1. Use the following code to illustrate the Bankers algorithm and explain what is occurring at each step. Annotate your output to illustrate what is happening.

Bankers.java

```
1 //on a high level:
2 //Banker's algorithm keeps deadlock at bay by tracking resources
3 //and allocating appropriately. It does this by tracking 3 things:
4 //The maximum a process can allocate (MAX)
5 //The amount it has already allocated (ALLOCATE)
6 //The resources available to use (AVAIL)
7 //It allows resources to be allocated if the amount requested (NEED) is less
8 //than or equal to the amount available. If not, it waits until they are.
9
10 import java.util.Scanner;
11
12 public class Bankers{
13     //Banker's algorithm works on 3 things:
14     //This implementation includes extras: NEED and 2 helper variables
15     //np,nr store user input and put it into the right place in input()
16     private int need[][],allocate[][],max[][],avail[][],np,nr;
17
18     //input() handles adding all the variables to the arrays:
19     //MAX,ALLOCATE,AVAIL
20     private void input(){
21      //Scans user input
22      Scanner sc=new Scanner(System.in);
23      System.out.print("Enter no. of processes and resources : ");
24      //Sets length/width of the 2D arrays using user input
25      np=sc.nextInt();  //no. of process
```

```
26      nr=sc.nextInt();  //no. of resources
27      need=new int[np][nr];  //initializing arrays
28      max=new int[np][nr];
29      allocate=new int[np][nr];
30      avail=new int[1][nr];
31
32      //uses user input to define 2D arrays
33      System.out.println("Enter allocation matrix -->");
34      for(int i=0;i<np;i++)
35           for(int j=0;j<nr;j++)
36          allocate[i][j]=sc.nextInt();  //allocation matrix
37
38      System.out.println("Enter max matrix -->");
39      for(int i=0;i<np;i++)
40           for(int j=0;j<nr;j++)
41          max[i][j]=sc.nextInt();  //max matrix
42
43         System.out.println("Enter available matrix -->");
44         for(int j=0;j<nr;j++)
45          avail[0][j]=sc.nextInt();  //available matrix
46         //closes input
47         sc.close();
48      }
49
50      private int[][] calc_need(){
51         for(int i=0;i<np;i++)
52           for(int j=0;j<nr;j++)  //calculating need matrix
53            //subtracts max it CAN request from what it's already allocated to
54            //find remainders
55            need[i][j]=max[i][j]-allocate[i][j];
```

```
56
57          return need;
58      }
59
60      private boolean check(int i){
61          //checking if all resources for ith process can be allocated
62          for(int j=0;j<nr;j++)
63          //if available resources are less than needed resources, return false
64          if(avail[0][j]<need[i][j])
65              return false;
66      //else return true
67      return true;
68      }
69
70      public void isSafe(){
71          //calls input to gather user data
72          input();
73          //calls calc_need to calculate what each process wants
74          calc_need();
75          //each process boolean to see if it gets wanted resources
76          boolean done[]=new boolean[np];
77          int j=0;
78
79          while(j<np){  //until all process allocated
80          boolean allocated=false;
81          for(int i=0;i<np;i++)
82           //calls check to see if resources can be allocated
83           if(!done[i] && check(i)){  //trying to allocate
84               for(int k=0;k<nr;k++)
85               //allocates resources to the process, thus subtracting from
```

```
86              //available resources.
87              avail[0][k]=avail[0][k]-need[i][k]+max[i][k];
88            System.out.println("Allocated process : "+i);
89            //tells the allocation array that it's successful
90            allocated=done[i]=true;
91                  j++;
92                }
93             //if no allocation occured, break out of forloop; it failed
94             if(!allocated) break;  //if no allocation
95          }
96          if(j==np)  //if all processes are allocated
97           //everything went okay!
98           System.out.println("\nSafely allocated");
99          else
100          //not so much this time
101          System.out.println("All proceess cant be allocated safely");
102     }
103
104     public static void main(String[] args) {
105        //calls main logic
106        new Bankers().isSafe();
107     }
108 }
```

Output

```
////SUCCESSFUL

Enter no. of processes and resources : 2 1 //2 processes, 1 resource
Enter allocation matrix -->
2 2 //already has 2 allocated
Enter max matrix -->
4 4 //can go up to 4
Enter available matrix -->
6 6 //there's 6 available for both processes
Allocated process : 0
Allocated process : 1
Safely allocated //(4 - 2) < 6, you're good to go


////UNSUCCESSFUL

Enter no. of processes and resources : 2 1 //2 processes, 1 resource
Enter allocation matrix -->
2 2 //already has 2 allocated
Enter max matrix -->
4 4 //can go up to 4
Enter available matrix -->
1 1 //only has 1 available for both
All proceess cant be allocated safely //(4 - 2) > 2, fails
```

2. Give examples of inputs where a safe allocation of processes occurs and one where processes cannot be allocated safely.

   See Output, above. If `(allocation - max)` >`available`**, allocation fails.**

3. What conditions cause the former to happen? The latter? Clearly indicate these in your writeup. (e.g., for all i, j, when max[i][j] <avail[i][j])

   See answer 2. Output contains both a failure and success and explains why; resources available were less than resources needed by a process.

4. From a big picture perspective, why is this implementation of resource allocation so widely appreciated?

   It's so simple that it can be understood and implemented intuitively, but it solves a majority of the problems associated with resource allocation.