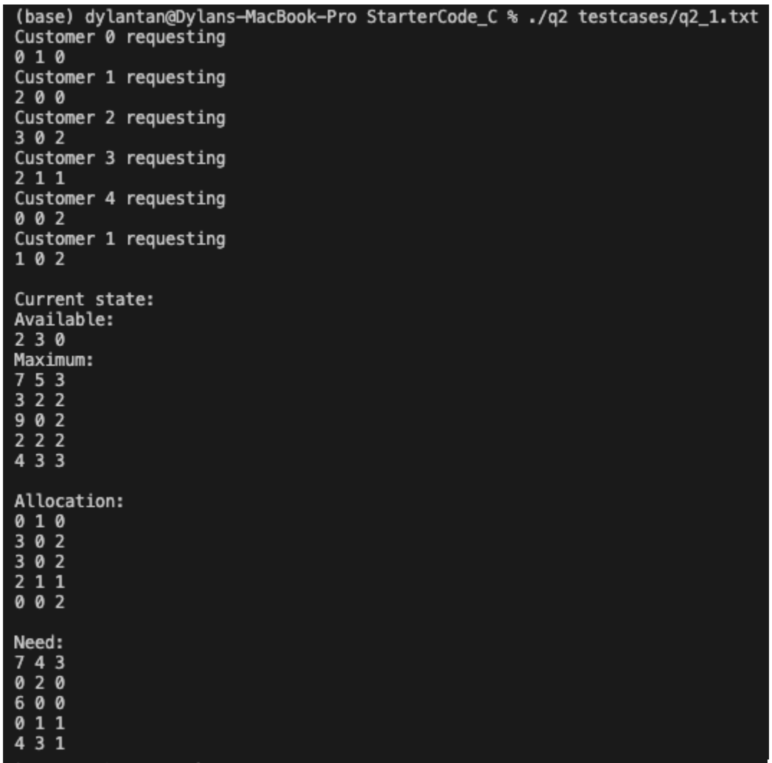
Name: Tan Ze Xin Dylan
Student ID: 1004385

50005 Lab 2 Report

Q2

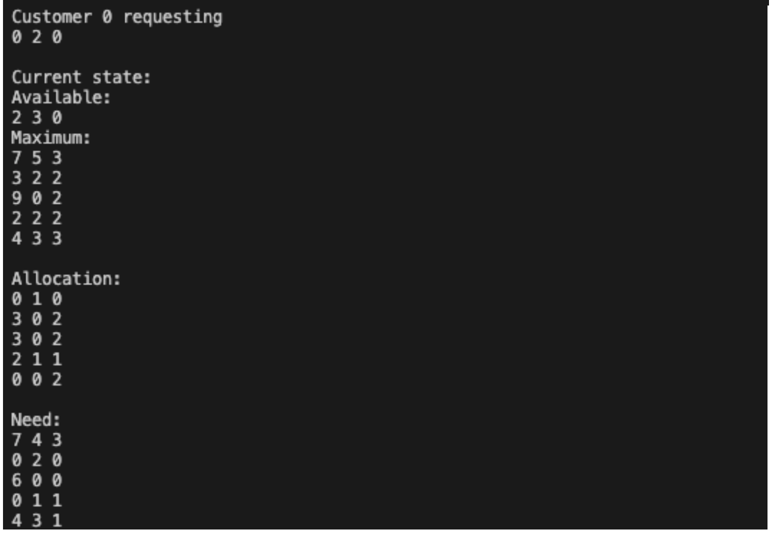


```
(base) dylantan@Dylans-MacBook-Pro StarterCode_C % ./q2 testcases/q2_1.txt
Customer 0 requesting
0 1 0
Customer 1 requesting
2 0 0
Customer 2 requesting
3 0 2
Customer 3 requesting
2 1 1
Customer 4 requesting
0 0 2
Customer 1 requesting
1 0 2

Current state:
Available:
2 3 0
Maximum:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Allocation:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

Need:
7 4 3
0 2 0
6 0 0
0 1 1
4 3 1
```

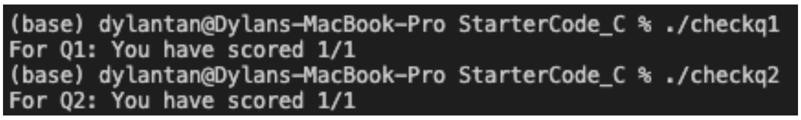


```
Customer 0 requesting
0 2 0

Current state:
Available:
2 3 0
Maximum:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

Allocation:
0 1 0
3 0 2
3 0 2
2 1 1
0 0 2

Need:
7 4 3
0 2 0
6 0 0
0 1 1
4 3 1
```



```
(base) dylantan@Dylans-MacBook-Pro StarterCode_C % ./checkq1
For Q1: You have scored 1/1
(base) dylantan@Dylans-MacBook-Pro StarterCode_C % ./checkq2
For Q2: You have scored 1/1
```

Q3

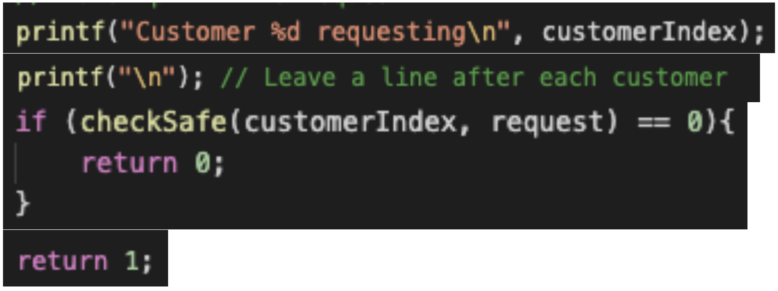
There are 2 parts to the banker's algorithm:

1) Resource request algorithm → O(m)
2) Safety algorithm → O(mn^2)

where m is the number of resources and n is the number of customers.

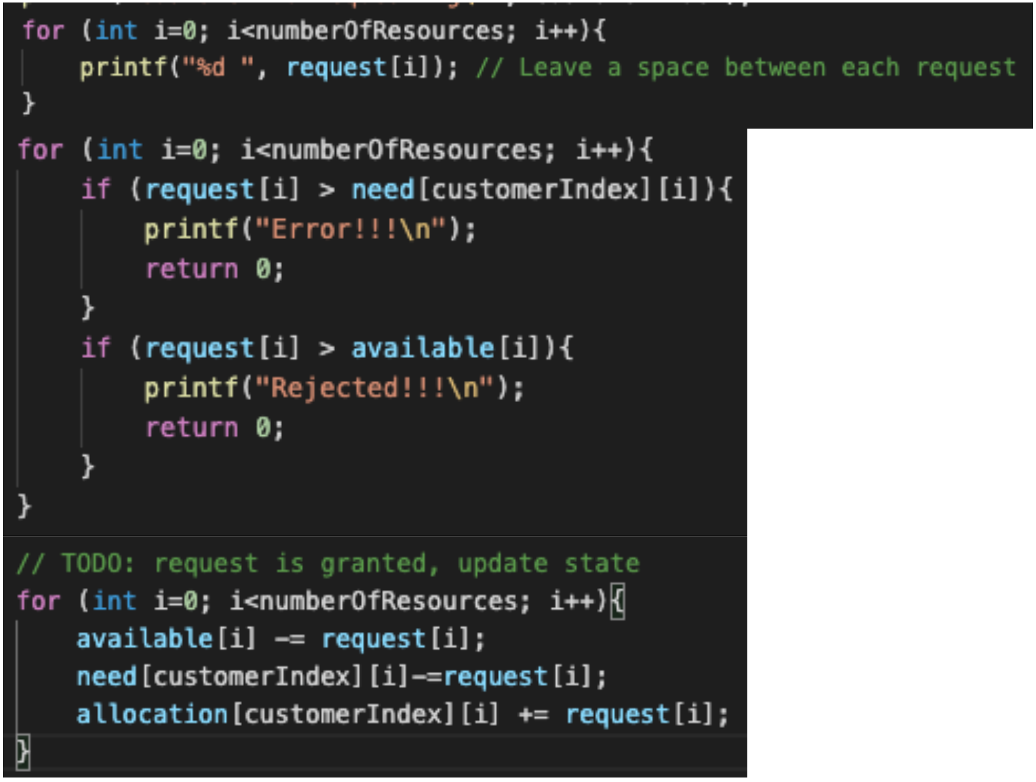
Resource request algorithm:

The print statements, return statement, and if-statement takes O(1) time if we exclude the safety algorithm.



```
printf("Customer %d requesting\n", customerIndex);
printf("\n"); // Leave a line after each customer
if (checkSafe(customerIndex, request) == 0){
    return 0;
}
return 1;
```

The 3 for-loops takes O(m) time each in the worst case.



```
for (int i=0; i<numberOfResources; i++){
    printf("%d ", request[i]); // Leave a space between each request
}
for (int i=0; i<numberOfResources; i++){
    if (request[i] > need[customerIndex][i]){
        printf("Error!!!\n");
        return 0;
    }
    if (request[i] > available[i]){
        printf("Rejected!!!\n");
        return 0;
    }
}
// TODO: request is granted, update state
for (int i=0; i<numberOfResources; i++){
    available[i] -= request[i];
    need[customerIndex][i]-=request[i];
    allocation[customerIndex][i] += request[i];
}
```

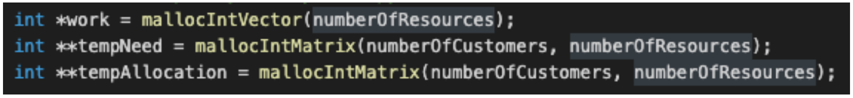
Therefore, time complexity is O(3m + 1), which is just O(m).

Safety Algorithm:

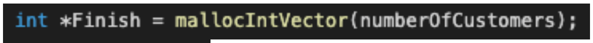
Allocation of space:

1) work →O(m)
2) tempNeed and tempAllocation → O(nm)
3) Finish → O(n).

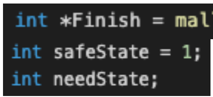
The declarations and return statement has O(1) time complexity.
Therefore, it takes O(m+nm+n+1), which is just O(mn).



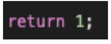
```
int *work = mallocIntVector(numberOfResources);
int **tempNeed = mallocIntMatrix(numberOfCustomers, numberOfResources);
int **tempAllocation = mallocIntMatrix(numberOfCustomers, numberOfResources);
```



```
int *Finish = mallocIntVector(numberOfCustomers);
```

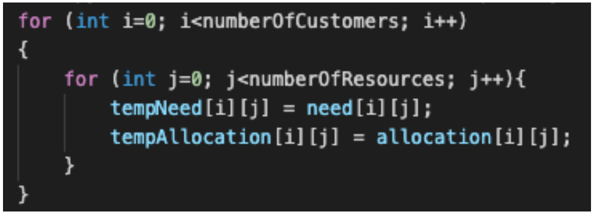


```
int safeState = 1;
int needState;
```



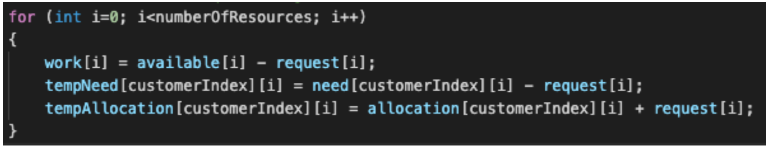
```
return 1;
```

Setting of temp arrays uses a nested for-loop and has a time complexity of O(mn).



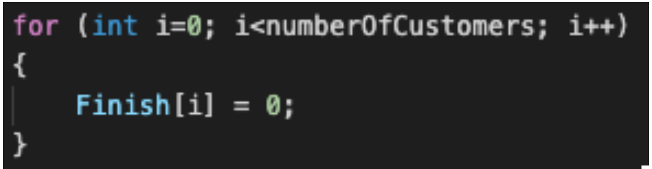
```
for (int i=0; i<numberOfCustomers; i++)
{
    for (int j=0; j<numberOfResources; j++){
        tempNeed[i][j] = need[i][j];
        tempAllocation[i][j] = allocation[i][j];
    }
}
```

Temporarily granting the request uses a for-loop with time complexity of O(m).

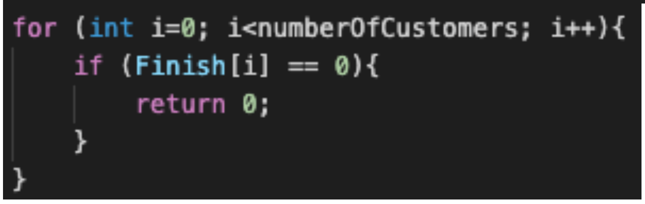


```
for (int i=0; i<numberOfResources; i++)
{
    work[i] = available[i] - request[i];
    tempNeed[customerIndex][i] = need[customerIndex][i] - request[i];
    tempAllocation[customerIndex][i] = allocation[customerIndex][i] + request[i];
}
```

Initialization and checking of all slots for the Finish array takes O(n) time each.



```
for (int i=0; i<numberOfCustomers; i++)
{
    Finish[i] = 0;
}
```



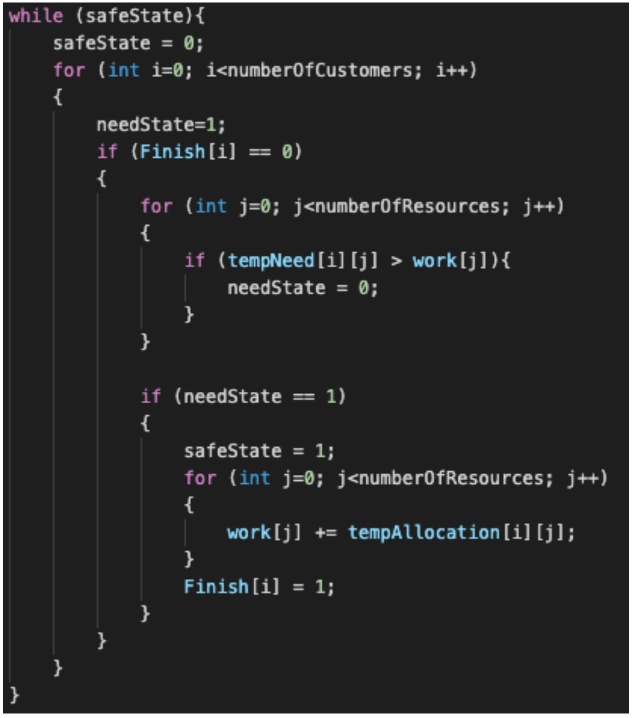
```
for (int i=0; i<numberOfCustomers; i++){
    if (Finish[i] == 0){
        return 0;
    }
}
```

The worst-case scenario for the while-loop would be when only 1 slot of the Finish array is set to 1 each loop. It will thus loop for n times just to set all the slots in the Finish array to 1, taking O(n) time.

Outermost for-loop takes O(n) to loop through all customers.

Innermost for-loops takes O(m) to loop through all resources.

Therefore, time complexity is O(n*n*m), which is O(mn^{2}).

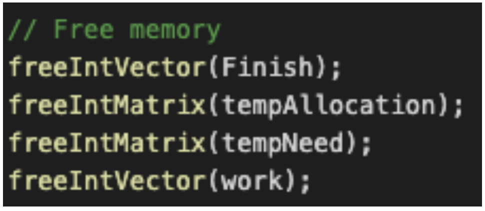


```
while (safeState){
    safeState = 0;
    for (int i=0; i<numberOfCustomers; i++)
    {
        needState=1;
        if (Finish[i] == 0)
        {
            for (int j=0; j<numberOfResources; j++)
            {
                if (tempNeed[i][j] > work[j]){
                    needState = 0;
                }
            }

            if (needState == 1)
            {
                safeState = 1;
                for (int j=0; j<numberOfResources; j++)
                {
                    work[j] += tempAllocation[i][j];
                }
                Finish[i] = 1;
            }
        }
    }
}
```

Freeing:

1) finish and work → O(1)
2) tempAllocation and tempNeed →O(n) time
3) Therefore, the whole freeing process takes O(n) time.



```
// Free memory
freeIntVector(Finish);
freeIntMatrix(tempAllocation);
freeIntMatrix(tempNeed);
freeIntVector(work);
```

Therefore, the combined time complexity for the safety algorithm is O(2mn + m + 3n + mn^{2}), which is O(mn^{2}).

In conclusion, the whole banker's algorithm takes O(m+mn^{2}), which is O(mn^{2}).