1. Banker's Algorithm

The Core Concept: The Banker's Algorithm 🏦

Imagine you're a banker who has a certain amount of money (let's say $1000). Several customers have taken loans from you, and each has a maximum credit limit they might need.

The **Banker's Algorithm** is a strategy the banker uses to avoid a "deadlock." A **deadlock** is a situation where all customers need more money, but the banker doesn't have enough to satisfy any single one of them. Now, no one can proceed, and the system freezes.

To prevent this, the banker follows a simple rule: **Only grant a request for more money if you are certain that even after giving it, you can still find a sequence to let every customer finish their work and repay their loan.**

A **"Safe State"** is a state where the banker (the Operating System) has enough resources to find at least one sequence of customers (processes) that can all finish. If no such sequence exists, the system is in an **"Unsafe State,"** which *could* lead to a deadlock.

The algorithm works by pretending to grant a process's request, and then checking if a safe sequence can still be found.

Key Terms in the Algorithm

To understand the code, you need to know these four data structures:

1. **Allocation**: What each process **currently holds**.
   - *Analogy*: How much money each customer has already borrowed.
2. **Max**: The maximum amount each process **might ever need**.
   - *Analogy*: The pre-approved credit limit for each customer.
3. **Available**: The resources the system **currently has free**.
   - *Analogy*: The money the banker has in the vault right now.
4. **Need**: The remaining resources each process **still needs** to finish its job. This isn't given; we calculate it.
   - **Formula**: `Need = Max - Allocation`
   - *Analogy*: A customer's credit limit minus what they've already borrowed.

Explaining the Java Code

Your Java code implements this exact logic. Let's walk through it step-by-step.

`public class BankersAlgorithm { ... }`

This is the main container for our program. It holds all the variables (the matrices and arrays) and the methods to work with them.

Java

```
private int n; // Number of processes (customers)
private int m; // Number of resource types (e.g., CPU, RAM, Printers)
private int[][] allocation; // The Allocation matrix
private int[][] max; // The Max matrix
private int[][] need; // The Need matrix
private int[] available; // The Available array
```

These variables are defined at the class level so all methods (`input` , `isSafe`) can access them.

`public void input()` - Gathering Information

This method's job is to ask the user for all the initial data, just like a banker would take down details from new customers.

Java

```
// Asks for the number of processes (n) and resources (m).
System.out.print("Enter no. of process: ");
n = sc.nextInt();
System.out.print("Enter no. of resources: ");
m = sc.nextInt();

// Initializes the matrices and array with the correct dimensions.
allocation = new int[n][m];
// ... and so on for max, need, and available.

// Fills the Allocation, Max, and Available data from user input
// using nested for-loops to go through each cell of the matrices.
System.out.println("Enter Allocation Matrix :");
// ...
System.out.println("Enter Max Matrix: ");
// ...
System.out.println("Enter Available Resources: ");
// ...
```

The most important calculation happens at the end of this method:

The most important calculation happens at the end of this method:

Java

```
// Calculating Need Matrix
for(int i = 0; i < n; i++){
    for(int j = 0; j < m; j++){
        need[i][j] = max[i][j] - allocation[i][j];
    }
}
```

Here, the code calculates the `Need` matrix for every process by subtracting what it **has** (`Allocation`) from what it might **need** (`Max`). This is the crucial first step.

`public void isSafe()` - The Safety Check ✅

This is the heart of the algorithm. It determines if the system is in a safe state.

Java

```
boolean[] finish = new boolean[n]; // Tracks which processes have finished. Initial
int[] work = available.clone();    // A temporary copy of 'available'. We can't mess
int[] safeSeq = new int[n];        // This array will store the final safe sequence.
int count = 0;                     // Counts how many processes have been added to t
```

The logic then enters a `while` loop that continues until all processes are in the safe sequence (`count < n`).

Java

```java
while(count < n){
    boolean found = false; // A flag to check if we found a process that can run i

    // 1. Find a process that can run.
    for(int i = 0; i < n; i++){ // Loop through all processes.
        if(!finish[i]){ // Check only if the process is not yet finished.

            // 2. Check if the process's 'Need' can be satisfied by what's 'Availa
            int j;
            for(j = 0; j < m; j++){
                if (need[i][j] > work[j])
                    break; // If even one resource type is not available, this proc
            }

            // 3. If the process CAN run...
            if(j == m){ // This means the inner loop completed without breaking. A

                // a. Pretend it runs and releases its resources.
                // Add its 'allocation' back to the 'work' array.
                for (int k = 0; k < m; k++){
                    work[k] += allocation[i][k];
                }

                // b. Record this process in our safe sequence.
                safeSeq[count++] = i;

                // c. Mark this process as finished.
```

If the `while` loop completes successfully (meaning `count` reached `n`), it means we found a safe sequence for all processes.

Java

```
// This code only runs if the loop finishes.
System.out.println("System is in Safe state.");
System.out.print("Safe Sequence : ");
for(int i = 0; i < n; i++){
    System.out.print("P"+safeSeq[i]+" "); // Print the discovered safe sequence.
}
```

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

This is the entry point of the program. It's very simple:

1. It creates a new `BankersAlgorithm` object.
2. It calls `ba.input()` to get the data.
3. It calls `ba.isSafe()` to run the check and print the result.

And that's it! The code elegantly translates the banker's cautious lending strategy into a robust algorithm for operating systems to prevent deadlocks.