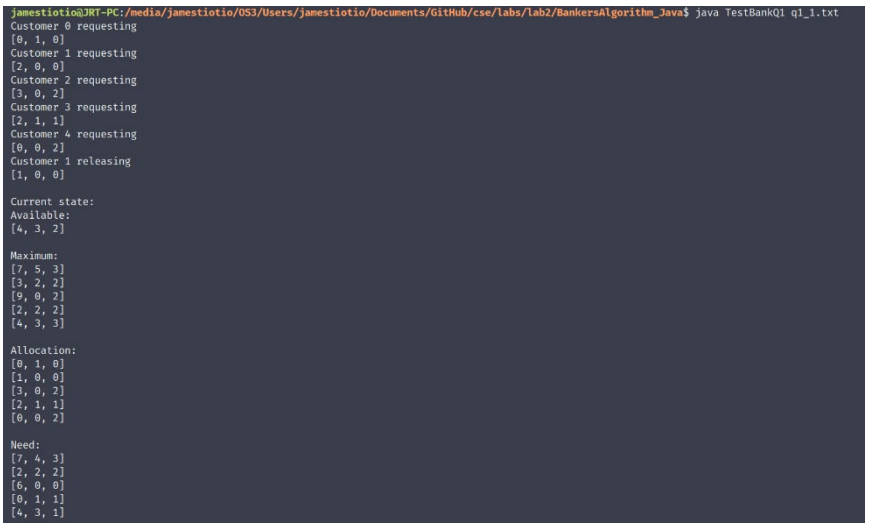
SUTD 2021 50.005 Lab 2 Report

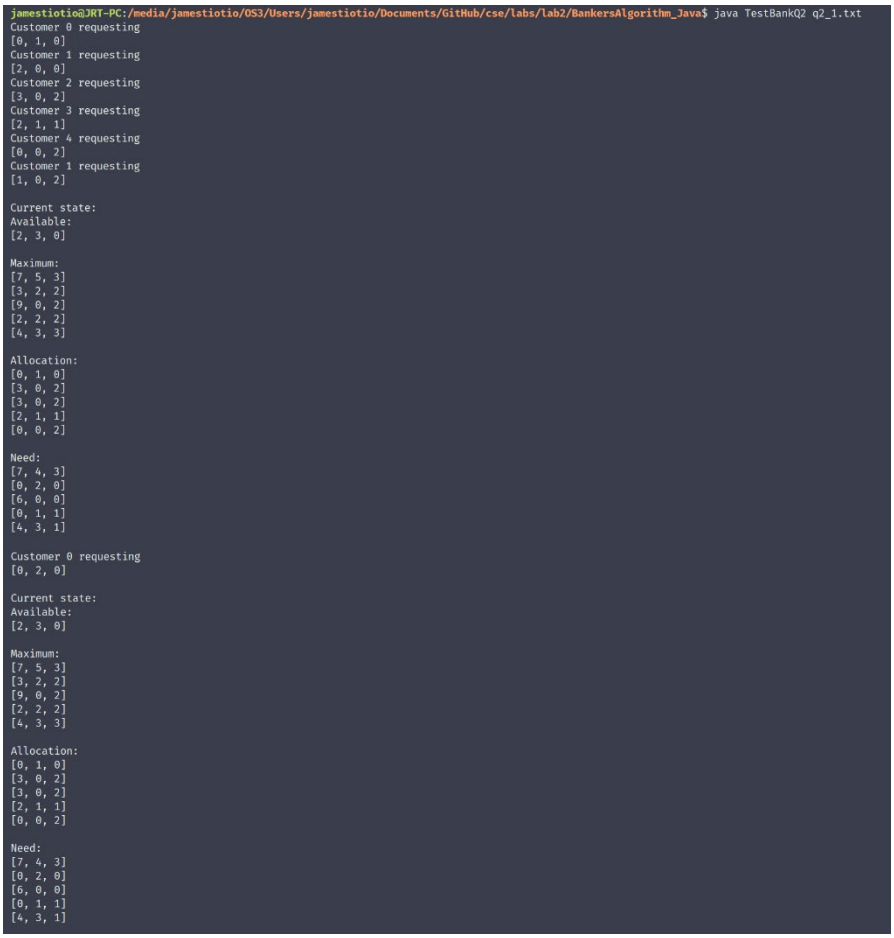
James Raphael Tiovalen

Lab 2 `Banker.java` Screenshots

- Q1 Output:



- Q2 Output:



- Proof that the code passes the test cases given by the checker files:

```
jamestiotio@JRT-PC:/media/jamestiotio/OS3/Users/jamestiotio/Documents/GitHub/cse/labs/lab2/BankersAlgorithm_Java$ make
javac Banker.java TestBankQ1.java TestBankQ2.java
gcc -o checkq1 checkerQ1.c
gcc -o checkq2 checkerQ2.c
jamestiotio@JRT-PC:/media/jamestiotio/OS3/Users/jamestiotio/Documents/GitHub/cse/labs/lab2/BankersAlgorithm_Java$ make testq1
./checkq1
For Q1: You have scored 1/1
jamestiotio@JRT-PC:/media/jamestiotio/OS3/Users/jamestiotio/Documents/GitHub/cse/labs/lab2/BankersAlgorithm_Java$ make testq2
./checkq2
For Q2: You have scored 1/1
```

Q3 – Analysis on Time Complexity of Banker's Algorithm

Let's say we have `n` customers and `m` resources in total.

- ✓ The `Banker()` constructor has O(1) time complexity since it mainly just initializes the class attributes.
- ✓ The `setMaximumDemand()` method has O(m) time complexity due to the for loop.
- ✓ The `printState()` method has O(n) time complexity due to the for loops.
- ✓ The `releaseResources()` method has O(m) time complexity due to the for loop.
- ✓ Since the `requestResources()` method contains a call to the `checkSafe()` method, we need to check whether the time complexity of the `checkSafe()` method dominates the remaining O(m) time complexity in the other parts of the `requestResources()` method due to the for loops.
- ✓ This is the main `checkSafe()` safety algorithm implemented in our code:

```
/**
 * Checks if the request will leave the bank in a safe state.
 * @param customerIndex  The customer's index (0-indexed).
 * @param request        An array of the requested count for each
resource.
 * @return true if the requested resources will leave the bank in a
 *         safe state, else false
 */
private synchronized boolean checkSafe(int customerIndex, int[] request)
{
    // Check if the state is safe

    // Copy the available, need and allocation arrays
    int[] tempAvailable = this.available.clone();
    int[][] tempNeed = this.need.clone();
    int[][] tempAllocation = this.allocation.clone();

    // Initialize finish vector (defaults to false)
    boolean[] finish = new boolean[this.numberOfCustomers];

    // Initialize a boolean flag
    boolean possible = true;

    for (int i = 0; i < this.numberOfResources; i++) {
        tempAvailable[i] -= request[i];
        tempNeed[customerIndex][i] -= request[i];
```

```
            tempAllocation[customerIndex][i] += request[i];
        }

        // Initialize work vector
        int[] work = tempAvailable.clone();

        while (possible) {
            possible = false;
            for (int i = 0; i < this.numberOfCustomers; i++) {
                boolean needDoesNotExceedWork = true;

                for (int j = 0; j < this.numberOfResources; j++) {
                    if (tempNeed[i][j] > work[j]) needDoesNotExceedWork =
false;
                }

                if (!finish[i] && needDoesNotExceedWork) {
                    possible = true;

                    for (int j = 0; j < this.numberOfResources; j++) {
                        work[j] += tempAllocation[i][j];
                    }

                    finish[i] = true;
                }
            }
        }

        // Undo the temporary changes that have been made to tempAllocation
and tempNeed
        for (int i = 0; i < this.numberOfResources; i++) {
            tempAllocation[customerIndex][i] -= request[i];
            tempNeed[customerIndex][i] += request[i];
        }

        // Check if all of the entries in the finish vector are true
        for (int i = 0; i < this.numberOfCustomers; i++) {
            if (!finish[i]) return false;
        }

        return true;
    }
```

The main contributor to the time complexity would be the double-nested for loops within the while loop. The outermost for loop will have O(n) time complexity since it needs to check for all `n` customers and the next inner for loop will have O(m) time complexity since it needs to check for all `m` resources. If the conditions specified in the if branching statement are satisfied (i.e., enough resources can be given to that specific customer), then the work vector is updated in O(m) time complexity as well since it needs to update for all m resources. Finally, the while loop needs to check for all `n`

customers in the worst case, which has O(n) time complexity. In total, we have $O(mn^2)$ time complexity, which definitely dominates the other O(m) and O(n) for loops within the `checkSafe()` method, as well as the O(m) time complexity in the other parts of the `requestResources()` method. Therefore, the **overall time complexity** of the Banker's algorithm is **$O(mn^2)$**.