MAKERERE UNIVERSITY

COLLEGE OF COMPUTING AND INFORMATION SCIENCES(CoCIS)

CSC 2200 OPERATING SYSTEMS

BSSE EVENING GROUP H

Name	Registration number	Student number
Bataringaya Bridget	23/U/07471/EVE	2300707471
Johnson Makmot Kabira	23/U/26794/EVE	2300726794
Mwesigwa Isaac	23/U/12539/PS	2300712539
Mariam Wambui	23/U/18494/PS	2300718494
Jonathan Katongole	23/U/27072/EVE	2300727072

```java
import java.util.*;

public class BankersAlgorithm {

    // Function to implement Banker's Algorithm
    public static boolean bankersAlgorithm(int[] available, int[][] max, int[][]
allocation) {
        int n = allocation.length;
        int m = available.length;
        int[] work = Arrays.copyOf(available, m);
        boolean[] finish = new boolean[n];

        // Initialize finish array
        for (int i = 0; i < n; i++) {
            finish[i] = true;
            for (int j = 0; j < m; j++) {
                if (allocation[i][j] != 0) {
                    finish[i] = false;
                    break;
                }
            }
        }

        // Find an index i such that both conditions are satisfied
        while (true) {
            boolean found = false;
            for (int i = 0; i < n; i++) {
                if (!finish[i]) {
                    boolean canAllocate = true;
                    for (int j = 0; j < m; j++) {
                        if (max[i][j] - allocation[i][j] > work[j]) {
                            canAllocate = false;
                            break;
                        }
                    }
                    if (canAllocate) {
                        for (int j = 0; j < m; j++) {
                            work[j] += allocation[i][j];
                        }
                        finish[i] = true;
                        found = true;
                    }
                }
            }
            if (!found) {
                break;
            }
        }

        // Check if all processes are finished
        for (boolean f : finish) {
            if (!f) {
                return false;
            }
        }
        return true;
    }

    // Function to implement Safe State Algorithm
    public static boolean isSafeState(int[] available, int[][] max, int[][]
allocation) {
        int n = allocation.length;
        int m = available.length;
        int[] work = Arrays.copyOf(available, m);
        boolean[] finish = new boolean[n];

        // Initialize finish array
        Arrays.fill(finish, false);

        // Find a safe sequence
        int[] safeSequence = new int[n];
        int count = 0;
        while (count < n) {
            boolean found = false;
            for (int i = 0; i < n; i++) {
                if (!finish[i]) {
                    boolean canAllocate = true;
                    for (int j = 0; j < m; j++) {
                        if (max[i][j] - allocation[i][j] > work[j]) {
                            canAllocate = false;
                            break;
                        }
                    }
                    if (canAllocate) {
                        for (int j = 0; j < m; j++) {
                            work[j] += allocation[i][j];
                        }
                        safeSequence[count++] = i;
                        finish[i] = true;
                        found = true;
                    }
                }
            }
            if (!found) {
                return false;
            }
        }

        // Print safe sequence
        System.out.println("Safe sequence:");
        for (int i = 0; i < n; i++) {
            System.out.print("P" + safeSequence[i] + " ");
        }
        System.out.println();
        return true;
    }

    // Function to implement Deadlock Detection Algorithm
    public static boolean detectDeadlock(int[] available, int[][] max, int[][]
allocation) {
        int n = allocation.length;
        int m = available.length;
        int[] work = Arrays.copyOf(available, m);
        boolean[] finish = new boolean[n];

        // Initialize finish array
        for (int i = 0; i < n; i++) {
            finish[i] = true;
            for (int j = 0; j < m; j++) {
                if (allocation[i][j] != 0) {
                    finish[i] = false;
                    break;
                }
            }
        }

        // Find an index i such that both conditions are satisfied
        while (true) {
            boolean found = false;
            for (int i = 0; i < n; i++) {
                if (!finish[i]) {
                    boolean canAllocate = true;
                    for (int j = 0; j < m; j++) {
                        if (max[i][j] - allocation[i][j] > work[j]) {
```

```java
                            canAllocate = false;
                            break;
                        }
                    }
                    if (canAllocate) {
                        for (int j = 0; j < m; j++) {
                            work[j] += allocation[i][j];
                        }
                        finish[i] = true;
                        found = true;
                    }
                }
            }
            if (!found) {
                break;
            }
        }

        // Check if all processes are finished
        for (boolean f : finish) {
            if (!f) {
                return true; // Deadlock detected
            }
        }
        return false; // No deadlock
    }

    public static void main(String[] args) {
        // Example data
        int[] available = {3, 3, 2};
        int[][] max = {
            {7, 5, 3},
            {3, 2, 2},
            {9, 0, 2},
            {2, 2, 2},
            {4, 3, 3}
        };
        int[][] allocation = {
            {0, 1, 0},
            {2, 0, 0},
            {3, 0, 2},
            {2, 1, 1},
            {0, 0, 2}
        };
        // Banker's Algorithm
        System.out.println("Banker's Algorithm Result: " +
bankersAlgorithm(available, max, allocation));

        // Safe State Algorithm
        System.out.println("Safe State Algorithm Result: " +
isSafeState(available, max, allocation));

        // Deadlock Detection Algorithm
        System.out.println("Deadlock Detection Algorithm Result: " +
detectDeadlock(available, max, allocation));
    }
}
```

```
Banker's Algorithm Result: true

Safe sequence:

P1 P3 P4 P0 P2

Safe State Algorithm Result: true

Deadlock Detection Algorithm Result: false
```