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;

public class SetCover {
interface Filter<T> {
boolean matches(T t);
}
public static void main(String... args) throws IOException {
Integer[][] arrayOfSets = {
        {1, 2, 3, 8, 9, 10},
        {1, 2, 3, 4, 5},
        {4, 5, 7},
        {5, 6, 7},
        {6, 7, 8, 9, 10},
  };
  Integer[] solution = {1,2,3,4,5,6,7,8,9,10};
  List<Set<Integer>>listOfSets = new ArrayList<Set<Integer>>();
  for (Integer[] array : arrayOfSets)
        listOfSets.add(new LinkedHashSet<Integer>(Arrays.asList(array)));
  final Set<Integer>solutionSet = new LinkedHashSet<Integer>(Arrays.asList(solution));

  Filter<Set<Set<Integer>>> filter = new Filter<Set<Set<Integer>>>() {
  public boolean matches(Set<Set<Integer>> integers) {
        Set<Integer> union = new LinkedHashSet<Integer>();
        for (Set<Integer>ints : integers)
              union.addAll(ints);
        return union.equals(solutionSet);
     }
  };
  Set<Set<Integer>>firstSolution = shortestCombination(filter, listOfSets);
  System.out.println("The shortest combination was "+firstSolution);
        }

        private static <T> Set<T>shortestCombination(Filter<Set<T>> filter, List<T>listOfSets)
{
              final int size = listOfSets.size();
              if (size > 20) throw new IllegalArgumentException("Too many combinations");
              int combinations = 1 << size;
              List<Set<T>>possibleSolutions = new ArrayList<Set<T>>();
```

```
                for(int l = 0;l<combinations;l++) {
    Set<T> combination = new LinkedHashSet<T>();
    for(int j=0;j<size;j++) {
      if (((l >> j) & 1) != 0)
                combination.add(listOfSets.get(j));
    }
    possibleSolutions.add(combination);
  }
  // the possible solutions in order of size.
                Collections.sort(possibleSolutions, new Comparator<Set<T>>() {
                public int compare(Set<T> o1, Set<T> o2) {
                return o1.size()-o2.size();
    }
  });
      for (Set<T>possibleSolution : possibleSolutions) {
      if (filter.matches(possibleSolution))
      return possibleSolution;
  }
      return null;
}
}
```

---

output

The shortest combination was [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]]

14.The Subset-Sum Problem

```
import java.io.*;
 import java.util.*;

 class set
  {
        int n,d,sum,i,flag=0;
        int S[]=new int[100];
        int x[]=new int[100];
        public void Read_Find()
          {
                Scanner in=new Scanner(System.in);
                System.out.println("********** SUBSET-SUM PROBLEM *********");
                System.out.println("Enter the size of set :");
                n=in.nextInt();
                System.out.println("Enter set elements in increasing order: ");
                for(int i=1;i<=n;i++)
                        S[i]=in.nextInt();
                System.out.println("Enter maximum limit: ");
                d=in.nextInt();
                System.out.println("The subsets which forms sum "+d+" are:");
                for( i=1;i<=n;i++)
                        sum=sum+S[i];
                if(sum>=d)
                        sumofsub(0,1,sum);
                if(flag==0)
                                System.out.println("{}");
          }



        public void sumofsub(int s,int k,int r)
          {
                x[k]=1;
                if(s+S[k]==d)
                  {
                        System.out.print("{ ");
                        for(int i=1;i<=n;i++)
                                if(x[i]==1)
                                    {
                                        System.out.print(S[i]+" ");
                                        flag=1;
                                    }
                        System.out.println("}");
```

```
                }
                else
                {
                        if(s+S[k]+S[k+1]<=d)
                        {
                                sumofsub(s+S[k],k+1,r-S[k]);
                                x[k+1]=0;
                        }
                        if((s+r-S[k]>=d)&&(s+S[k+1]<=d))
                        {
                                x[k]=0;
                                sumofsub(s,k+1,r-S[k]);
                                x[k+1]=0;
                        }
                }
        }
}

class SubSet
{       public static void  main (String[] args)
        {
                set s=new set();
                s.Read_Find();
        }
}
```

---

output

```
********** SUBSET-SUM PROBLEM *********
Enter the size of set :
4
Enter set elements in increasing order:
1
2
3
4
Enter maximum limit:
3
The subsets which forms sum 3 are:
{ 1 2 }
{ 3 }
```

15.Maximum Bipartite Algorithm

```
import java.io.*;
import java.util.Set;
import java.util.HashSet;


/**
 * Implementation of maximum bipartite matching algorithm. Transcribe the labeling-flipping
 * implementation of augumenting-path algorithm on the adjacency matrix representation of
 * graph. Current implementation assumes undirected, bipartite graph.
 */
public class MaxBipartite {
  int adjacency [][];                  // the adjacency matrix of the graph. 1=edge, 0=no edge.
  private final int MATCH = 2;         // 1* in original algorithm
  private final int NOT_MATCH = 1;
  private final int NOT_LABELED = -1;
  private final int POUND_LABELED = -2;  // # label in original algorithm
  private int rows;
  private int cols;
  private int colLabel [];             // labeling flag of columns
  private int rowLabel [];             // labeling flag of rows
  private boolean colScan [];          // scaning flag of columns
  private boolean rowScan [];          // scaning flag of rows

  /**
   * Constructor given the adjacency matrix in "reduced" form.
   */
  public MaxBipartite(int adj [][]) {
    rows = adj.length;
    cols = adj[0].length;
    adjacency = new int [rows][cols];
    // make an identical copy of the given adjacency matrix
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++)
        adjacency [i][j] = adj [i][j];
    }
    colLabel = new int [cols];
    rowLabel = new int [rows];
    colScan = new boolean [cols];
    rowScan = new boolean [rows];
  }

  /**
   * Constructor given a full square adjacency matrix.
   */
```

```
public MaxBipartite (double adj [][]) {
  rows = adj.length;
  cols = adj[0].length;
  adjacency = new int [rows][cols];
  // convert the given ajdacency matrix of double to an upper triangle adjacency matrix of int.
  for (int i = 0; i < rows; i++) {
    for (int j = i+1; j < cols; j++) {
      if (adj [i][j] == 1.0)
        adjacency [i][j] = NOT_MATCH;
    }
  }
  colLabel = new int [cols];
  rowLabel = new int [rows];
  colScan = new boolean [cols];
  rowScan = new boolean [rows];
}

/**
 * The access point to start the match computation. As mentioned in the algorithm, this
 * algorithm needs to start with some matching. Although empty set is a match, it is
 * desirable to start with a match of bigger size. A simple greedy search is applied to
 * scan row-by-row and find "1" which column id isn't identical to previously selected
 * 1s.
 */
public void match () {
  // the accumulating set of column IDs of "1" already selected into the initial matching.
  Set usedIdx = new HashSet ();
  for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
      if (adjacency [i][j] == NOT_MATCH) {
        String idx = String.valueOf (j);
        if (!usedIdx.contains (idx)) {
          adjacency [i][j] = MATCH;
          usedIdx.add (idx);
          break;// only one needed for each row, break out to next row.
        }
      }
    }
  }
  resetFlags ();
  label ();
}

/**
 * Labeling phase. It's a little tricky to control the flow as meant in the algorithm.
 */
```

```
  protected void label () {
    // column scanning
    boolean colScanDone = true;// flow control flag. true = no column exists as labeled but not
scanned
    for (int i = 0; i < cols; i++) {
      if (colLabel [i] != NOT_LABELED && !colScan [i]) {
        colScanDone = false;
        for (int j = 0; j < rows; j++) {
          if (adjacency [j][i] == NOT_MATCH && rowLabel [j] == NOT_LABELED)
            rowLabel [j] = i;
        }
        colScan [i] = true;
      }
    }
    // row scanning
    boolean rowScanDone = true;// flow control flag, similar to colScanDone.
    int freeRow = -1;// this number is used to control the flow. As described in the algorithm,
when a labeled row contains no 1*, should exit from this sub and go to flipping(). I call this row
"freeRow". A value of -1 indicates no free row is found and should continue at column scanning.
Otherwise, go to flipping().
    for (int i = 0; i < rows; i++) {
      if (rowLabel [i] != NOT_LABELED && !rowScan [i]) {
        rowScanDone = false;
        boolean foundMatch = false;// flag to remember if 1* is found in this row
        for (int j = 0; j < cols; j++) {
          if (adjacency [i][j] == MATCH) {
            colLabel [j] = i;
            foundMatch = true;
            break;// found 1*, exit from this row scaning
          }
        }
        rowScan [i] = true;
        if (!foundMatch) {// no 1* found in this row, should (prematurely) exit from labeling
phase and go to flipping phase
          freeRow = i;
          break;
        }
      }
    }
    if (freeRow != -1)
      flipping (freeRow);// go to flipping phase
    else {
      if (colScanDone && rowScanDone) // algorithm finished
        return;
      else
        label();// otherwise, recursively continue column labeling without reset the flags
```

```
  }
}

/**
 * Flipping phase
 */
protected void flipping (int freeRow) {
  int c = rowLabel [freeRow];
  adjacency [freeRow][c] = MATCH;
  int r = colLabel [c];
  if (r == POUND_LABELED) {// # labeled, go back to labeling phase with all flags reset.
    resetFlags ();
    label ();
  }
  else {// otherwise, recursively flip the labeled row
    adjacency [r][c] = NOT_MATCH;
    flipping (r);
  }
}

/**
 * Reset the labeling of rows and columns. This is a separate sub because it is called
 * not only at the beginning of labeling, but also before flipping exit back to labeling.
 */
protected void resetFlags () {
  // reset row scan/label flags
  for (int i = 0; i < rows; i++) {
    rowLabel [i] = NOT_LABELED;
    rowScan [i] = false;
  }
  // reset column scan/label flags
  for (int i = 0; i < cols; i++) {
    colScan [i] = false;
    colLabel [i] = POUND_LABELED;// labeled # as default
    for (int j = 0; j < rows; j++) {
      if (adjacency [j][i] == MATCH) {// find 1* (match), remove label
        colLabel [i] = NOT_LABELED;
        break;
      }
    }
  }
}

/**
 * Return the new adjacency matrix which contains the solution of maximum matching.
 * Remember that an entry of 2 means the corresponding edge is in the solution.
```

```
 */
public int [][] getAdj () {
  return adjacency;
}

/**
 * Utility function. Post process the adjacency matrix and return the matching number |M|.
 */
public int countMatch () {
  System.err.println ("The maximum matching is as follows:");
  int count = 0;
  for (int i = 0; i< rows; i++) {
    for (int j = 0; j< cols; j++) {
      if (adjacency[i][j] == MATCH) {
        count ++;
        System.err.println ("("+i+","+j+")");
      }
    }
  }
  return count;
}

/**
 * Take example input, find maximum matching and output.
 */
public static void testCase (int adj [][]) {
  MaxBipartite mbm = new MaxBipartite (adj);
  mbm.match ();
  System.out.println ("Matching Number = " + mbm.countMatch ());
  System.out.println ("Matching matrix:");
  int match[][] = mbm.getAdj ();
  int rows = match.length;
  int cols = match[0].length;
  System.out.println ("\t0\t1\t2");
  System.out.println ("\t_\t_\t_");
  for (int i = 0; i < rows; i++) {
    System.out.print (i+"|\t");
    for (int j = 0; j < cols; j++)
      System.out.print (match [i][j] + "\t");
    System.out.println ();
  }
}


public static void main(String args []) throws Exception {
        System.out.println("The adjacency matrix are ");
```

```
    int adj[][] = {{1,1,1}, {1,0,0}, {1,0,1}};
    for(int i=0;i<adj.length;i++)
    {
        System.out.println("\n");
        for(int j=0;j<adj.length;j++)
            System.out.print(adj[i][j]+" \t");
    }

    System.out.println("\n");
        testCase (adj);
        }

}
```

----

output

The adjacency matrix are

```
1    1    1

1    0    0

1    0    1
```

Matching Number = 3

Matching matrix:

```
     0    1    2
     _    _    _
0|   1    2    1
1|   2    0    0
2|   1    0    2
```

The maximum matching is as follows:

(0,1)

(1,0)

(2,2)