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.

   First, it calls isSafe(), which runs through every other function in the class to check if something is safe. First it gathers user input, then it calculates how much each process needs to have. It then initializes a boolean array with the same number of booleans as there are processes. It iterates through the resources needed to check if there's enough resources to fulfill each processes' needs. If it succeeds, it prints a success message. Otherwise, it prints a failure message.

   There is a line-by-line explanation of the code below.

Bankers.java

```java
//on a high level:
//Banker's algorithm keeps deadlock at bay by tracking resources
//and allocating appropriately. It does this by tracking 3 things:
//The maximum a process can allocate (MAX)
//The amount it has already allocated (ALLOCATE)
//The resources available to use (AVAIL)
//It allows resources to be allocated if the amount requested (NEED) is less
//than or equal to the amount available. If not, it waits until they are.

import java.util.Scanner;

public class Bankers{
    //Banker's algorithm works on 3 things:
```

```java
    //This implementation includes extras: NEED and 2 helper variables
    //np,nr store user input and put it into the right place in input()
    private int need[][],allocate[][],max[][],avail[][],np,nr;

    //input() handles adding all the variables to the arrays:
    //MAX,ALLOCATE,AVAIL
    private void input(){
     //Scans user input
     Scanner sc=new Scanner(System.in);
     System.out.print("Enter no. of processes and resources : ");
     //Sets length/width of the 2D arrays using user input
     np=sc.nextInt();  //no. of process
     nr=sc.nextInt();  //no. of resources
     need=new int[np][nr];  //initializing arrays
     max=new int[np][nr];
     allocate=new int[np][nr];
     avail=new int[1][nr];

     //uses user input to define 2D arrays
     System.out.println("Enter allocation matrix -->");
     for(int i=0;i<np;i++)
          for(int j=0;j<nr;j++)
         allocate[i][j]=sc.nextInt();  //allocation matrix

     System.out.println("Enter max matrix -->");
     for(int i=0;i<np;i++)
          for(int j=0;j<nr;j++)
         max[i][j]=sc.nextInt();  //max matrix

        System.out.println("Enter available matrix -->");
    for(int j=0;j<nr;j++)
     avail[0][j]=sc.nextInt();  //available matrix
    //closes input
    sc.close();
}

private int[][] calc_need(){
   for(int i=0;i<np;i++)
     for(int j=0;j<nr;j++)  //calculating need matrix
      //subtracts max it CAN request from what it’s already allocated to
      //find remainders
      need[i][j]=max[i][j]-allocate[i][j];

   return need;
}

private boolean check(int i){
   //checking if all resources for ith process can be allocated
   for(int j=0;j<nr;j++)
   //if available resources are less than needed resources, return false
   if(avail[0][j]<need[i][j])
      return false;
//else return true
return true;
}

public void isSafe(){
   //calls input to gather user data
   input();
   //calls calc_need to calculate what each process wants
```

```
       calc_need();
       //each process boolean to see if it gets wanted resources
       boolean done[]=new boolean[np];
       int j=0;

       while(j<np){  //until all process allocated
       boolean allocated=false;
       for(int i=0;i<np;i++)
        //calls check to see if resources can be allocated
        if(!done[i] && check(i)){  //trying to allocate
            for(int k=0;k<nr;k++)
            //allocates resources to the process, thus subtracting from
            //available resources.
            avail[0][k]=avail[0][k]-need[i][k]+max[i][k];
         System.out.println("Allocated process : "+i);
         //tells the allocation array that it's successful
         allocated=done[i]=true;
               j++;
             }
          //if no allocation occured, break out of forloop; it failed
          if(!allocated) break;  //if no allocation
       }
       if(j==np)  //if all processes are allocated
        //everything went okay!
        System.out.println("\nSafely allocated");
       else
        //not so much this time
        System.out.println("All proceess cant be allocated safely");
    }

```

```java
    public static void main(String[] args) {
       //calls main logic
       new Bankers().isSafe();
    }
}
```

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. It guarantees that resources are not exhausted, and that deadlock never occurs; it can be used to help schedulers work properly, and it prevents the deadliest errors in resource allocation, all in a function which only takes 70 ish lines of Java to implement. Its elegance and guarantees are what makes it so famous.