

**EXPERIMENT 1A****PROGRAM STATEMENT**

Create a Java class called *Student* with the following details as variables within it.

- (i) USN
- (ii) Name
- (iii) Branch
- (iv) Phone

Write a Java program to create *nStudent* objects and print the USN, Name, Branch, and Phone of these objects with suitable headings.

**CONCEPT**

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. This type can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class.

A class is declared using **class** keyword. A class contains both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**.

**A simple class example**

```
class Student{  
    String USN,name , branch;  
    int phono;  
}
```

Object is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object encapsulates state and behavior.

An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables.

Creating variables of your class type is similar to creating variables of primitive data types, such as integer or float. Each time you create an object, a new set of instance variables comes into existence which defines the characteristics of that object. If you want to create an object of the class and have the reference variable associated with this object, you must also allocate memory for the object by using the **new operator**. This process is called instantiating an object or creating an object instance. In following statement **obj** is instance/object of Student class.

```
Student obj=new Student();
```

An array of objects is created just like an array of primitive type data items in the following way.

```
Student[] obj_Array = new Student[7];
```

However, in this particular case, we may use a for loop since all Student objects are created with the same default constructor.

```
for ( int i=0; i<obj_Array.length; i++) {  
    obj_Array[i]=new Student();  
}
```

**Constructor**

Constructor in java is a *special type of method* that is used to initialize the object. Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

**Types of java constructors**

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

A constructor that have no parameter is known as default constructor.

```
Student()  
{ //block of code  
}
```

Object creation:

```
Student obj=new Student();
```

Constructor with arguments is known as parameterized constructor.

```
Student(int i,String n){  
id = i;  
name = n;  
}
```

Object creation:

```
Student4 s1 = new Student4(111,"Karan");
```

**PROGRAM**

```
import java.util.Scanner;  
import java.io.*;  
public class student  
{ String USN,name , branch;  
int phoneno;  
public void getinfo() throws Exception  
{ InputStreamReader r=new InputStreamReader(System.in);  
BufferedReader br=new BufferedReader(r);  
System.out.println("Enter USN");  
USN=br.readLine();  
System.out.println("Enter Name");  
name=br.readLine();  
System.out.println("Enter Branch");  
branch=br.readLine();  
Scanner integer=new Scanner(System.in);  
System.out.println("Enter phone number");  
phoneno=integer.nextInt();  
}  
public void display()  
{ System.out.printf("%s\t\t%s\t\t%s\t\t%d\n",USN,name,branch,phoneno);  
}  
public static void main(String[] args) throws Exception  
{ int n,i;  
Scanner integer=new Scanner(System.in);
```

```
System.out.println("Enter Number of student");
n=integer.nextInt();
//declare object with array
student[] obj=new student[n];
//object creation
for(i=0;i<n;i++)
{ obj[i]=new student();
System.out.printf("Student : %d\n",i+1);
obj[i].getinfo();
}
System.out.println("USN\tName\tBranch\tPhone Number");
for(i=0;i<n;i++)
obj[i].display();
}
}
```

**OUTPUT:**

```
lab06@cmrit-ThinkCentre-M70E: ~/DAA/lab/1a
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/1a$ javac student.java
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/1a$ java student
Enter Number of student
2
Student : 1
Enter USN
1
Enter Name
Geethu
Enter Branch
CS
Enter phone number
1234567890
Student : 2
Enter USN
2
Enter Name
manu
Enter Branch
ce
Enter phone number
1234567890
USN      Name      Branch      Phone Number
1        Geethu    CS          1234567890
2        manu     ce          1234567890
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/1a$
```

**EXPERIMENT 1B****PROGRAM STATEMENT**

Write a Java program to implement the Stack using arrays. Write Push(), Pop(), and Display() methods to demonstrate its working.

**CONCEPT**

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. This type can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class. A class is declared using **class** keyword. A class contains both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**.

**A simple class example**

```
class Student{  
String USN,name , branch;  
int phoneno;  
}
```

Object is an instance of a class created using a new operator. The new operator returns a reference to a new instance of a class. This reference can be assigned to a reference variable of the class. The process of creating objects from a class is called instantiation. An object encapsulates state and behavior. An object reference provides a handle to an object that is created and stored in memory. In Java, objects can only be manipulated via references, which can be stored in variables. Creating variables of your class type is similar to creating variables of primitive data types, such as integer or float. Each time you create an object, a new set of instance variables comes into existence which defines the characteristics of that object. If you want to create an object of the class and have the reference variable associated with this object, you must also allocate memory for the object by using the **new operator**. This process is called instantiating an object or creating an object instance. In following statement **obj** is instance/object of Student class.

```
Student obj=new Student();
```

An array of objects is created just like an array of primitive type data items in the following way.

```
Student[] obj_Array = new Student[7];
```

However, in this particular case, we may use a for loop since all Student objects are created with the same default constructor.

```
for ( int i=0; i<obj_Array.length; i++)  
{  
obj_Array[i]=new Student();  
}
```

**Constructor**

Constructor in java is a *special type of method* that is used to initialize the object. Java constructor is *invoked at the time of object creation*. It constructs the values i.e. provides data for the object that is why it is known as constructor.

**Types of java constructors**

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

A constructor that have no parameter is known as Default Constructor.

```
Student()  
{ //block of code  
}
```

**Object Creation:**

```
Student obj=new Student();
```

Constructor with arguments is known as parameterized constructor.

```
Student(int i,String n){  
id = i;  
name = n;  
}
```

Object creation:

```
Student4 s1 = new Student4(111,"Karan");
```

**PROGRAM**

```
/*stack.java*/  
import java.io.*;  
import java.util.Scanner;  
public class stack  
{ int element,maxsize,top;  
int[] st;  
public stack()  
{ Scanner integer=new Scanner(System.in);  
System.out.println("Enter stack size");  
maxsize=integer.nextInt();  
st=new int[maxsize];  
top=-1;  
}  
public void push(int element)  
{ /*if(top>=maxsize)  
{ System.out.println("Overflow!!");  
//return(0);  
}*/  
try  
{ st[++top]=element;  
} catch(ArrayIndexOutOfBoundsException e)  
{ System.out.println(e);  
}  
}  
public int pop()  
{ if(top== -1)  
{ System.out.println("UnderFlow");  
return(-1);
```

```
}
return(st[top--]);
}
public void display(int[] st, int max_size)
{ int i;
System.out.println("Stack Elements:");
for(i=0;i<=max_size;i++)
System.out.println(st[i]);
}
} /*myStack.java*/
import java.io.*;
import java.util.Scanner;
public class myStack
{ static int option;
public static void main(String[] args)
{ stack obj=new stack();
while(true)
{ System.out.println("\nEnter yours choice\n1.PUSH\n2.POP\n3.Display\n4..EXIT");
Scanner integer=new Scanner(System.in);
option=integer.nextInt();
switch(option)
{ case 1: System.out.println("Enter Element");
obj.element=integer.nextInt();
obj.push(obj.element);
break;
case 2: System.out.printf("Poped element is %d",obj.pop());
break;
case 3: obj.display(obj.st,obj.top);
break;
case 4: System.exit(0);
default: System.out.println("Wrong option");
}
}
}
}
```

**OUTPUT:**

```
lab06@cmrit-ThinkCentre-M70E: ~/DAA/lab/1b
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/1b$ javac *.java
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/1b$ java myStack
Enter stack size
2
Enter yours choice
1.PUSH
2.POP
3.Display
4..EXIT
1
Enter Element
10
Enter yours choice
1.PUSH
2.POP
3.Display
4..EXIT
1
Enter Element
20
Enter yours choice
1.PUSH
2.POP
3.Display
4..EXIT
3
Stack Elements:
10
20
Enter yours choice
1.PUSH
2.POP
3.Display
4..EXIT
2
Poped element is 20
Enter yours choice
```





**EXPERIMENT 2A****PROGRAM STATEMENT:**

Design a super class called Staff with details as StaffId, Name, Phone, Salary. Extend this class by writing three subclasses namely Teaching (domain, publications), Technical (skills), and Contract (period).

Write a Java program to read and display at least 3 staff objects of all three categories.

**CONCEPT:**

Here in this given problem we shall use inheritance for extending Staff class into: Teaching, Technical and Contract 3 subclasses using extends keyword. Each class is having the variables as given in the bracket. We will import the util package Scanner class to read 3 objects of each class. Create a constructor of Staff class to initialize StaffId, Name, Phone, Salary. And one display function into the Staff class to display the entered values.

All the data members of Staff will be inherited in Teaching, Technical and Contract using super keyword that calls the super class constructor to base class. Other additional data members of the subclasses would be initialized using their own constructor. Also along with their own constructors all 3 subclasses will have their own display method that invokes display method of super class Staff. Now in main() method using Scanner class we will read the values accordingly. To display these values we will create array of object of size 3 for each subclass Teaching, Technical and Contract. Using this array of objects we will display the values entered previously by invoking display method of each subclass. Below is the program that demonstrates the same.

**PROGRAM:**

/\*\* Design a super class called Staff with details as StaffId, Name, Phone, Salary. Extend this class by writing three subclasses namely Teaching (domain, publications), Technical (skills), and Contract (period). Write a Java program to read and display at least 3 staff objects of all three categories.\*/

```
package daa.EXP2AB;
import java.util.Scanner;
class Staff
{ //data members of Staff class
String StaffId=null, Name=null, Phone=null;
float Salary;
//constructor to initialize Staff class data members
Staff(String S,String N,String P,float Sa)
{ StaffId=S;
Name=N;
Phone=P;
Salary=Sa;
}
//display method to display the data members
public void display()
{ System.out.println("Entered Details are:");
System.out.println("StaffID:"+StaffId);
System.out.println("Name:"+Name);
System.out.println("Phone:"+Phone);
```



```
System.out.println("Salary:"+Salary);
}
} //Subclass of Staff
class Teaching extends Staff
{ //data members of Teaching class
String domain=null, publications=null;
public Teaching(String S, String N, String P, float Sa,String d,String p) {
//super invokes super class Staff constructor
super(S, N, P, Sa);
domain=d;
publications=p;
}
public void display()
{ //super invokes display super class method
super.display();
System.out.println("Domain:"+domain);
System.out.println("Publication:"+publications);
}
}
class Technical extends Staff
{ String skills=null;
public Technical(String S, String N, String P, float Sa,String Sk) {
//super invokes Staff super class constructor
super(S, N, P, Sa);
skills=Sk;
}
public void display() {
//super invokes display super class method
super.display();
System.out.println("Skills:"+skills);
}
} class Contract extends Staff
{ float period;//in years
public Contract(String S, String N, String P, float Sa,float p) {
super(S, N, P, Sa);
period=p;
}
public void display() {
//super invokes display super class method
super.display();
System.out.println("Period:"+period);
}
}
public class StaffDemo {
public static void main(String args[])
{ Scanner s = new Scanner(System.in);
String StaffId[]=new String[3];
```

```
String Name[]=new String[3];
String Phone[]=new String[3];
float Salary[]=new float[3];
String Domain[]=new String[3];
String Publications[]=new String[3];
System.out.println("Enter the Details for Teachers");
for(int i=0;i<3;i++)
{ //reading variables of Staff
System.out.println("Enter StaffId");
StaffId[i] = s.next();
System.out.println("Enter Name");
Name[i] = s.next();
System.out.println("Enter Phone");
Phone[i] = s.next();
System.out.println("Enter Salary");
Salary[i] = s.nextFloat();
//reading variables of Teaching
System.out.println("Enter Domain");
Domain[i] = s.next();
System.out.println("Enter Publications");
Publications[i] = s.next();
}
//Displaying variables of Teaching
Teaching T1[]=new Teaching[3];
for(int k=0;k<3;k++)
{ T1[k]=new Teaching(StaffId[k],Name[k],Phone[k],Salary[k],Domain[k],Publications[k]);
T1[k].display();
}
Scanner s2 = new Scanner(System.in);
String StaffIdT[]=new String[3];
String NameT[]=new String[3];
String PhoneT[]=new String[3];
float SalaryT[]=new float[3];
String Skills[]=new String[3];
System.out.println("Enter the Details for Technical");
for(int j=0;j<3;j++)
{ //reading variables of Staff
System.out.println("Enter StaffId");
StaffIdT[j] = s2.next();
System.out.println("Enter Name");
NameT[j] = s2.next();
System.out.println("Enter Phone");
PhoneT[j] = s2.next();
System.out.println("Enter Salary");
SalaryT[j] = s2.nextFloat();
//reading variables of Technical
System.out.println("Enter Skills");
```

```
Skills[j] = s2.next();
}
//Displaying variables of Technical
Technical Te1[]=new Technical[3];
for(int x=0;x<3;x++)
{ Te1[x]=new Technical(StaffIdT[x],NameT[x],PhoneT[x],SalaryT[x],Skills[x]);
Te1[x].display();
}
Scanner s3 = new Scanner(System.in);
String StaffIdC[]=new String[3];
String NameC[]=new String[3];
String PhoneC[]=new String[3];
float SalaryC[]=new float[3];
float period[]=new float[3];
System.out.println("Enter the Details for Contract");
for(int k=0;k<3;k++)
{ //reading variables of Staff
System.out.println("Enter StaffId");
StaffIdC[k] = s3.next();
System.out.println("Enter Name");
NameC[k] = s3.next();
System.out.println("Enter Phone");
PhoneC[k] = s3.next();
System.out.println("Enter Salary");
SalaryC[k] = s3.nextFloat();
//reading variables of Contract
System.out.println("Enter Period");
period[k] = s3.nextFloat();
}
//Displaying variables of Contract
Contract C1[]=new Contract[3];
for(int y=0;y<3;y++)
{
C1[y]=new Contract(StaffIdC[y],NameC[y],PhoneC[y],SalaryC[y],period[y]);
C1[y].display();
}
}
```

**OUTPUT:**

```
Enter the Details for Teachers
Enter StaffId
A1
Enter Name
Sonali
Enter Phone
9425959337
```

Enter Salary  
100000  
Enter Domain  
CSE  
Enter Publications  
XYZ  
Enter StaffId  
A2  
Enter Name  
Seema  
Enter Phone  
9425959326  
Enter Salary  
130000  
Enter Domain  
MCA  
Enter Publications  
XYZ  
Enter StaffId  
A3  
Enter Name  
Saurabh  
Enter Phone  
961794994  
Enter Salary  
1200000  
Enter Domain  
IT  
Enter Publications  
ABC  
Entered Details are:  
StaffID:A1  
Name:Sonali  
Phone:9425959337  
Salary:100000.0  
Domain:CSE  
Publication:XYZ  
Entered Details are:  
StaffID:A2  
Name:Seema  
Phone:9425959326  
Salary:130000.0  
Domain:MCA  
Publication:XYZ  
Entered Details are:  
StaffID:A3  
Name:Saurabh



Phone:961794994  
Salary:1200000.0  
Domain:IT  
Publication:ABC  
Enter the Details for  
Technical  
Enter StaffId  
B1  
Enter Name  
Vijay  
Enter Phone  
9425757331  
Enter Salary  
180000  
Enter Skills  
Java  
Enter StaffId  
B2  
Enter Name  
Anil  
Enter Phone  
9523373371  
Enter Salary  
120000  
Enter Skills  
C  
Enter StaffId  
B3  
Enter Name  
Anita  
Enter Phone  
9877868761  
Enter Salary  
130000  
Enter Skills  
C++  
Entered Details are:  
StaffID:B1  
Name:Vijay  
Phone:9425757331  
Salary:180000.0  
Skills:Java  
Entered Details are:  
StaffID:B2  
Name:Anil  
Phone:9523373371  
Salary:120000.0  
Skills:C



Entered Details are:

StaffID:B3

Name:Anita

Phone:9877868761

Salary:130000.0

Skills:C++

Enter the Details for

Contract

Enter StaffId

C1

Enter Name

Smith

Enter Phone

9870123512

Enter Salary

130000

Enter Period

3

Enter StaffId

C2

Enter Name

John

Enter Phone

9865456712

Enter Salary

150000

Enter Period

4

Enter StaffId

C3

Enter Name

Aruna

Enter Phone

190000

Enter Salary

190000

Enter Period

4

Entered Details are:

StaffID:C1

Name:Smith

Phone:9870123512

Salary:130000.0

Period:3.0





**EXPERIMENT 2B****PROGRAM STATEMENT**

Write a Java class called Customer to store their name and date\_of\_birth. The date\_of\_birth format should be dd/mm/yyyy. Write methods to read customer data as <name, dd/mm/yyyy> and display as <name,dd,mm,yyyy>using StringTokenizer class considering the delimiter character as “/”.

**CONCEPT**

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments. The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

**Constructors of StringTokenizer class:**

There are 3 constructors defined in the StringTokenizer class.

**Constructor Description**

StringTokenizer(String str) creates StringTokenizer with specified string.

StringTokenizer(String str,String delim) creates StringTokenizer with specified string and delimiter.

StringTokenizer(String str,String delim, Boolean returnValue) creates StringTokenizer with specified string, delimiter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.

**Methods of StringTokenizer class**

The 6 useful methods of StringTokenizer class are as follows:

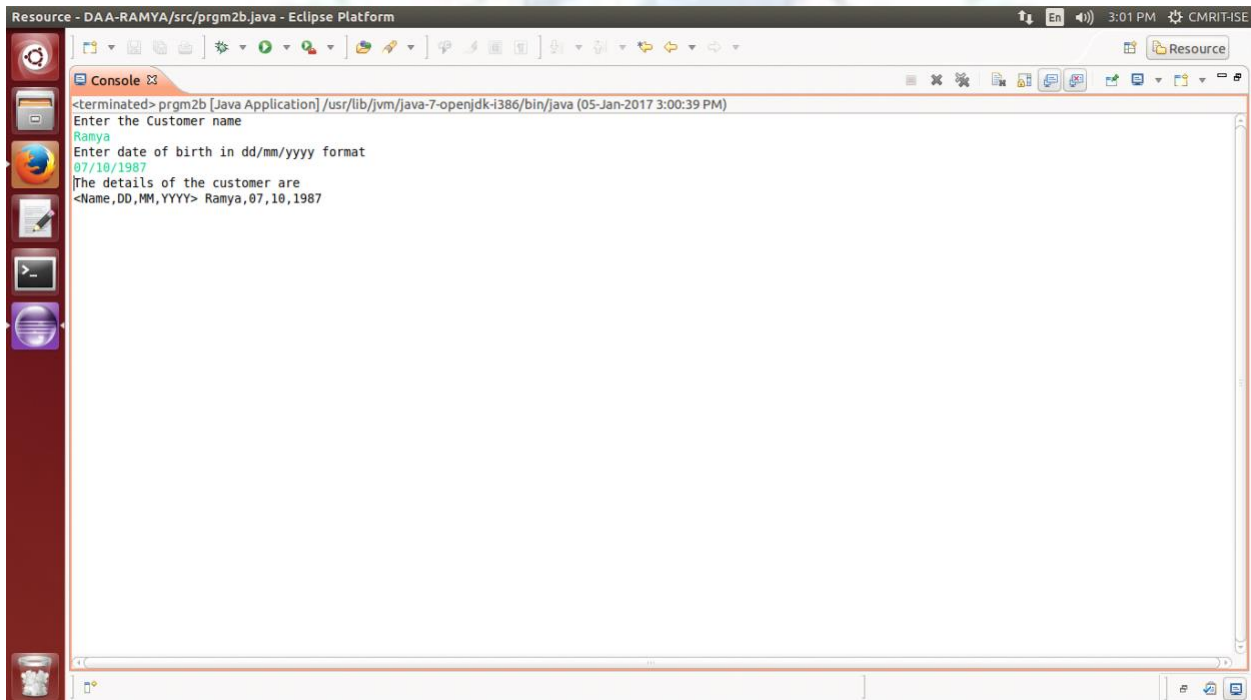
**Public method Description**

- 1)boolean hasMoreTokens() checks if there is more tokens available.
- 2)String nextToken() returns the next token from the StringTokenizer object.
- 3)String nextToken(String delim) returns the next token based on the delimiter.
- 4)boolean hasMoreElements() same as hasMoreTokens() method.
- 5)Object nextElement() same as nextToken() but its return type is Object.
- 6)int countTokens() returns the total number of tokens.

**PROGRAM :**

```
import java.util.Scanner;
import java.util.StringTokenizer;
class Customer
{ String name, date_of_birth;
void readdata()
{ Scanner sc = new Scanner(System.in);
System.out.println("Enter the Customer name");
name = sc.next();
System.out.println("Enter date of birth in dd/mm/yyyy format");
date_of_birth = sc.next();
sc.close();
```

```
}  
void display()  
{ StringTokenizer st = new StringTokenizer(date_of_birth,"/");  
System.out.println("The details of the customer are");  
System.out.print("<Name,DD,MM,YYYY> " + name );  
while(st.hasMoreTokens())  
System.out.print(", " + st.nextToken());  
}  
}  
public class prgm2b  
{ public static void main(String[] args)  
{ Customer c = new Customer();  
c.readdata();  
c.display();  
}  
}
```

**OUTPUT :**

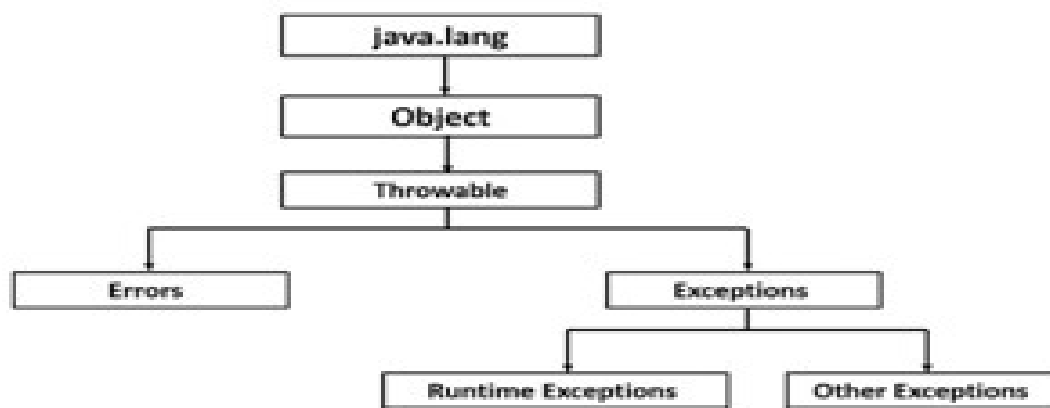
```
Resource - DAA-RAMYA/src/prgm2b.java - Eclipse Platform  
<terminated> prgm2b [Java Application] /usr/lib/jvm/java-7-openjdk-1386/bin/java (05-Jan-2017 3:00:39 PM)  
Enter the Customer name  
Ramyra  
Enter date of birth in dd/mm/yyyy format  
07/10/1987  
The details of the customer are  
<Name,DD,MM,YYYY> Ramya,07,10,1987
```

**EXPERIMENT 3A****PROGRAM STATEMENT**

Write a java program to read two integers and b. Compute a/b and print when b is not zero. Raise an exception when b is equal to zero.

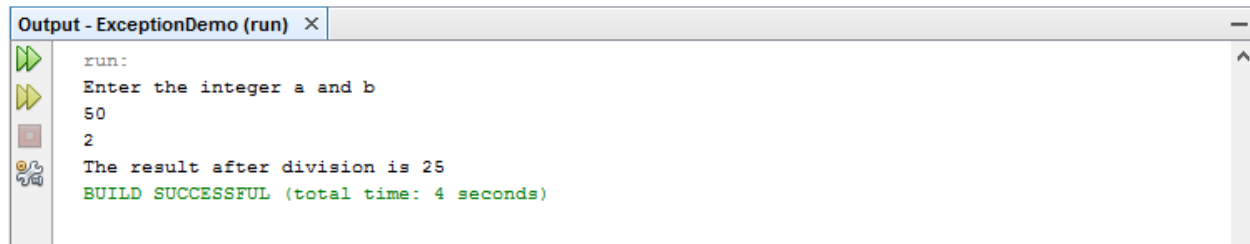
**CONCEPT****Exception Handling**

1. An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
2. All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.
3. The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

**PROGRAM**

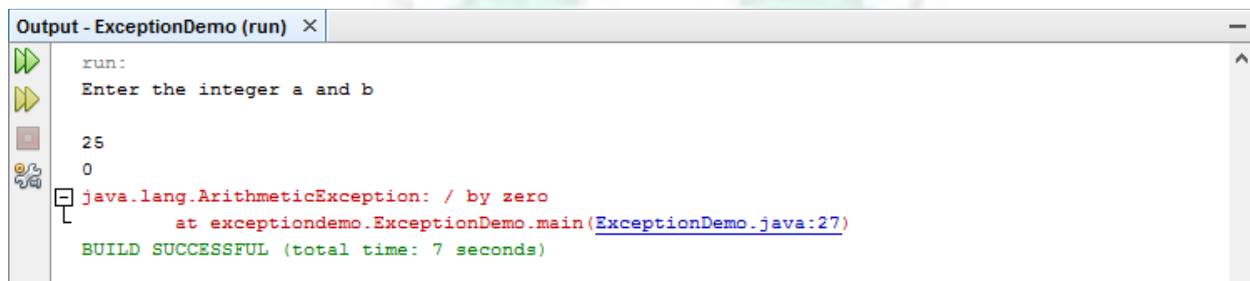
```
import java.util.*;
public class ExceptionDemo {
    public static void main(String[] args) {
        int a,b,result;
        Scanner s= new Scanner(System.in);
        System.out.println("Enter the integer a and b");
        a=s.nextInt();
        b=s.nextInt();
        try //try block
        {
            result=a/b;
            System.out.println("The result after division is " + result);
        }
        catch(Exception e) //catch Block to catch any exceptions
```

```
{  
e.printStackTrace();  
}  
}  
}
```

**OUTPUT:**

Output - ExceptionDemo (run) X

```
run:  
Enter the integer a and b  
50  
2  
The result after division is 25  
BUILD SUCCESSFUL (total time: 4 seconds)
```



Output - ExceptionDemo (run) X

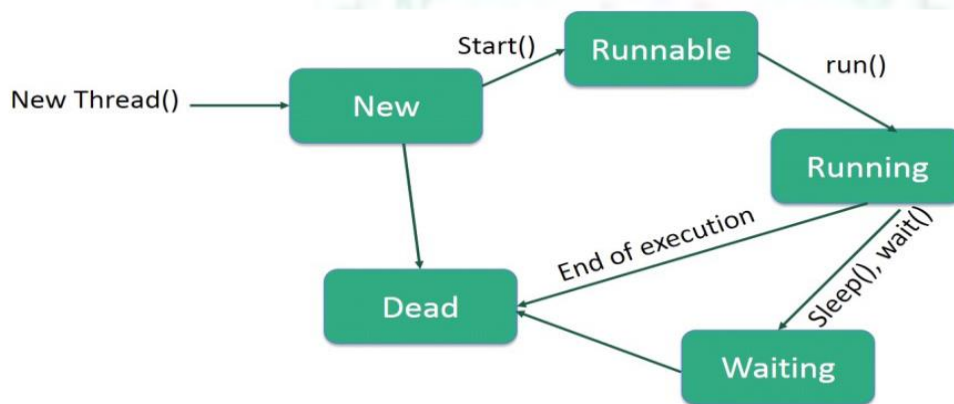
```
run:  
Enter the integer a and b  
  
25  
0  
java.lang.ArithmeticException: / by zero  
    at exceptiondemo.ExceptionDemo.main(ExceptionDemo.java:27)  
BUILD SUCCESSFUL (total time: 7 seconds)
```

**EXPERIMENT 3B****PROGRAM STATEMENT**

Write a Java program that implements a multi-thread application that has three threads. First thread generates a random integer for every 1 second; second thread computes the square of the number and prints; third thread will print the value of cube of the number.

**CONCEPT:**

Thread is basically a lightweight sub-process, a smallest unit of processing. Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

**Life Cycle of a Thread**

Following are the stages of the life cycle –

- **New** – A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** – After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** – Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** – A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** – A runnable thread enters the terminated state when it completes its task or otherwise terminates.

**Create a Thread by Extending a Thread Class**

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1: You will need to override **run()** method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Syntax of run() method – `public void run()`

Step 2 : Once Thread object is created, you can start it by calling **start()** method, which executes a call to run() method. Syntax of start() method – `void start()`;

### Synchronization :

Interthread communication is important when you develop an application where two or more threads exchange some information. At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. To do this we can use either Synchronized block or Synchronized method. There are three simple methods which makes thread communication possible, listed below –

#### Method &Description:

##### 1 **public void wait()**

Causes the current thread to wait until another thread invokes the notify().

##### 2 **public void notify()**

Wakes up a single thread that is waiting on this object's monitor.

##### 3 **public void notifyAll()**

Wakes up all the threads that called wait() on the same object.

These methods have been implemented as **final** methods in Object, so they are available in all the classes. All three methods can be called only from within a **synchronized** context.

### PROGRAM

```
class Compute
{
    int n;
    int flag=0, flag1=0;
    synchronized void generate()
    {
        while(flag==1 || flag1==1)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
                System.out.println("Interrupted Exception caught");
            }
        }
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted Exception caught");
        }
        n = (int) (Math.random()*100+1);
        System.out.println("The random number generated is " + n);
    }
}
```

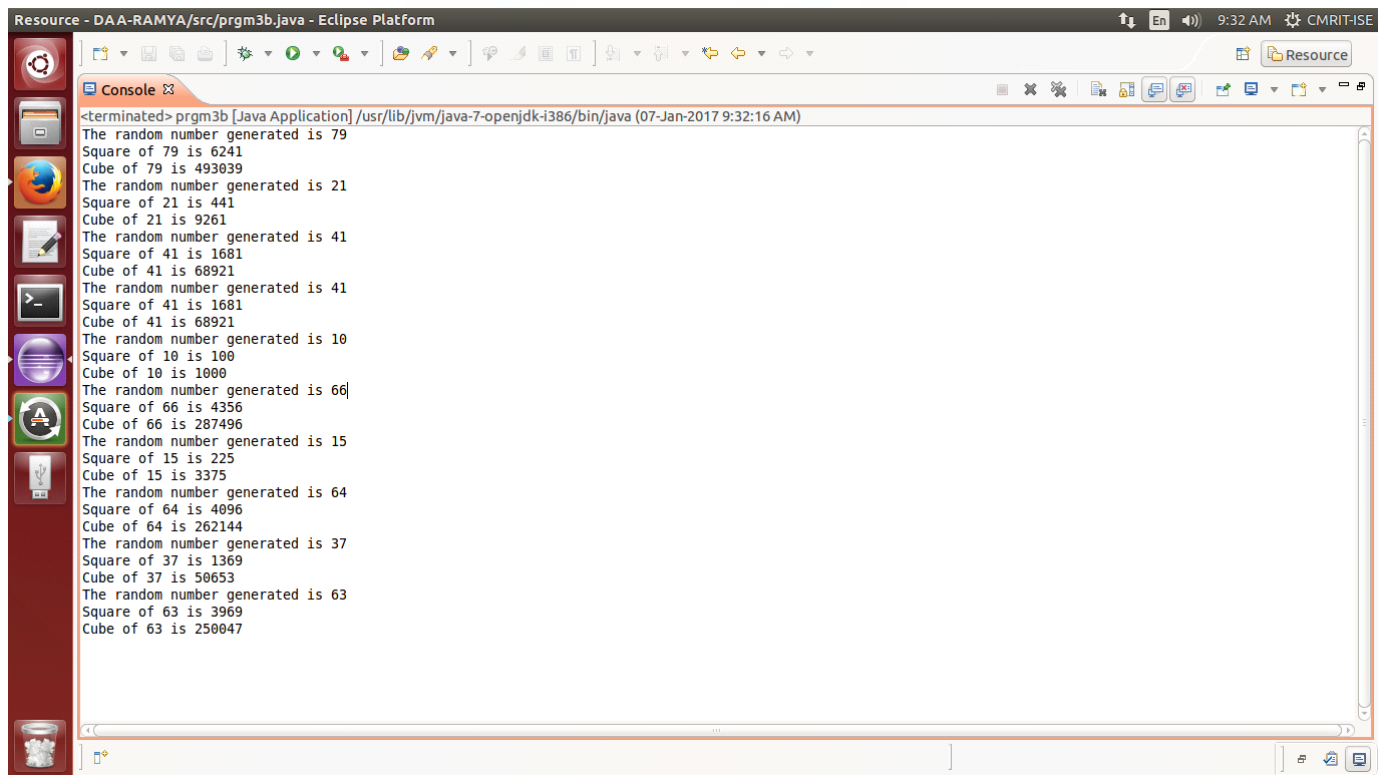


```
flag = flag1 = 1;
notifyAll();
} synchronized void square()
{ while(flag==0)
{ try
{
wait();
} catch(InterruptedException e)
{
System.out.println("Interrupted Exception caught");
}
}
System.out.println("Square of " +n + " is " + (n*n));
flag = 0;
notify();
} synchronized void cube()
{ while(flag1==0)
{ try
{
wait();
} catch(InterruptedException e)
{
System.out.println("Interrupted Exception caught");
}
}
System.out.println("Cube of "+ n + " is " + (n*n*n));
flag1=0;
notify();
}
}

public class prgm3b
{ public static void main(String[] args)
{ final Compute c = new Compute();
new Thread()
{ public void run()
{ for(int i=1;i<=10;i++)
c.generate();
}
}.start();
new Thread()
{ public void run()
{ for(int i=1;i<=10;i++)
c.square();
}
}.start();
new Thread()
{ public void run()
```

```
{ for(int i=1;i<=10;i++)  
c.cube();  
}  
}.start();  
}  
}
```

**OUTPUT:**



```
<terminated> prgm3b [Java Application] /usr/lib/jvm/java-7-openjdk-i386/bin/java (07-Jan-2017 9:32:16 AM)  
The random number generated is 79  
Square of 79 is 6241  
Cube of 79 is 493039  
The random number generated is 21  
Square of 21 is 441  
Cube of 21 is 9261  
The random number generated is 41  
Square of 41 is 1681  
Cube of 41 is 68921  
The random number generated is 41  
Square of 41 is 1681  
Cube of 41 is 68921  
The random number generated is 10  
Square of 10 is 100  
Cube of 10 is 1000  
The random number generated is 66  
Square of 66 is 4356  
Cube of 66 is 287496  
The random number generated is 15  
Square of 15 is 225  
Cube of 15 is 3375  
The random number generated is 64  
Square of 64 is 4096  
Cube of 64 is 262144  
The random number generated is 37  
Square of 37 is 1369  
Cube of 37 is 50653  
The random number generated is 63  
Square of 63 is 3969  
Cube of 63 is 250047
```

**EXPERIMENT 4****PROGRAM STATEMENT**

Sort a given set of  $n$  integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of  $n > 5000$  and record the time taken to sort. Plot a graph of the time taken versus  $n$  on graph sheet. The elements can be read from a file or can be generated using the random number generator.

Demonstrate using Java how the divide-and-conquer method works along with its time complexity analysis: worst case, average case and best case.

**CONCEPT**

**Divide and Conquer:** In divide and conquer approach, the problem in hand, is divided into smaller subproblems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem. Broadly, we can understand **divide-and-conquer** approach in a three-step process.

**Divide/Break:** This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

**Conquer/Solve:** This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

**Merge/Combine:** When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps work so close that they appear as one.

The following computer algorithms are based on **divide-and-conquer** programming approach –

- ☐ Merge Sort
- ☐ Quick Sort

**Quick Sort Method:**

Quick Sort divides the array according to the value of elements. It rearranges elements of a given array  $A[0..n-1]$  to achieve its partition, where the elements before position  $s$  are smaller than or equal to  $A[s]$  and all the elements after position  $s$  are greater than or equal to  $A[s]$ .

$A[0] \dots A[s-1] \ A[s] \ A[s+1] \dots A[n-1]$

All are  $\leq A[s]$  all are  $\geq A[s]$

**Algorithm :** QUICKSORT( $a[l..r]$ ) //Sorts a subarray by quicksort

//Input: A subarray  $A[l..r]$  of  $A[0..n-1]$ , defined by its left and right indices  $l$  and  $r$  //Output:

Subarray

$A[l..r]$  sorted in nondecreasing order

{ if  $l < r$

{ $s$

$\leftarrow$  Partition( $A[l..r]$ ) //  $s$  is a split position QUICKSORT( $A[l..s-1]$ )

QUICKSORT( $A[s+1..r]$ )

}}

**Algorithm :** Partition( $A[l..r]$ )

//Partition a subarray by using its first element as its pivot

//Input: A subarray A[l..r] of A[0..n-1], defined by its left and right indices l and r (l < r) //Output: A partition of A[l..r], with the split position returned as this function's value

```
{
p ← A[l]
i ← l; j ← r+1 repeat
{ repeat i ← i+1 until A[i] >=p
repeat j ← j-1 until A[j] <=p swap(A[i],A[j])
} until i>=j
swap(A[i],A[j]) // undo last swap when i>=j swap(A[l],A[j])
return j
}
```

**Complexity:**  $C_{\text{best}}(n) = 2 C_{\text{best}}(n/2) + n$  for  $n > 1$   $C_{\text{best}}(1) = 0$

$C_{\text{worst}}(n) = (n^2)$

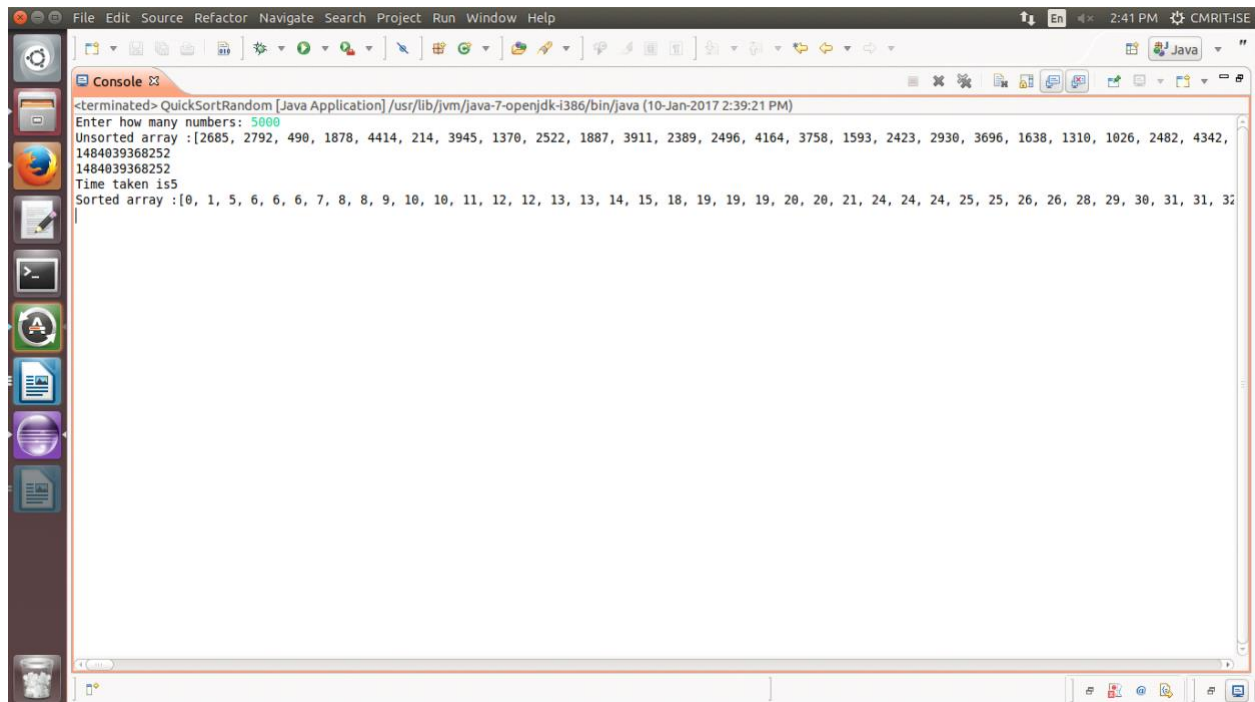
$C_{\text{avg}}(n) \approx 1.38n \log_2 n$

## PROGRAM

```
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;
public class QuickSortRandom{
public static void main(String args[]) {
// unsorted integer array
Scanner scanner = new Scanner(System.in);
System.out.print("Enter how many numbers: ");
int n = scanner.nextInt();
int[] unsorted=new int[n];
for(int i=0;i<n;i++)
{ Random random = new Random();
unsorted[i]=random.nextInt(5000);
}
System.out.println("Unsorted array :" + Arrays.toString(unsorted));
QuickSort1 algorithm = new QuickSort1();
// sorting integer array using quicksort algorithm
algorithm.sort(unsorted);
// printing sorted array
System.out.println("Sorted array :" + Arrays.toString(unsorted));
}
}
class QuickSort1 {
private int input[];
private int length;
long t1,t2,t;
public void sort(int[] numbers) {
if (numbers == null || numbers.length == 0) {
return;
}
}
```

```
this.input = numbers;
length = numbers.length;
t1=System.currentTimeMillis();
//System.out.println(t1);
quickSort1(0, length - 1);
t2=System.currentTimeMillis();
//System.out.println(t2);
t=t2-t1;
System.out.println("Time taken is to sort elements is "+ t);
}
/* * This method implements in-place quicksort algorithm recursively*/
private void quickSort1(int low, int high) {
int i = low;
int j = high;
int pivot = input[low + (high - low) / 2];
// Divide into two arrays
while (i <= j) {
while (input[i] < pivot) {
i++;
}
while (input[j] > pivot) {
j--;
}
if (i <= j) {
swap(i, j);
// move index to next position on both sides
i++;
j--;
}
}
// calls quickSort() method recursively
if (low < j) {
quickSort1(low, j);
}
if (i < high) {
quickSort1(i, high);
}
}
private void swap(int i, int j) {
int temp = input[i];
input[i] = input[j];
input[j] = temp;
}
}
```

## OUTPUT



```
<terminated> QuickSortRandom [Java Application] /usr/lib/jvm/java-7-openjdk-1386/bin/java (10-Jan-2017 2:39:21 PM)
Enter how many numbers: 5000
Unsorted array :[2685, 2792, 490, 1878, 4414, 214, 3945, 1370, 2522, 1887, 3911, 2389, 2496, 4164, 3758, 1593, 2423, 2930, 3696, 1638, 1310, 1026, 2482, 4342,
1484039368252
Time taken is5
Sorted array :[0, 1, 5, 6, 6, 6, 7, 8, 8, 9, 10, 10, 11, 12, 12, 13, 13, 14, 15, 18, 19, 19, 19, 20, 20, 21, 24, 24, 24, 25, 25, 26, 26, 28, 29, 30, 31, 31, 32]
```

**EXPERIMENT 5**



**PROGRAM STATEMENT**

Sort a given set of  $n$  integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of  $n > 5000$ , and record the time taken to sort. Plot a graph of the time taken versus  $n$  on graph sheet. The elements can be read from a file or can be generated using the random number generator.

Demonstrate using Java how the divide- and-conquer method works along with its time complexity analysis: worst case, average case and best case.

**CONCEPT**

Merge sort is a perfect example of a successful application of the divide-and-conquer technique.

- ☐ Split array  $A[1..n]$  in two and make copies of each half in arrays  $B[1.. n/2]$  and  $C[1.. n/2]$
- ☐ Sort arrays  $B$  and  $C$
- ☐ Merge sorted arrays  $B$  and  $C$  into array  $A$  as follows:
  - a) Repeat the following until no elements remain in one of the arrays:
    - i) compare the first elements in the remaining unprocessed portions of the arrays
    - ii) copy the smaller of the two into  $A$ , while incrementing the index indicating the unprocessed portion of that array
  - b) Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into  $A$ .

**Algorithm:** MergeSort ( $A [0..n-1]$ )

//This algorithm sorts array  $A [0..n-1]$  by recursive mergesort. //Input: An array  $A [0..n-1]$  of orderable elements.

//Output: Array  $A [0..n-1]$  sorted in non-decreasing order

```
{ if  $n > 1$ 
{ Copy  $A [0... \lfloor n/2 \rfloor - 1]$  to  $B [0... \lfloor n/2 \rfloor - 1]$  Copy  $A [\lfloor n/2 \rfloor ... n-1]$ 
to  $C [0... \lceil n/2 \rceil - 1]$  MergeSort ( $B [0... \lfloor n/2 \rfloor - 1]$ )
MergeSort ( $C [0... \lceil n/2 \rceil - 1]$ ) Merge ( $B, C, A$ )
}
}
```

**Algorithm:** Merge ( $B [0...p-1], C [0...q-1], A [0...p+q-1]$ ) //Merges two sorted arrays into one sorted array.

//Input: Arrays  $B [0...p-1]$  and  $C [0...q-1]$  both sorted. //Output: Sorted array  $A [0...p+q-1]$  of the elements of  $B$  and  $C$ .

```
{
 $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$  while  $i < p$  and  $j < q$ 
do
{
if  $B[i] \leq C[j]$ 
 $A[k] \leftarrow B[i]; i \leftarrow i+1$ 
else
 $A[k] \leftarrow C[j]; j \leftarrow j+1; k \leftarrow k+1$ 
} if  $i = p$ 
copy  $C [j...q-1]$  to  $A[k...p+q-1]$ 
else
copy  $B [i...p-1]$  to  $A[k...p+q-1]$ 
}
```

**Complexity:**

All cases have same efficiency:  $\Theta(n \log n)$

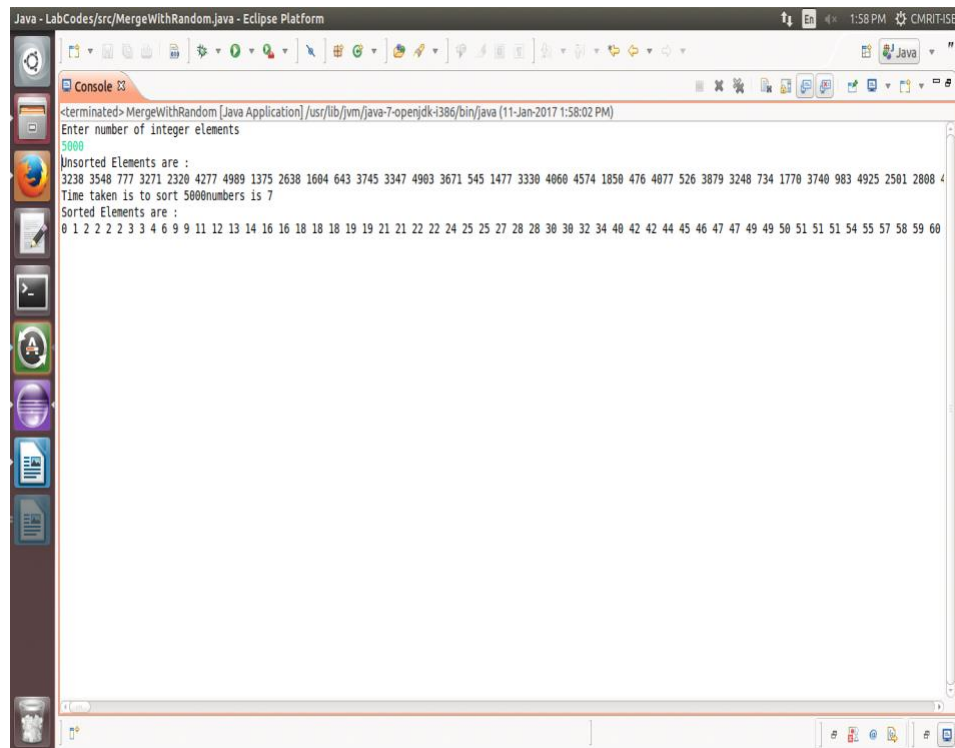
Number of comparisons is close to theoretical minimum for comparison-based sorting:  $\log n! \approx n \lg n - 1.44 n$

**PROGRAM:**

```
import java.util.Arrays;
import java.util.Scanner;
import java.util.Random;
public class MergeWithRandom {
    private int[] array;
    private int[] temp;
    private int length;
    public static void main(String a[]){
        Random random = new Random();
        Scanner scan = new Scanner( System.in );
        int l;
        long t1,t2,t;
        System.out.println("Enter number of integer elements");
        int n = scan.nextInt();//get input
        int arr[] = new int[ n ];
        /* Accept elements */
        // System.out.println("\nEntered "+ n +" integer elements");
        for (l = 0; l < n; l++)
            arr[l] = random.nextInt(5000);
        MyMergeSort m = new MyMergeSort();
        System.out.println("Unsorted Elements are :");
        for(int i:arr){
            System.out.print(i);
            System.out.print(" ");
        }
        t1=System.currentTimeMillis();//get the current time before calling MergeSort
        // System.out.println("t1 = "+t1);
        m.sort(arr);
        t2=System.currentTimeMillis();//get the current time after calling MergeSort
        System.out.println();
        //System.out.println("t2 = " + t2);
        t=t2-t1;
        System.out.println("Time taken is to sort "+ n +"numbers is "+ t);
        System.out.println("Sorted Elements are :");
        for(int i:arr){
            System.out.print(i);
            System.out.print(" ");
        }
    }
    public void sort(int inputArr[]) {
        this.array = inputArr;
        this.length = inputArr.length;
    }
}
```

```
this.temp = new int[length];
doMergeSort(0, length - 1);
}
private void doMergeSort(int low, int high) {
if (low < high) {
int middle = low + (high - low) / 2;
// Below step sorts the left side of the array
doMergeSort(low, middle);
// Below step sorts the right side of the array
doMergeSort(middle + 1, high);
// Now merge both sides
mergeParts(low, middle, high);
}
}
private void mergeParts(int low, int middle, int high) {
for (int i = low; i <= high; i++) {
temp[i] = array[i];
}
int i = low;
int j = middle + 1;
int k = low;
while (i <= middle && j <= high) {
if (temp[i] <= temp[j]) {
array[k] = temp[i];
i++;
} else {
array[k] = temp[j];
j++;
}
k++;
}
while (i <= middle) {
array[k] = temp[i];
k++;
i++;
}
}
}
```

## OUTPUT



```
Java - LabCodes/src/MergeWithRandom.java - Eclipse Platform
<terminated> MergeWithRandom [Java Application] /usr/lib/jvm/java-7-openjdk-i386/bin/java (11-Jan-2017 1:58:02 PM)
Enter number of integer elements
5000
Unsorted Elements are :
3238 3548 777 3271 2320 4277 4989 1375 2638 1604 643 3745 3347 4903 3671 545 1477 3330 4060 4574 1850 476 4077 526 3879 3248 734 1770 3740 983 4925 2501 2808 4
Time taken is to sort 5000 numbers is 7
Sorted Elements are :
0 1 2 2 2 3 3 4 6 9 9 11 12 13 14 16 16 18 18 18 19 19 21 21 22 22 24 25 25 27 28 28 30 30 32 34 40 42 42 44 45 46 47 47 49 49 50 51 51 51 54 55 57 58 59 60
```

## EXPERIMENT 6A

**PROGRAM STATEMENT**

Implement in Java, the 0/1 Knapsack problem using Dynamic Programming method

**CONCEPT**

Dynamic-Programming Solution to the 0-1 Knapsack Problem: Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

In the knapsack problem we are given a set of  $n$  items, where each item  $i$  is specified by a size  $w_i$  and a value  $v_i$

We are also given a size bound  $W$  (the size of our knapsack). The goal is to find the subset of items of maximum total value such that sum of their sizes is at most  $W$  (they all fit into the knapsack). Now, instead of being able to take a certain weight of an item, you can only either take the item or not take the item.

Algorithm:

//(n items, W weight of sack) Input:  $n, w, v$  and  $W$  – all integers

//Output:  $V(n, W)$

// Initialization of first column and first row elements

- Repeat for  $i = 0$  to  $n$

set  $V(i, 0) = 0$

- Repeat for  $j = 0$  to  $W$

Set  $V(0, j) = 0$

//complete remaining entries row by row

- Repeat for  $i = 1$  to  $n$

repeat for  $j = 1$  to  $W$

if ( $w_i \leq j$ )  $V(i, j) = \max\{V(i-1, j), V(i-1, j-w_i) + v_i\}$

if ( $w_i > j$ )  $V(i, j) = V(i-1, j)$

- Print  $V(n, W)$

**PROGRAM**

```
import java.util.Scanner;
```

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

```
public class knapsack1
```

```
{ int[] weights;
```

```
int[] values;
```

```
int objects, W;
```

```
int[][] table;
```

```
int i, j;
```

```
int[][] keep;
```

```
knapsack1(Scanner sc)
```

```
{ //reading number of objects and capacity of knapsack
```

```
System.out.println("Enter no of objects and Total Weights");
```

```
objects=sc.nextInt();
```

```
W=sc.nextInt();
```

```
values=new int[objects+1];
```

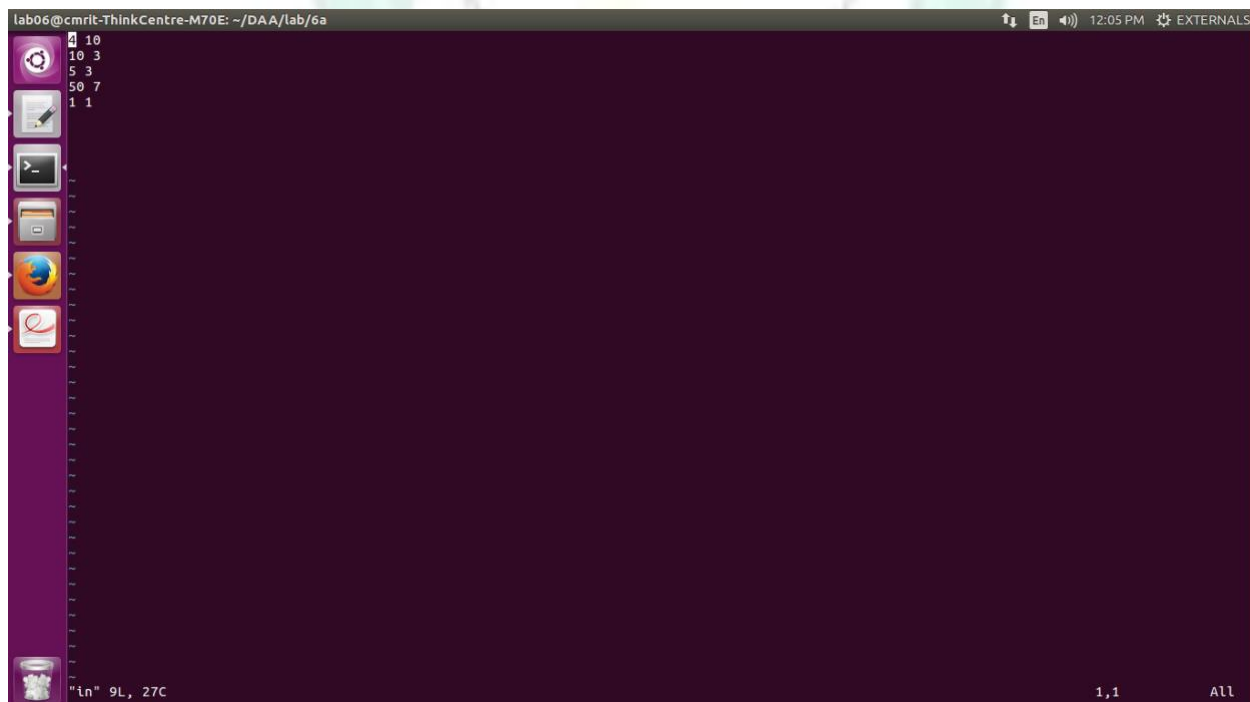
```
weights=new int[objects+1];
```

```
table=new int[objects+1][W+1];
keep=new int[objects+1][W+1];
//reading each object value and weight
for(i=1;i<=objects;i++)
{ System.out.printf("Enter %d object value and weight",i);
values[i]=sc.nextInt();
weights[i]=sc.nextInt();
}
value();
}
void value()
{//find the knapsack value and objects selected
for(i=0;i<W+1;i++)
{ table[0][i]=0;
keep[0][i]=0;
}
for(i=0;i<objects+1;i++)
{ keep[0][i]=0;
table[i][0]=0;
}
//Let's fill the values row by row
for(i=1;i<objects+1;i++)
{ for(j=1;j<W+1;j++)
{
//Given a weight, check if the value of the current item + value of the item that we could afford
withthe remaining weight
//is greater than the value without the current item itself
if(weights[i]<=j && (table[i-1][j]) < (table[i-1][j-weights[i]]+values[i]))
{ table[i][j]=table[i-1][j-weights[i]]+values[i];
keep[i][j]=1;
}
else
{//If the current item's weight is more than the running weight,
Just carry forward the value without the current item
table[i][j]=table[i-1][j];
keep[i][j]=0;
}
}
}
System.out.println("Value=%d\n",table[objects][W]);
for(i=0;i<objects+1;i++)
{ for(j=0;j<W+1;j++)
{ System.out.println("%d\t",keep[i][j]);
}
}
System.out.println("\n");
}
System.out.println("Selected objects are:");
```



```
int n=objects;
int Wt=W;
for(i=n;i>0;i--)
{ if(keep[i][Wt]==1)
{ System.out.println(i);
Wt=Wt-weights[i];
}
}
}
}
import java.util.Scanner;
public class MyClass
{
public static void main(String[] arg)
{
Scanner sc=new Scanner(System.in);
knapsack1 obj=new knapsack1(sc);
}
}
```

## OUTPUT



```
lab06@cmrit-ThinkCentre-M70E: ~/DAA/lab/6a
10
3
5 3
50 7
1 1
```

```
lab06@cmrit-ThinkCentre-M70E: ~/DAA/lab/6a
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/6a$ javac *.java
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/6a$ java MyClass < in
Enter no of objects and Total Weights
Enter 1 object value and weightEnter 2 object value and weightEnter 3 object value and weightEnter 4 object value and weightValue=60
0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1
0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 1 1 1
0 1 1 0 1 1 0 0 1 1 0
Selected objects are:
3
1
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/6a$
```

**EXPERIMENT 6B**

**PROGRAM STATEMENT**

Implement in Java, the Fractional Knapsack problem using Greedy method

**CONCEPT**

Greedy Solution to the Fractional Knapsack Problem

The basic idea of greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with highest ratio and add them until we can't add the next item as whole and at the end add the next item as much as we can. This will always be optimal solution of this problem.

□ Algorithm:

o Assume knapsack holds weight  $W$  and items have value  $v_i$  and weight  $w_i$

o Rank items by value/weight ratio:  $v_i / w_i$

□ Thus:  $v_i / w_i \geq v_j / w_j$ , for all  $i \leq j$

o Consider items in order of decreasing ratio

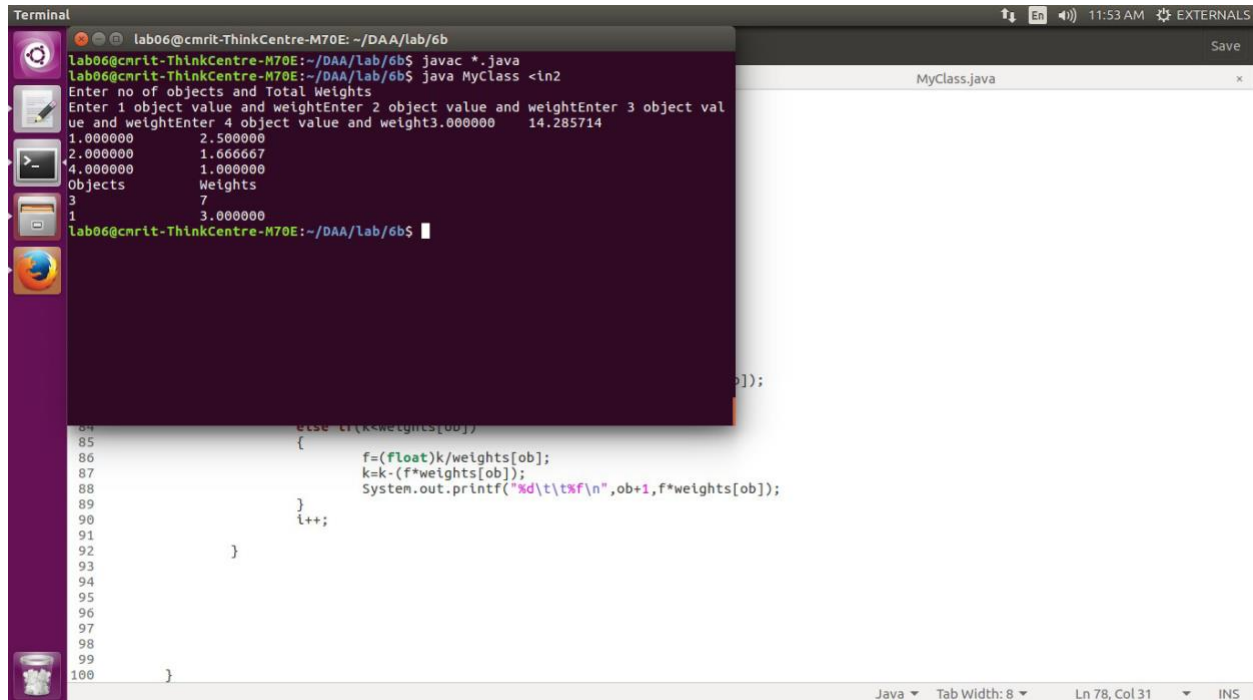
o Take as much of each item as possible

**PROGRAM**

```
import java.util.Scanner;
public class Greedy
{ int[] weights;
  int[] values;
  int objects;
  float W;
  float[][] v;
  Greedy(Scanner sc)
  { //reading number of objects and capacity of knapsack
    System.out.println("Enter no of objects and Total Weights");
    objects=sc.nextInt();
    W=sc.nextInt();
    values=new int[objects];
    weights=new int[objects];
    v=new float[objects][2];
    for(int i=0;i<objects;i++)
    { //reading each object value and weight
      System.out.printf("Enter %d object value and weight",i+1);
      values[i]=sc.nextInt();
      weights[i]=sc.nextInt();
    }
    value();
  }
  public void sort()
  { int i,j;
    float temp;
    for(i=0;i<objects;i++)
    { for(j=0;j<objects-1;j++)
      { if(v[j][1]<v[j+1][1])
        { temp=v[j][1];
          v[j][1]=v[j+1][1];
          v[j+1][1]=temp;
        }
      }
    }
  }
}
```

```
v[j][1]=v[j+1][1];
v[j+1][1]=temp;
temp=v[j][0];
v[j][0]=v[j+1][0];
v[j+1][0]=temp;
}
}
}
}
public void value()
{ //find cost per value
int i;
for(i=0;i<objects;i++)
{ v[i][0]=i+1;
v[i][1]=(float)values[i]/weights[i];
}
// sort cost per value
sort();
// display sorted array
for(i=0;i<objects;i++)
System.out.printf("%f\t%f\n", v[i][0],v[i][1]);
float k=W,f;
int ob;
i=0;
System.out.println("Objects\tWeights");
while(k>0)
{ ob=(int)v[i][0];
ob=ob-1;//index starting from 0
if(k>=weights[ob])
{
k=k-weights[ob];
System.out.printf("%d\t\t%d\n",ob+1,weights[ob]);
}
else if(k<weights[ob])
{ f=(float)k/weights[ob];
k=k-(f*weights[ob]);
System.out.printf("%d\t\t%f\n",ob+1,f*weights[ob]);
}
i++;
}
}
}
import java.util.Scanner;
public class MyClass
{ public static void main(String[] arg)
{
Scanner sc=new Scanner(System.in);
```

```
Greedy obj=new Greedy(sc);  
}  
}
```

**OUTPUT:**

```
lab06@cmrit-ThinkCentre-M70E: ~/DAA/lab/6b  
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/6b$ javac *.java  
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/6b$ java MyClass <in2  
Enter no of objects and Total Weights  
Enter 1 object value and weightEnter 2 object value and weightEnter 3 object val  
ue and weightEnter 4 object value and weight3.000000 14.285714  
1.000000 2.500000  
2.000000 1.666667  
4.000000 1.000000  
Objects Weights  
3 7  
1 3.000000  
lab06@cmrit-ThinkCentre-M70E:~/DAA/lab/6b$  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
}
```

**EXPERIMENT 7**

**PROGRAM STATEMENT**

From a given vertex in weighted connected graph, find shortest path to other vertices using Dijkstras algorithm. Write the program in java.

**CONCEPT**

It is a greedy algorithm which finds the shortest path from the source vertex to all other vertices of the graph.

**Steps**

1. Input a cost matrix for graph. Read the source vertex and n from user
2. Create the array d [1...n] which stores the distance from source vertex to all other vertices of graph. Initialize distance to source vertex as 0 (i.e. d[source] = 0) and remaining vertices as 999.
3. Create the array visited [1...n] which keeps track of all the visited nodes. Visit the source vertex and initialize visited[source] = 1.
4. For all adjacent vertices [v<sub>i</sub>, v<sub>i+1</sub>, ...] for source vertex calculate distance using formula  $d[v_i] = \min(d[v_i], d[\text{source}] + \text{cost}[\text{source}][v_i])$ . Update the array d [1...n].
5. For all adjacent vertices find vertex v<sub>i</sub> which has minimum distance from source vertex.
6. Initialize source = v<sub>i</sub>. Repeat the steps 4, 5 until there are some vertices which are unvisited.
7. Stop

**PROGRAM**

```
import java.util.*;
public class DijkstraDemo {
    public static int min(int a, int b) // method to return the minimum of two numbers
    { if(a<b)
      return a;
      else
      return b;
    }
    public static boolean check(int[] v, int n)
    { for(int i=1; i<n+1; i++)
      {
        if(v[i]!=1)
          return true;
      }
      return false;
    }
    public static void main(String[] args) {
      Scanner s=new Scanner(System.in);
      System.out.println("Enter the number of nodes in graph");
      int n=s.nextInt();
      int i,j;
      int graph[][]=new int[n+1][n+1];
      int visited[]=new int[n+1];
      int d[]= new int[n+1];
      TreeMap<Integer,Integer> map= new TreeMap<Integer,Integer>();
      System.out.println("Enter cost adjacency matrix for graph. If two nodes
        are not connected enter 999");
      for(i=1; i<n+1; i++)
```

```
{ for(j=1;j<n+1;j++)
graph[i][j]=s.nextInt();
}
for(i=1;i<n+1;i++)
{ visited[i]=0;
d[i]=999;
}
System.out.println("Enter the source vertex");
int source=s.nextInt();
visited[source]=1;
d[source]=0;
int u=source;
int v,a=0;
do // Calculating the d[] array
{ for(v=1;v<n+1;v++)
{
if(graph[u][v]!=999 && visited[v]!=1 && graph[u][v]!=0)
{ d[v]=min(d[v],(d[u]+graph[u][v]));
map.put(d[v],v);
}
}
u=map.firstEntry().getValue();
visited[u]=1;
map.clear();
}
while(check(visited,n));
for(i=1;i<n+1;i++)
System.out.println("Distance from source to " + i + " is " + d[i]);
}
}
```



**OUTPUT**

```
reshma@reshma-SVE1513ACNB: ~/Desktop
reshma@reshma-SVE1513ACNB:~/Desktop$ javac DijkstraDemo.java
reshma@reshma-SVE1513ACNB:~/Desktop$ java DijkstraDemo
Enter the number of nodes in graph
6
Enter cost adjacency matrix for graph. If two nodes are not connected enter 999
0 9 7 7 999 1
9 0 999 2 9 3
7 999 0 3 6 999
7 2 3 0 10 999
999 9 6 10 0 7
1 3 999 999 7 0
Enter the source vertex
1
Distance from source to 1 is 0
Distance from source to 2 is 4
Distance from source to 3 is 7
Distance from source to 4 is 6
Distance from source to 5 is 8
Distance from source to 6 is 1
```

**EXPERIMENT 8**

**PROGRAM STATEMENT**

Find Minimum Cost Spanning Tree of a given connected undirected graph using **Kruskal's algorithm**.

Implement the program in Java. Use Union-Find algorithm in your program.

**CONCEPT**

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

We found three spanning trees off one complete graph. A complete undirected graph can have maximum  $n-2$  number of spanning trees, where  $n$  is the number of nodes. In the above addressed example,  $3-2 = 3$  spanning trees are possible.

**General Properties of Spanning Tree**

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph G –

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

**Minimum Spanning Tree**

Given a connected and undirected graph, a *spanning tree* of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A *minimum spanning tree (MST)* or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

To understand Kruskal's algorithm let us consider the following example –

**Step 1 - Remove all loops and Parallel Edges**

Remove all loops and parallel edges from the given graph.

In case of parallel edges, keep the one which has the least cost associated and remove all others.

**Step 2 - Arrange all edges in their increasing order of weight**

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

**Step 3 - Add the edge which has the least weightage**

Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall

keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.

The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.

Next cost is 3, and associated edges are A,C and C,D. We add them again –

Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –

We ignore it. In the process we shall ignore/avoid all edges that create a circuit.

We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.

Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.

By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

### PROGRAM

```
package kruskaldemo;
import java.util.Scanner;
public class Kruskal
{ static int find(intx, intparent[])
{ while (parent[x] >= 0)
x = parent[x];
returnx;
}
//Applying union find algorithm
Static void setunion(intx,inty, intparent[])
{ if (parent[x] <parent[y])
{ parent[x]=parent[x]+parent[y];
parent[y] = x;
} else
{ parent[y]=parent[x]+parent[y];
parent[x] = y;
}
}s
taticvoid KruskalAlgo(int[][]edge, intn)
{ inti, x, y, cost=0, ecoun = 0;
int [] parent = newint[n];
for( i=0; i<n; i++)
parent[i] = -1;
i = 0;
int[][] mst = newint[n][2];
while ( i< ( n * n ) &&ecoun<n-1 )
{ if(edge[i][2] == 999)
break;
x = find(edge[i][0],parent);
y = find(edge[i][1],parent);
if ( x!=y )
{ cost = cost + edge[i][2];
mst[ecoun][0] = edge[i][0];
```

```

mst[ecount++][1] = edge[i][1];
setunion(x,y,parent);
} i++;
} if(ecount<n-1)
System.out.println("The minimal spanning tree could not be found ");
else
{ System.out.println("the minimal spanning tree cost : " + cost);
System.out.println("the minimal spanning tree is: ");
for(i=0;i<n-1;i++)
System.out.println(mst[i][0] + " - " + mst[i][1]);
}
}
}

public static void main(String[] args) {
Scanner s = new Scanner(System.in);
System.out.println("Enter the number of vertices: ");
int n = s.nextInt();
int i, j, k = 0;
int[][] a = new int[n][n];
int[][] edge = new int[n*n][3];
System.out.println("Enter the adjacency matrix(999-no edge): ");
for ( i = 0 ; i<n ; i++ )
for ( j = 0 ; j<n ; j++ )
{ a [ i ] [ j ] = s.nextInt();
edge [ k ] [ 0 ] = i;
edge [ k ] [ 1 ] = j;
edge [ k++ ] [ 2 ] = a [ i ] [ j ];}
// Java Comparator interface is used to order the objects of user-defined class.
java.util.Arrays.sort(edge, new java.util.Comparator<int[]>() {
public int compare(int[] a, int[] b) {
return Integer.compare(a[2], b[2]);
}
});
KruskalAlgo(edge,n);
s.close();
}
}

```

**OUTPUT:**

Enter the number of vertices:

6

Enter the adjacency matrix(999-no edge):

999 7 8 999 999 999

7 999 6 3 999 999

999 6 999 4 2 5

8 3 4 999 3 999

999 999 2 3 999 2

999 999 5 999 2 999

the minimal spanning tree cost :17

the minimal spanning tree is:

2 - 4

4 - 5

1 - 3

3 - 4



## **EXPERIMENT 9**

### **PROGRAM STATEMENT**

Find Minimum Cost Spanning Tree of a given connected undirected graph using **Prim's algorithm**.

### CONCEPT

Prim's algorithm to find minimum cost spanning tree uses the greedy approach. Prim's algorithm shares a similarity with the **shortest path first** algorithms.

Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

To contrast with Kruskal's algorithm and to understand Prim's algorithm better, we shall use the same

Example –

#### Step 1 - Remove all loops and parallel edges

Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

#### Step 2 - Choose any arbitrary node as root node

In this case, we choose **S** node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

#### Step 3 - Check outgoing edges and select the one with less cost

After choosing the root node **S**, we see that S,A and S,C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.

Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.

After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again.

However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.

After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we

are showing a spanning tree with both edges included.

### PROGRAM

```
import java.util.Scanner;
class Prim
{
    public static void my_prim(int[][] adj,int N)
    {
        int i,j,nv,min,min_cost=0,u=0,v=0;
        int[] visit=new int[N];
        for(i=0;i<N;i++)
        {
            visit[i]=-0;
        }
        visit[0]=1;
        nv=1;
        while(nv<N)
        {
            min=999;
            for(i=0;i<N;i++)
```



```

{ if(visit[i]==1)//visited
{ for(j=0;j<N;j++)
{ if(adj[i][j]<min)
{ min=adj[i][j];
adj[i][j]=999;
u=i;
v=j;
}
}
} if(visit[u]==1
&& visit[v]==0)
{ visit[v]=1;
min_cost+=min;
nv++;
System.out.printf("Edge %d - %d : (%d)\n",u,v,min);
}
}
System.out.println("Cost : " +min_cost);//prints minimum cost
}
}
public class Primsalgo8b
{ public static void main(String[] arg )
{ int[][] adj;
int N,i,j,S,D;
Scanner sc=new Scanner(System.in);
System.out.println("Enter number of nodes in the graph");
N=sc.nextInt();
adj=new int[N][N];
for(i=0;i<N;i++)
for(j=0;j<N;j++)
adj[i][j]=-1;
System.out.println("enter the adjacency matrix");
System.out.println("enter 0 for no connection and weights for connection");
for(i=0;i<N;i++)
{ for(j=0;j<N;j++)
{ adj[i][j]=sc.nextInt();
if(adj[i][j]==0)
adj[i][j]=999;
}
}
Prim.my_prim(adj,N);
}
}

```

**OUTPUT**

Enter number of nodes in the graph



4 enter the adjacency matrix  
enter 0 for no connection and weights for connection  
0 2 2 0  
2 0 0 1  
2 0 0 5  
0 1 5 0  
Edge 0 - 1 : (2)  
Edge 1 - 3 : (1)  
Edge 3 - 2 : (5)  
Cost:8



## **EXPERIMENT 10A**

### **PROGRAM STATEMENT**

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

### CONCEPT

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem.

The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Following Algorithm is used to find shortest path:

given:  $w[i][j]$  is the weighted matrix that contains weights  $k$  is the iteration variable for intermediate vertices

Following is the algorithm

Function floyd

for  $i=1$  to  $n$

for  $j=1$  to  $n$

$w[i][j]=\text{infinity}$ //if entered  $w[i][j] == 0$

for each edge  $(i,j)$  in  $E$

$\text{dmatrix}[i][i] = \text{amatrix}[i][i]$ ;//copy the entered weighted matrix to distance matrix

//calculate distance matrix

for  $k=1$  to  $n$

for  $i=1$  to  $n$

for  $j=1$  to  $n$

if  $(\text{dmatrix}[i][k] + \text{dmatrix}[k][j] < \text{dmatrix}[i][j])$

$\text{dmatrix}[i][j] = \text{dmatrix}[i][k] + \text{dmatrix}[k][j]$ ;

### PROGRAM

```
import java.util.Scanner;
```

```
public class Warshal
```

```
{
```

```
//distance matrix declaration
```

```
private int dmatrix[][];
```

```
//number of vertices
```

```
private int n;
```

```
//parameter if there is no edge or path between 2 vertices
```

```
public static final int INFINITY = 999;
```

```
//constructor to initialize number of vertices
```

```
public Warshal(int n)
```

```
{
```

```
dmatrix = new int[n + 1][n + 1];
```

```
this.n = n;
```

```
}
```

```
//function that calculates shortest path
```

```
public void floydwarshall(int amatrix[][])
```

```
{
```

```
for (int i = 1; i <= n; i++)
```

```
{
```

```
for (int j = 1; j <= n; j++)
```

```
{//assigning the value of entered weighted matrix to distance matrix
```

```
dmatrix[i][j] = amatrix[i][j];
```

```
}
}
for (int k = 1; k <= n; k++)
{
for (int i = 1; i <= n; i++)
{
for (int j = 1; j <= n; j++)
{
//calculating shortest distance
if (dmatrix[i][k] + dmatrix[k][j]
< dmatrix[i][j])
dmatrix[i][j] = dmatrix[i][k]
+ dmatrix[k][j];
}
}
}
for (int source = 1; source <= n; source++)
System.out.print("\t" + source);
System.out.println();
for (int source = 1; source <= n; source++)
{
System.out.print(source + "\t");
for (int destination = 1; destination <= n; destination++)
{
System.out.print(dmatrix[source][destination] + "\t");
}
System.out.println();
}
}
public static void main(String... arg)
{
int wmatrix[][];
int n;
Scanner scan = new Scanner(System.in);
System.out.println("Enter the number of vertices");
n = scan.nextInt();
wmatrix = new int[n + 1][n + 1];
System.out.println("Enter the Weighted Matrix for the graph");
for (int source = 1; source <= n; source++)
{
for (int destination = 1; destination <= n; destination++)
{
wmatrix[source][destination] = scan.nextInt();
if (source == destination)
{
wmatrix[source][destination] = 0;
continue;
}
```

```
}  
if (wmatrix[source][destination] == 0)  
{  
    wmatrix[source][destination] = INFINITY;  
}  
}  
}  
System.out.println("Following matrix shows the shortest distances between every pair of  
vertices");  
Warshal floydwarshall = new Warshal(n);  
//calculating and printing the shortest distance by invoking the function  
floydwarshall.floydwarshall(wmatrix);  
scan.close();  
}  
}
```

### OUTPUT

Enter the number of vertices

4

Enter the Weighted Matrix for the graph

0 0 3 1

2 0 0 1

0 7 0 1

7 0 0 0

Following matrix shows the shortest distances between every pair of vertices

1 2 3 4

1 0 10 3 1

2 2 0 5 1

3 8 7 0 1

4 7 17 10 0

### EXPERIMENT 10B PROGRAM STATEMENT

Write a Java program to implement Travelling Salesman Problem using Dynamic programming

### CONCEPT

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between [Hamiltonian Cycle](#) and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the above graph. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is  $10+25+30+15$  which is 80.

### PROGRAM

```
package traversal;
import java.util.Scanner;
class traversal_demo
{ private int cost[][];
  private int visit[];
  private int n,total_cost;
  traversal_demo()//constructor to initialize the data members
  { cost=new int[10][10];//allocates memory dynamically for cost and visit array
    visit=new int[10];
    total_cost=0;
  }
  public void getdata()//getdata() function used to accept details
  { int i,j;
    Scanner in= new Scanner(System.in);
    System.out.println("Enter No. of Cities: ");
    n=in.nextInt();//accept number of cities
    System.out.println("Enter Cost Matrix");
    for(i=0;i<n;i++)//accept the cost matrix
    { for( j=0;j < n;j++)
      cost[i][j]=in.nextInt();
      visit[i]=0;// all the cities are not visited in the starting
    } in.close();
  }
  public void costfromcity(int city)//mincost() is used to find minimum cost from one city to other
  { int mincity;
    visit[city]=1; //indicates that city is visited
    System.out.println(+city+1+ " ");
    mincity=least_path(city);//finds the minimum path from city
    if(mincity==999)
    { mincity=0;
      System.out.println(mincity+1);
      total_cost+=cost[city][mincity];
      return;
    } costfromcity(mincity);
  }
```

```
}
public int least_path(int c)//function is used to find minimum cost edge
{ int i,nc=999;
int min=999,minadd=0;
for(i=0;i<n;i++)
{ if((cost[c][i]!=0)&&(visit[i]==0))//checks cost of city to other is not zero and city should
not be visited
if(cost[c][i] < min)//checks for cost of city is less than min then it consider its cost
as minimum and its is added to total cost
{ min=cost[c][i];
minadd=cost[c][i];
nc=i;
}
}
if(min!=999)
total_cost+=minadd;
return nc;//returns the visited city back to costfromcity() function
}
public void display();//display the minimum cost
{
System.out.println("Minimum cost:" +total_cost);
}
}
public class traversal
{ public static void main(String args[])
{ traversal_demo tr=new traversal_demo();//object creation
tr.getdata();//invocation of function
System.out.println("The minimum path:");
tr.costfromcity(0);
tr.display();
}
}
```

## OUTPUT

Enter No. of Cities:

4

Enter Cost Matrix

0 1 2 4

1 0 999 1

2 999 0 1

4 1 1 0

The minimum path:

1

2

4

3

1

Minimum cost:5



## **EXPERIMENT 11 PROGRAM STATEMENT**



Find a subset of a given set  $S = \{s_1, s_2, \dots, s_n\}$  of  $n$  positive integers whose sum is equal to a given positive integer  $d$ . For example, if  $S = \{1, 2, 5, 6, 8\}$  and  $d = 9$  there are two solutions  $\{1, 2, 6\}$  and  $\{1, 8\}$ . A suitable message is to be displayed if the given problem instance doesn't have a solution.

## CONCEPT

Subset-Sum Problem is to find a subset of a given set  $S = \{s_1, s_2, \dots, s_n\}$  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 backtracking algorithm, the solutions could be obtained. Some instances of the problem may have no solutions

**Algorithm** SumOfSub( $s, k, r$ )

//Find all subsets of  $w[1 \dots n]$  that sum to  $m$ . The values of  $x[j]$ ,  $1 \leq j < k$ , have already been determined.  $s = \sum_{k=1}^j w[j] * x[j]$  and  $r = \sum_n w[j]$ . The  $w[j]$ 's are in ascending order.

$j=1$   $j=k$

{  $x[k] \leftarrow 1$  //generate left child if  $(s+w[k] = m)$

write  $(x[1 \dots n])$  //subset found else if  $(s + w[k] + w[k+1] \leq m)$

SumOfSub( $s + w[k], k+1, r-w[k]$ ) //Generate right child

if  $((s + r - w[k] \geq m) \text{ and } (s + w[k+1] \leq m))$

{  $x[k] \leftarrow 0$

SumOfSub( $s, k+1, r-w[k]$ )

}

}

## PROGRAM

**package** subset;

**import** java.util.Scanner;

**class** subset

{ **int**  $s[], x[];$  //variables required to store the subset values

**int**  $d, n;$

subset()//constructor

{  $s = \text{new int}[10];$

$x = \text{new int}[10];$

}

**void** read()//function used accept the number of sets and values in the set

{ **int**  $\text{sum} = 0;$

Scanner  $\text{in} = \text{new Scanner}(\text{System.in});$

System.out.println("enter the total number of elements in set");

$n = \text{in.nextInt()};$  //read number of elements in set

System.out.println("enter the set");

**for**(**int**  $i = 1; i \leq n; i++$ )

$s[i] = \text{in.nextInt()};$  //accept the elements into the sets

System.out.println("enter the maximum set value");

$d = \text{in.nextInt()};$  //maximum set value

**for**(**int**  $i = 1; i \leq n; i++$ )

$\text{sum} = \text{sum} + s[i];$  //calculate the sum of all the elements

**if**( $\text{sum} < d$ ) //if sum is less than d or if first

System.out.println("subset is not possible");

**else**

System.out.println("the sets are");

subset\_fun(0, 1, sum);

in.close();

```
}  
void subset_fun(int S,int k,int r)//function to find subset  
{ x[k]=1;  
if((S+s[k])==d)//if current value of subset k element is equal to sum then print the sets  
{ for(int i=1;i<=k;i++)  
if(x[i]==1)  
System.out.println(+s[i]+" ");  
System.out.println();  
} else  
if(S+s[k]+s[k+1]<=d)//if the current subset value with k and k+1 element is less than d  
subset_fun(S+s[k],k+1,r-s[k]);  
if((S+r-s[k]>=d)&&(S+s[k+1]<=d))  
{ x[k]=0;  
subset_fun(S,k+1,r-s[k]);  
}  
}  
}  
public class subset_demo {  
public static void main(String[] args)  
{ subset st=new subset();  
st.read();  
}  
}
```

## OUTPUT

enter the total number of elements in set : 5

enter the set: 1 2 5 6 8

enter the maximum set value : 9

the sets are

1  
2  
6  
1  
8

## EXPERIMENT 12 PROGRAM STATEMENT

Design and implement in java to find all Hamiltonian Cycles in a connected undirected graph G with n vertices using backtracking principle.

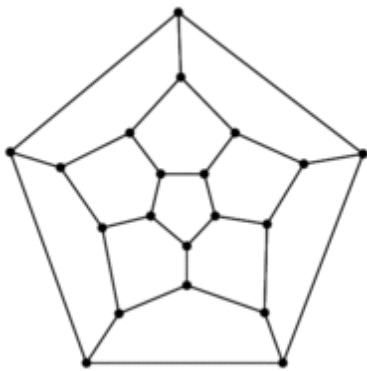
### CONCEPT

A *Hamiltonian cycle*, *Hamiltonian circuit* is a **cycle** that visits each vertex exactly once (except for the vertex that is both the start and end, which is visited twice). A graph that contains a Hamiltonian cycle is called a **Hamiltonian graph**.

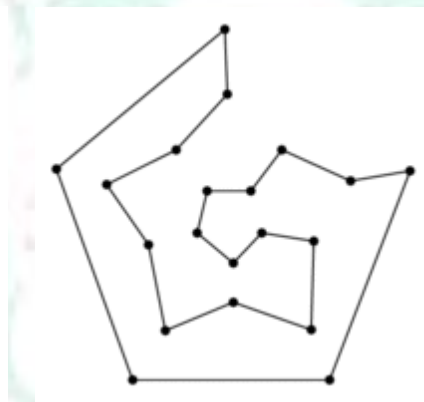
Following is one way of checking whether a graph contains a Hamiltonian Cycle or not.

A Hamiltonian Path in a graph having N vertices is nothing but a permutation of the vertices of the graph  $[v_1, v_2, v_3, \dots, v_{N-1}, v_N]$ , such that there is an edge between  $v_i$  and  $v_{i+1}$  where  $1 \leq i \leq N-1$ . So it can be checked for all permutations of the vertices whether it represents a Hamiltonian Cycle or not.

INPUT



OUTPUT



### PROGRAM

```
import java.util.Arrays;
import java.util.Scanner;
public class HamilDemo {
    int count;
    int path=1;
    void swap(int[] arr, int x, int y) {
        int temp = arr[x];
        arr[x] = arr[y];
        arr[y] = temp;
    }
    void permute(int[] arr, int[][] G) {
        permute(arr, 0, arr.length - 1, G);
    }
    void permute(int[] arr, int i, int n, int[][] cost) {
        int j;
        if (i == n)
        {
            HamilCycle(arr, cost);
        }
    }
}
```

```
else {
    for (j = i; j <= n; j++) {
        swap(arr, i, j);
        permute(arr, i + 1, n, cost);
        swap(arr, i, j); // backtrack
    }
}
}

void HamilCycle(int a[],int[][] G)
{
    count=0;
    for(int i=0;i<a.length-1;i++)
    {
        if(G[a[i]][a[i+1]]!=0)
            count++;
    }
    if(count==a.length-1 && G[a[a.length-1]][a[0]]==1)
    {
        System.out.print("Cycle No " + path + "--->");
        for(int i=0; i<a.length;i++)
            System.out.print(a[i] + " ");
        System.out.print(a[0]);
        System.out.println();
        path++;
    }
}

public static void main(String[] args) {
    Scanner s =new Scanner(System.in);
    System.out.println("Enter the number of nodes in graph");
    int n=s.nextInt();
    int graph[][]=new int[n][n];
    System.out.println("Enter the adjacency matrix");
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            graph[i][j]=s.nextInt();
    int arr[]=new int[n];
    for(int i=0;i<n;i++)
        arr[i]=i;
    System.out.println("All possible Hamiltonian Cycles in graph ");
    new HamilDemo().permute(arr,graph);
}
}
```

## OUTPUT

```
Output - HamilDemo (run) ×
run:
Enter the number of nodes in graph
5
Enter the adjacency matrix
0 1 0 1 1
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
1 0 1 1 0
All possible Hamiltonian Cycles in graph
Cycle No 1--->0 1 2 4 3 0
Cycle No 2--->0 3 1 2 4 0
Cycle No 3--->0 3 4 2 1 0
Cycle No 4--->0 4 2 1 3 0
Cycle No 5--->1 0 3 4 2 1
Cycle No 6--->1 2 4 3 0 1
Cycle No 7--->1 2 4 0 3 1
Cycle No 8--->1 3 0 4 2 1
Cycle No 9--->2 1 0 3 4 2
Cycle No 10--->2 1 3 0 4 2
Cycle No 11--->2 4 0 3 1 2
Cycle No 12--->2 4 3 0 1 2
Cycle No 13--->3 1 2 4 0 3
Cycle No 14--->3 0 1 2 4 3
Cycle No 15--->3 0 4 2 1 3
Cycle No 16--->3 4 2 1 0 3
Cycle No 17--->4 2 1 3 0 4
Cycle No 18--->4 2 1 0 3 4
Cycle No 19--->4 3 0 1 2 4
Cycle No 20--->4 0 3 1 2 4
BUILD SUCCESSFUL (total time: 23 seconds)
```

