

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
|  | | Banker's Algorithm Project | | | | |  | |
|  |  | | | | | | |  |
|  | | | |  |  | | | |
|  | | | | Atheer Ghazi al-Harbi |  | | | |
|  | | | | December 12, 2024—Operating systems—Instructor: Modhi Alsobeihy |  | | | |
|  | | |  | | |  | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | | |  |  | | |  |
|  | I. Overview | | | | | | |  |
|  |  | | |  |  | | |  |
|  |  | |  | | |  | |  |
|  |  |  | * **Purpose:**     This project implements the Banker's Algorithm, a deadlock avoidance algorithm used in operating systems for resource allocation. The algorithm aims to prevent deadlocks by ensuring that the system is always in a safe state, where there exists a sequence in which all processes can finish executing without waiting indefinitely for resources.   * **Implementation Details:**    This Java implementation uses arrays and matrices to represent the system's state. The core algorithm is implemented using iterative checks to find a safe sequence. The system uses command-line arguments as input which are validated and processed. | | |  |  |  |
|  | | | | |
|  |  |  |  |
|  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Decorative | |  |  | | |  |  | |
|  | | II. System Design | | | | |  | |
|  | |  |  | | |  |  | |
|  | Data Structures: int[] available: An array representing the number of available units for each resource type.  int[][] max: A matrix where max[i][j] represents the maximum number of units of resource type j that process i may request.  int[][] allocation: A matrix where allocation[i][j] represents the number of units of resource type j currently allocated to process i.  int[][] need: A matrix derived from max and allocation, where need[i][j] represents the remaining units of resource type j needed by process i. | | | | | | |  |
|  | Modules | | |  | Input/Output: | | |  |
|  | The code consists of a main class that handles input, validation, and algorithm execution, along with helper functions for input, validation, and matrix operations. | | |  | The program accepts input via command-line arguments, the output includes the resource matrices, a deadlock detection result (true/false), and a safe sequence (if one exists). Error messages are displayed for invalid input. | | |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | REPORT TITLE | PAGE 3 |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **The code,**  **How to Use,**  **and testing** |  | |
|  |  | |  |
|  | Decorative |  | |

|  |  |  |  |
| --- | --- | --- | --- |
|  | REPORT TITLE | PAGE 4 |  |

**The code**

**main Method:**Parses command-line arguments, performs input validation, and executes the Banker's Algorithm.

**public static void main(String[] args) {**

char use;

do {

System.out.println("\*\*\*\*\*\*Banker's Algorithm\*\*\*\*\*\*");

System.out.print("Enter 'y' to use, 'n' to exit: ");

use = in.next().charAt(0);

switch (use) {

case 'y':

System.out.print("Enter the number of processes: ");

int n\_P = in.nextInt();

System.out.print("Enter the number of resources: (the total resources)");

int n\_R = in.nextInt();

int[] available = available(n\_R);

int[][] max = max(n\_P, n\_R);

int[][] allocation = allocation(n\_P, n\_R);

if (!isValid(allocation, max, available, n\_P, n\_R)) {

in.close();

return;

}

int[][] need = need(allocation, max, n\_P, n\_R);

printMatrices(allocation, need, max, available, n\_P, n\_R);

boolean deadlock = isDeadlock(allocation, max, need, available, n\_P, n\_R);

System.out.println("\nDeadlock detected: " + deadlock);

System.out.println("Do you want for Pn to requast Rn (y/n)");

char Q=in.next().charAt(0);

if(Q == 'n'){

break;

}else{

request(allocation, max, need, available, n\_P, n\_R);

}

break;

case 'n':

System.out.println("Thank you for using this tool!");

System.out.println("Made by Atheer :)");

break;

default:

System.out.println("Invalid choice.");

}

} while (use != 'n');

in.close();

}

**available, max, allocation Methods:**Input methods to read and process command line arguments.

**static int[] available(int n\_R) {**

int[] available = new int[n\_R];

System.out.println("\nEnter available resources (separated by spaces): ");

for (int r = 0; r < n\_R; r++) {

System.out.print("R" + r + " ");

}

System.out.println();

for (int i = 0; i < n\_R; i++) {

available[i] = in.nextInt();

}

return available;

}

**static int[][] max(int n\_P, int n\_R) {**

int[][] max = new int[n\_P][n\_R];

System.out.println("Enter max matrix (each row represents a process):");

for (int r = 0; r < n\_R; r++) {

System.out.print(" R" + r + " ");

}

System.out.println();

for (int i = 0; i < n\_P; i++) {

System.out.print("P" + i + ": ");

for (int j = 0; j < n\_R; j++) {

max[i][j] = in.nextInt();

}

}

return max;

}

**static int[][] allocation(int n\_P, int n\_R) {**

int[][] allocation = new int[n\_P][n\_R];

System.out.println("\nEnter allocation matrix :");

for (int r = 0; r < n\_R; r++) {

System.out.print(" R" + r + " ");

}

System.out.println();

for (int i = 0; i < n\_P; i++) {

System.out.print("P" + i + ": ");

for (int j = 0; j < n\_R; j++) {

allocation[i][j] = in.nextInt();

}

}

return allocation;

}

* **need Method:** Calculates the need matrix (max - allocation)
* **, isValidAllocation, isValidAvailable Methods:** Validate the input matrices against constraints of the Banker's Algorithm.

**static int[][] need(int[][] allocation, int[][] max, int n\_P, int n\_R) {**

int[][] need = new int[n\_P][n\_R];

for (int i = 0; i < n\_P; i++) {

for (int j = 0; j < n\_R; j++) {

need[i][j] = max[i][j] - allocation[i][j];

} } return need;

}

**static boolean isValid(int[][] allocation, int[][] max, int[] available, int n\_P, int n\_R) {**

if (!isValidAvailable(allocation, available, n\_P, n\_R)) { return false;}

if (!isValidAllocation(allocation, max, n\_P, n\_R)) { return false; }

return true;}

**static boolean isValidAvailable(int[][] allocation, int[] available, int n\_P, int n\_R) {**

for (int i = 0; i < n\_P; i++) {

for (int j = 0; j < n\_R; j++) {

if (allocation[i][j] > available[j]) {

System.err.println("Invalid input: Allocation for P" + i + " on R" + j + " exceeds available resources.");

return false;

}

}

}

return true;

}

**static boolean isValidAllocation(int[][] allocation, int[][] max, int n\_P, int n\_R) {**

for (int i = 0; i < n\_P; i++) {

for (int j = 0; j < n\_R; j++) {

if (allocation[i][j] > max[i][j]) {

System.err.println("Invalid allocation:

Allocated resources exceed maximum needs for process P"

+ i + ", resource R" + j);

return false;

} } } return true;

}

* **isDeadlock Method:** Implements a simplified Banker's Algorithm to check for a deadlock by searching for a safe sequence and print it.
* **printMatrices Method:** Prints the resource matrices to the console in a user-friendly format.

**static void printMatrices(int[][] allocation,int[][] need,int[][] max,int[] available,int n\_P,int n\_R) {**

System.out.println("\*\*\*\*\*\* here is your Matrices \*\*\*\*\*\*");

System.out.println("\nAllocation Matrix:");

printMatrix(allocation, n\_P, n\_R);

System.out.println("\nMaximum Matrix:");

printMatrix(max, n\_P, n\_R);

System.out.println("\nNeed Matrix (Max - Allocation):");

printMatrix(need, n\_P, n\_R);

System.out.println("\nAvailable Resources:");

for (int r = 0; r < n\_R; r++) {

System.out.print("R" + r + " ");

} System.out.println();

for (int i = 0; i < n\_R; i++) {

System.out.print(available[i] + " ");

} System.out.println("\n\*\*\*\*\*\*\*\*\*\*\*\n");

}

**static void printMatrix(int[][] matrix, int rows, int cols) {**

for (int r = 0; r < cols; r++) {

System.out.print(" R" + r + " ");

}System.out.println();

for (int i = 0; i < rows; i++) {

System.out.print("P" + i + ": ");

for (int j = 0; j < cols; j++) {

System.out.print(matrix[i][j] + " ");

}System.out.println();

}}

**static boolean isDeadlock(int[][] allocation, int[][] max, int[][] need, int[] available, int n\_P, int n\_R) {**

boolean[] finished = new boolean[n\_P

int[] work = available.clone();

int[] safeSequence = new int[n\_P];

int count = 0;

while (count < n\_P) {

boolean foundProcess = false;

for (int i = 0; i < n\_P; i++) {

if (!finished[i]) {

boolean canFinish = true;

for (int j = 0; j < n\_R; j++) {

if (need[i][j] > work[j]) {

canFinish = false;

break; }}

if (canFinish) {

for (int j = 0; j < n\_R; j++) {

work[j] += allocation[i][j];

}

safeSequence[count++] = i;

finished[i] = true;

foundProcess = true;

}}}

if (!foundProcess) {

System.out.println("Deadlock detected! No safe sequence exists.");

return false;

}}

System.out.print("Safe sequence: [");

for (int i = 0; i < n\_P; i++) {

System.out.print("P" + safeSequence[i]);

if (i < n\_P - 1) {

System.out.print(", ");

}

}

System.out.println("]");

return true;

}

**request Method:** This method simulates a process requesting additional resources.

**static void request(int[][] allocation, int[][] max, int[][] need, int[] available, int n\_P, int n\_R) {**

System.out.println("\nProcess Request:");

System.out.print("Enter the process number (0 to n-1): \n ");

int P = in.nextInt();

int[] request = new int[n\_R];

// i made this because i don't want to chang the OGs arrays

int[][] new\_allocation = allocation.clone();

int[][] new\_max = max.clone();

int[][] new\_need = need.clone();

int[] new\_available = available.clone();

// read form the user

System.out.println("Enter the request for each resource (separated by spaces): ");

for (int i = 0; i < n\_R; i++) {

request[i] = in.nextInt();

}

// cheacking the 2 Condition

boolean validRequest = true;

for (int i = 0; i < n\_R; i++) {

if (request[i] >= new\_need[P][i] || request[i] >= new\_available[i]) {

validRequest = false;

break;

}

}

// cheacking for the last Condition

if (validRequest) {

for (int i = 0; i < n\_R; i++) {

new\_available[i] -= request[i];

new\_allocation[P][i] += request[i];

new\_need[P][i] -= request[i];

}

if (isDeadlock(new\_allocation, new\_max, new\_need, new\_available, n\_P, n\_R)) {

System.out.println("Request granted. System is in a safe state.");

} else {

System.out.println("Request cannot be granted. System will enter an unsafe state.");

}

} else {

System.out.println("Invalid request. Request exceeds need or available resources.");

}

}

}

**How to Use**

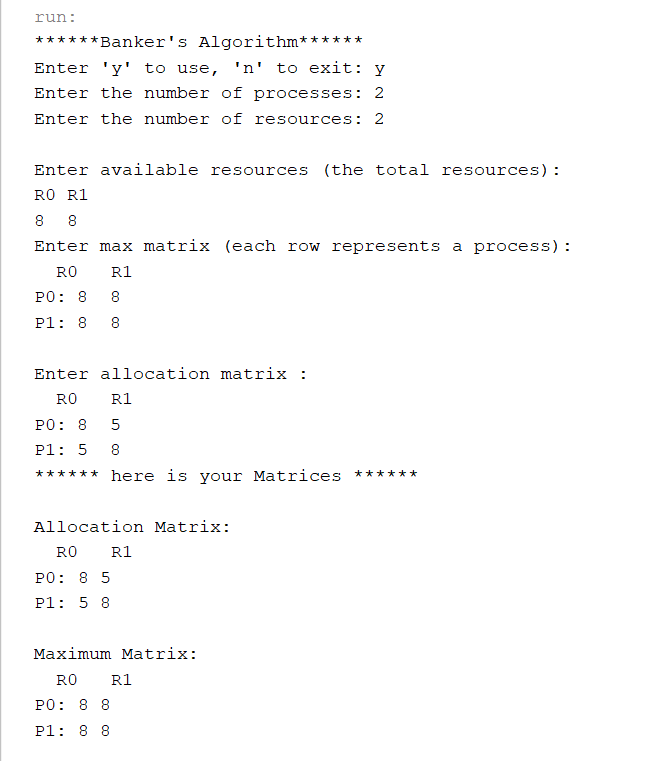
1. The program will guide you through the input process. You will be prompted to enter:
   * The number of processes.
   * The number of resources.
   * The available resources for each resource type (space-separated values).
   * The maximum resource needs matrix (row by row, space-separated values).
   * The current allocation matrix (row by row, space-separated values).
2. Review Results:

After you enter the input, the program will display the resource matrices, the deadlock detection result, the safe sequence (if one exists), and will allow you to simulate a resource request.

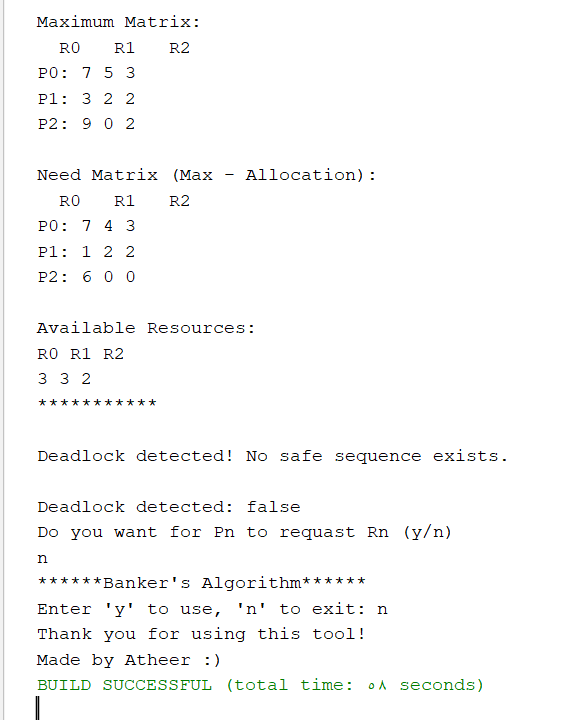
1. Error Handling:

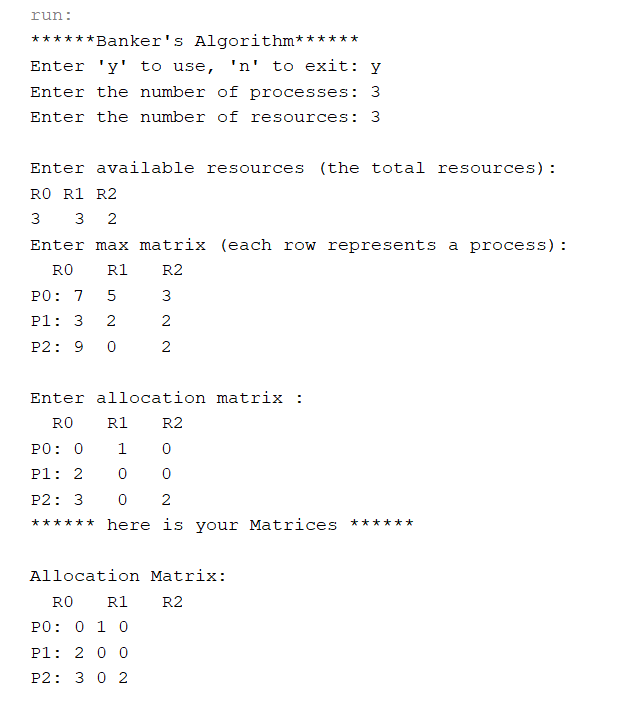
The program includes error handling. If you enter invalid data (e.g., negative numbers, allocation exceeding max needs or available resources), you'll see an error message, and the program will stop.

**Testing**

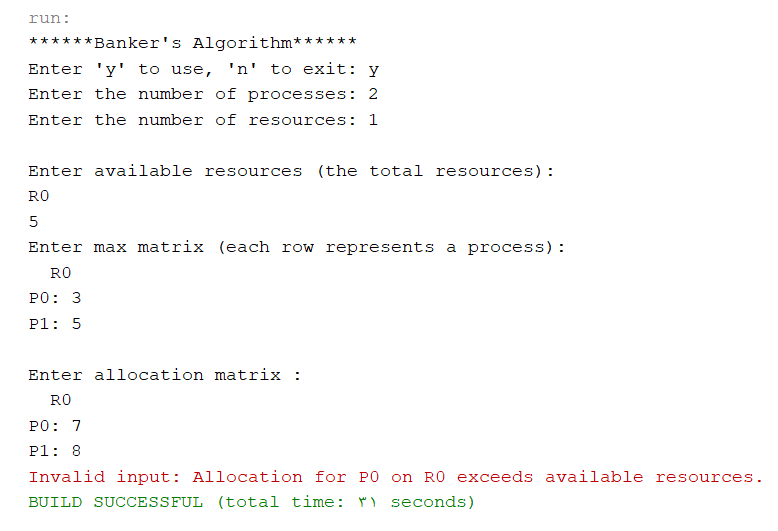
* **Test Case 1 (Safe State):**A screenshot of a computer program

  Description automatically generated
* **Test Case 2 (Deadlock):**

****

****

* **Test Case 3 (Invalid Input):**

****