Algorithm Design And Complexity Analysis of Deadlock Avoidance in Banks

By Vedanshee Upadhyay

Abstract

The major goal for building this project was to identify problems from the real-world and design an algorithm in order to solve them. The problem identified was; in banks testing for safety is required before passing any loan by simulating the allocation of predetermined maximum possible amounts of all resources, and then making a safe-state check to test for possible deadlock conditions for all other pending cases. The classical banker's algorithm of Dijkstra for resource allocation guarantees freedom from deadlock by deny- ing requests for resources that lead to unsafe allocation.

The Banker algorithm is commonly used in the Operating System (OS), but some improvement will have to be made on the algorithm since its complexity goes in the order of n square which would be seen in the report in chapter 6 that how its complexity has been reduced.

Algorithm design and its successful implementation has been a major look out for this project. There are three important matrices in this algorithm, namely the maximum demand matrix Max[M,N], the assigned matrix Allocation[M,N], and the demand matrix Need[M,N]. Their relationship is the demand matrix Need[M,N]= Max[M,N]- Allocation[M,N]. This algorithm can be applied in various places wherever there is a need of optimization of space/time/funds for avoiding the situation of insufficient fund/space/time availability. The banker algorithm is a dynamic strategy to avoid deadlock. The advantage of the banker algorithm is that it can effectively and reasonably arrange the existing resources of the system. Compared with other algorithms, the restrictions are less and the utilization of resources is improved.

1. Problem Identification

1.1 Problem Statement

To understand how an algorithm could have solved the debt crisis of a bank you have to go back to the basics of banking and figure out answers to these:

How do banks work?
How do banks decide the loan amount?
What does Banker's algorithm do?

A scenario where the moneylender is left with not enough money to pay the borrower and none of the jobs are complete due to insufficient funds, leaving incomplete tasks and cash stuck as bad debt, a deadlock happens. And to prevent deadlock Banker's Algorithm was made.
The goal of the Banker's algorithm is to handle all requests without entering into the unsafe state, also called a deadlock. It's called the Banker's algorithm because it could be used in the banking system so that banks never run out of resources and always stay in a safe state.

1.2 Constraints

I. The algorithm needs to know <u>how much of each resource a process could possibly request</u>. In most systems, this information is unavailable, making the Banker's algorithm useless.

II. Besides, it is unrealistic to assume that the <u>number of processes is static</u>. In most systems the number of processes varies dynamically. Moreover, the requirement that a <u>process will eventually release all its resources (when the process terminates) is sufficient for the correctness of the algorithm</u>, however it is not sufficient for a practical system.

III. <u>Waiting for hours (or even days) for resources to be released</u> is usually not acceptable.

2. Software Requirement Specification

2.1 Hardware Requirements

System	Intel Premium 3 or higher
Primary Memory	8 GB or more
Storage	256 GB or more

2.2 Software Requirements

Language	C++
Operating System	Windows 10
IDE	Visual Studio Code / CodeBlocks

3. Design

3.1 Data Structure for Algorithm

For the algorithm to work, it should know three things:

1. How much of each resource each person could maximum request [MAX]
2. How much of each resource each person currently holds [Allocated]
3. How much of each resource is available in the system for each person [Available]

So we need MAX and REQUEST.
If REQUEST is given MAX = ALLOCATED + REQUEST
NEED = MAX – ALLOCATED
A resource can be allocated only for a condition.
REQUEST<= AVAILABLE or else it waits until resources are available.

The data structure of the algorithm is described as follows:
Let ‘n’ be the number of processes in the system and ‘m’ be the number of resource types.

1. Available – It is a 1D array of size’m’. Available [j] = k means there are k occurrences of resource type Rj.
2. Maximum – It is a 2D array of size ‘m*n’ which represents the maximum demand of a section. Max[i,j] = k means that a process i can maximum demand ‘k’ amount of resources.
3. Allocated – It is a 2D array of size ‘m*n’ which represents the number of resources allocated to each process. Allocation [i,j] =k means that a process is allocated ‘k’ amount of resources.
4. Need – 2D array of size ‘m*n’. Need [i,j] = k means a maximum resource that could be allocated.
5. Need [i,j] = Max [i,j] – Allocation[i,j]

3.2 Algorithm Design

Algorithm 1: Structure of whole algorithm/code

Step 1: START
Step 2: Make a class
    class ResourceManager
Step 2.1: Declare variables and functions
Step 3: Function1: bankers
Step 4: Function2: input
Step 5: Function3: method
Step 6: Function4: search
Step 7: Function5: display
Step 8: Calling Functions in Main Function
Step 9: STOP

Algorithm 2: For Function1: bankers

```
Step 1: Start
Step 2: Initializing variable k=0 (k: current resource), MAX = 20
Step 3: LOOP: for(int i=0;i<MAX;i++)
  Step 3.1: LOOP: for(int j=0;j<MAX;j++)
                 allocation=0
                 maximum_need=0
                 need=0
          END LOOP
          available=0
          result=0
          finish=0
       END LOOP
Step 4: Stop
```

Algorithm 3: For Function2: input

Step 1: Start
Step 2: Initializing variables i,j
Step 3: INPUT: No. of people

```
        INPUT: No. of resources
        INPUT: Allocated resources
Step 4: LOOP: for(i=0;i<nop;i++)
                 DISPLAY: Person
                 for(j=0;j<nor;j++)
                    DISPLAY: Resource
Step 5: INPUT: Maximum resources
Step 6: LOOP: for(i=0;i<nop;i++)
                 DISPLAY: Person
                 for(j=0;j<nor;j++)
                    DISPLAY: Resource
                    need=maximum_need - allocation
Step 7: INPUT: Currently available resources
Step 8: LOOP: for(j=0;j<nor;j++)
                 DISPLAY: Resource
                 for(i=0;i<nop;i++)
                    finish[i]=0;
Step 9: Stop
```

<u>Algorithm 4: Function3: method -> To make method for safe sequence</u>

```
Step 1: Start
Step 2: Initializing variables i=0,j,flag
Step 3: MAIN LOOP:
  while(1)

    if(finish[i]==0)

      pnum =search(i);
      if(pnum!=-1)

        result[k++] = i
        finish[i] = 1
        for(j=0;j<nor;j++)
          available = available + allocated
      END IF
    END IF
    i++;
```

```
      if(i==nop)

         flag=0
         for(j=0;j<nor;j++)
            if(avail[j]!=work[j])

         flag=1
         for(j=0;j<nor;j++)
            work[j]=avail[j];

         if(flag==0)
            break
         else
            i=0
       END IF
  END WHILE
Step 4: Stop
```

<u>Algorithm 5: For Function4: search</u>

```
Step 1: Start
Step 2: Initializing variable j
Step 3: LOOP: for(j=0;j<nor;j++)
  Step 3.1: LOOP: if(need>available)
                        RETURN -1
RETURN 0
Step 4: Stop
```

<u>Algorithm 6: For Function5: display</u>

```
Step 1: Start
Step 2: Initializing variables i, j
Step 3: LOOP: for(j=0;j<nop;j++)
  Step 3.1: LOOP: for(j=0;j<nor;j++)
              DISPLAY: allocation
              DISPLAY: maximum_need
```

DISPLAY: need

Step 4: DISPLAY: Safe Sequence

Step 4.1: LOOP: for(i=0;i<k;i++)

temp = result[i]+1

Step 5: DISPLAY: Unsafe Sequence

Step 5.1: flg=0

Step 5.2: LOOP: for(i=0;i<nop;i++)

Step 5.2.1: LOOP: if(finish[i]==0)

flg=1

Step 6: LOOP: if(flg==1)

DISPLAY: Safe state

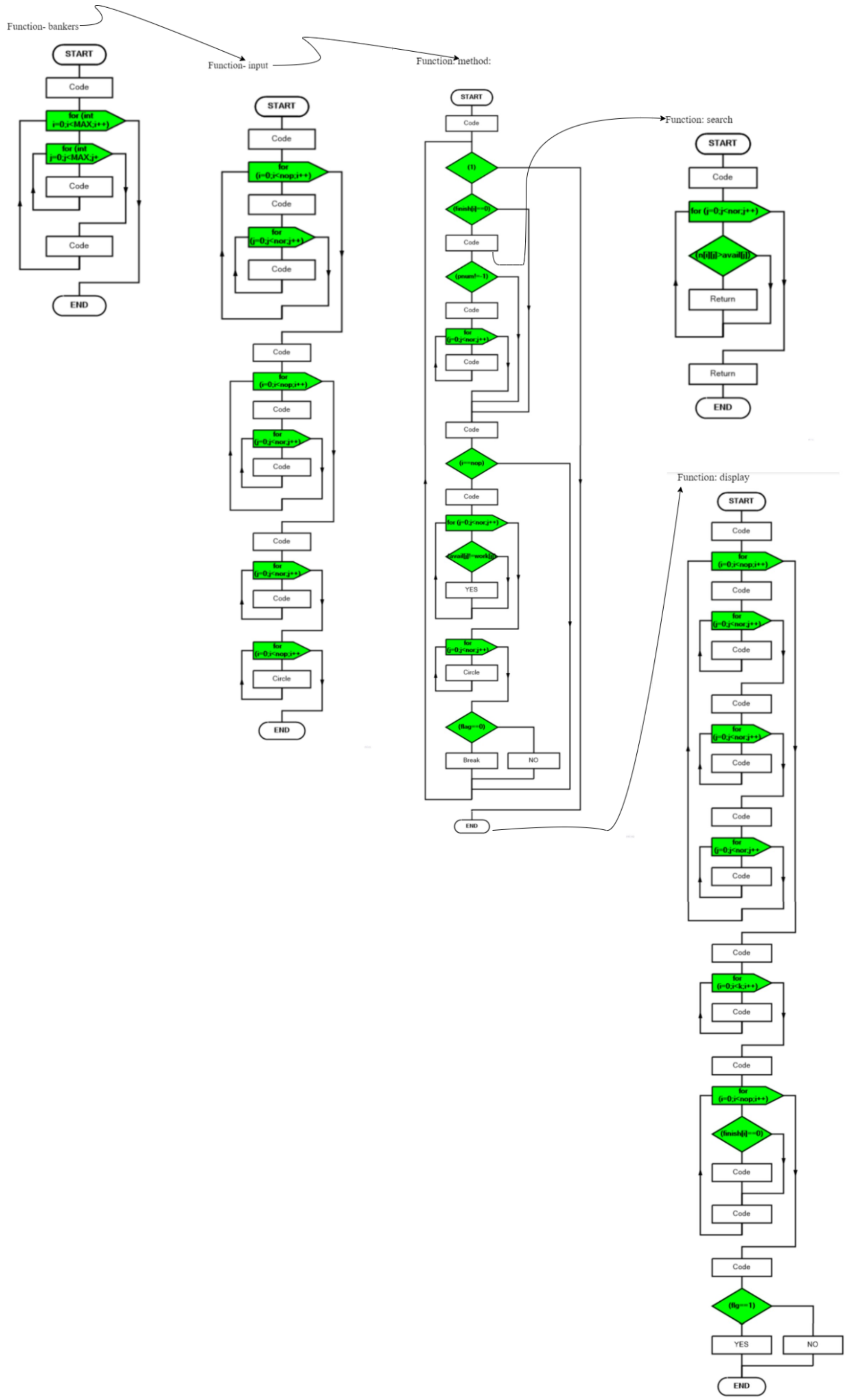
else

DISPLAY: Unsafe state

Step 7: Stop

NOTE: The algorithm has been optimized in ALGORITHM 4 as banker's algorithm has 2 algorithms but here the logic has been combined and applied in a single algorithm for reducing the complexity of the algorithm.

3.3 Flow Chart



4. Implementation

// A C++ program is made so that banks never run out of resources and always stay in a safe state.

#include <iostream>

#define MAX 20

using namespace std;

class bankers {

private:

// MAX: No. of items in array, al: allocation, m: maximum need, n: remaining need, avail: available resources

int al[MAX][MAX],m[MAX][MAX],n[MAX][MAX],avail[MAX];

// nop: No. of people, nor: No. of resources, k: current resources, pnum: process number

int nop,nor,k,result[MAX],pnum,work[MAX],finish[MAX];

// Declaring all the functions in public

```
    public:

        bankers();

        void input();

        void method();

        int search(int);

        void display();

};
```

// Main function where the logic of banker's algorithm being applied in an <u>optimized way</u>

```
void bankers::method()

{

    int i=0,j,flag;

    while(1)

    {

        if(finish[i]==0)
```

```
{
    pnum =search(i);
    if(pnum!=-1)
    {
        result[k++]=i;
        finish[i]=1;
        for(j=0;j<nor;j++)
        {
          // Condition for process of the person terminates leaving it's resource
          avail[j]=avail[j]+al[i][j];
        }
    }
}
i++;
if(i==nop)
```

```
    {
      flag=0;
      for(j=0;j<nor;j++)
          if(avail[j]!=work[j])



      flag=1;
      for(j=0;j<nor;j++)
          work[j]=avail[j];
```

// Condition to check the status of the process

```
      if(flag==0)
          break;
      else
          i=0;
    }
}
```

```
}

// This function gets called in method function to check a condition

// where a person needs to be found for whom need is greater than availability

int bankers::search(int i){

    for(int j=0;j<nor;j++)

        if(n[i][j]>avail[j])

            return -1;

return 0;

}
```

5. Testing

I. For Safe State Process

There are enough resources to grant the request

Expected Output: The request should be granted

Expected Output: Deadlock will not occur

INPUT:

Available system resources

	A	B	C	D
Free	3	1	0	2

Processes (currently allocated resources):

	A	B	C	D
P1	1	2	2	1
P2	1	0	3	3
P3	1	2	2	0

Processes (maximum resources):

	A	B	C	D
P1	3	3	2	2
P2	1	2	3	4
P3	1	3	5	0

OUTPUT:

```
DEADLOCK PREVENTION IN  A BANK
______________________________

Enter the number of people:3
Enter the number of resources:4


Enter the allocated resources for each person:

Person 1
Resource 1:1

Resource 2:2

Resource 3:2

Resource 4:1

Person 2
Resource 1:1

Resource 2:0

Resource 3:3

Resource 4:3

Person 3
Resource 1:1

Resource 2:2

Resource 3:2

Resource 4:0
```

```
Enter the maximum resources that are needed for each person:

Person 1
Resource 1:3

Resource 2:3

Resource 3:2

Resource 4:2

Person 2
Resource 1:1

Resource 2:2

Resource 3:3

Resource 4:4

Person 3
Resource 1:1

Resource 2:3

Resource 3:5

Resource 4:0


Enter the currently available resources in the system:

Resource 1:3


Resource 2:1


Resource 3:0


Resource 4:2
```

```
OUTPUT:
PERSON       ALLOTED        MAXIMUM        NEED
P1           1  2  2  1             3  3  2  2             2  1  0  1
P2           1  0  3  3             1  2  3  4             0  2  0  1
P3           1  2  2  0             1  3  5  0             0  1  3  0


Safe Sequence:
P1 P2 P3


Unsafe Sequence:
P1 P2 P3


RESULT:
The system is in safe state and deadlock will not occur!!


Process returned 0 (0x0)   execution time : 105.993 s
Press any key to continue.
```

TEST STATUS: Passed

II. For Unsafe State Process

There is not enough of the resources available to grant the request

Expected Output: The request should not be granted

Expected Output: Deadlock will occur

INPUT:

Available system resources:

A B C D

Free 3 0 1 2

Processes (currently allocated resources):

A B C D

P1 1 2 5 1

P2 1 1 3 3

P3 1 2 1 0

Processes (maximum resources):

A B C D

P1 3 3 2 2

P2 1 2 3 4

P3 1 3 5 0

OUTPUT:

```
DEADLOCK PREVENTION IN  A BANK
______________________________

Enter the number of people:3
Enter the number of resources:4


Enter the allocated resources for each person:

Person 1
Resource 1:1

Resource 2:2

Resource 3:5

Resource 4:1

Person 2
Resource 1:1

Resource 2:1

Resource 3:3

Resource 4:3

Person 3
Resource 1:1

Resource 2:2

Resource 3:1

Resource 4:0
```

```
Enter the maximum resources that are needed for each person:

Person 1
Resource 1:3

Resource 2:3

Resource 3:2

Resource 4:2

Person 2
Resource 1:1

Resource 2:2

Resource 3:3

Resource 4:4

Person 3
Resource 1:1

Resource 2:3

Resource 3:5

Resource 4:0


Enter the currently available resources in the system:

Resource 1:3


Resource 2:0


Resource 3:1


Resource 4:2
```

```
OUTPUT:
PERSON       ALLOTED         MAXIMUM         NEED
P1           1  2  5  1              3  3  2  2              2  1  -3  1
P2           1  1  3  3              1  2  3  4              0  1  0  1
P3           1  2  1  0              1  3  5  0              0  1  4  0


Safe Sequence:



Unsafe Sequence:
P1 P2 P3


RESULT:
The system is not in safe state and deadlock may occur!!


Process returned 0 (0x0)   execution time : 66.720 s
Press any key to continue.
```

TEST STATUS: **Passed**

6. Results

Time Complexity Analysis for the Entire Algorithm

Algorithm 1: For Function1: bankers

Step 1: Start
Step 2: Initializing variable k=0 (k: current resource), MAX = 20 ------------------------------1
Step 3: LOOP: for(int i=0;i<MAX;i++) ---n+1
Step 3.1: LOOP: for(int j=0;j<MAX;j++) ---n (n+1)
allocation=0 --n (n)
maximum_need=0 --n (n)
need=0 --n (n)
END LOOP
available=0 --n
result=0 ---n
finish=0 ---n
END LOOP
Step 4: Stop

Time Complexity T(n) : 1+n+1+n(n+1)+n^2+n^2+n^2+3n = 4n^2 + 4n + 2 : O(n^2)

Algorithm 2: For Function2: input

Step 1: Start
Step 2: Initializing variables i,j
---1
Step 3: INPUT: No. of people
---1
INPUT: No. of resources
---1
INPUT: Allocated resources
---1
Step 4: LOOP: for(i=0;i<nop;i++)
---n+1

```
                              DISPLAY: Person
-----------------------------------------------------------------------------------n
                              for(j=0;j<nor;j++)
-----------------------------------------------------------------------------------n (m+1)
                                   DISPLAY: Resource
-----------------------------------------------------------------------------------n (m)
Step 5: INPUT: Maximum resources
-----------------------------------------------------------------------------------1
Step 6: LOOP: for(i=0;i<nop;i++)
----------------------------------------------------------------------------------------n+1
                              DISPLAY: Person
-----------------------------------------------------------------------------------n
                              for(j=0;j<nor;j++)
-----------------------------------------------------------------------------------n (m+1)
                                   DISPLAY: Resource
-----------------------------------------------------------------------------------n (m)
                                   need=maximum_need - allocation
----------------------------------------------------------------- n (m)
Step 7: INPUT: Currently available resources
-----------------------------------------------------------------------------------1
Step 8: LOOP: for(j=0;j<nor;j++)
---------------------------------------------------------------------------------------------m+1
                              DISPLAY: Resource
-----------------------------------------------------------------------------------m
                              for(i=0;i<nop;i++)
-----------------------------------------------------------------------------------m (n+1)
                                   finish[i]=0
-----------------------------------------------------------------------------------m (n)
Step 9: Stop
-----------------------------------------------------------------------------------------------
```

Time Complexity T(n): 7nm+6n+3m+9 : O(nm)

Algorithm 3: Function3: method -> To make method for safe sequence

```
Step 1: Start
Step 2: Initializing variables i=0,j,flag
-----------------------------------------------------------------------------------1
Step 3: MAIN LOOP:
   while(1) ------------------------------------------------------------------------------------1

      if(finish[i]==0) ------------------------------------------------------------------------------------n

         pnum =search(i) ------------------------------------------------------------------------------------n
         if(pnum!=-1) ------------------------------------------------------------------------------------n (n)

            result[k++] = i ------------------------------------------------------------------------------------n (n)
            Finish[i] = 1 ------------------------------------------------------------------------------------n (n)
            for(j=0;j<nor;j++)
-----------------------------------------------------------------------------------n (m+1)
               available = available + allocated
-------------------------------------------------------------m
      END IF
      i++;
      if(i==nop) ------------------------------------------------------------------------------------n
         flag=0 ------------------------------------------------------------------------------------n
         for(j=0;j<nor;j++)
-----------------------------------------------------------------------------------n (m+1)
            if(avail[j]!=work[j])
-----------------------------------------------------------------------------------n (m)
         flag=1 ------------------------------------------------------------------------------------n
         for(j=0;j<nor;j++)
-----------------------------------------------------------------------------------n (m+1)
            work[j]=avail[j]
-----------------------------------------------------------------------------------n (m)

         if(flag==0) ------------------------------------------------------------------------------------n (n)
            break ------------------------------------------------------------------------------------n (n)
         else ----------------------------------------------------------------------------------n(n)
            i=0 ------------------------------------------------------------------------------------n (n)
Step 4: Stop
----------------------------------------------------------------------------------------------------
```

Time Complexity T(n): 7n^2+5nm+8n+m+2 : O(n^2)

Algorithm 4: For Function4: search

Step 1: Start

Step 2: Initializing variable j --1

Step 3: LOOP: for(j=0;j<nor;j++) --m+1

Step 3.1: LOOP: if(need>available) --m (n)

RETURN -1 --m (n)

RETURN 0

Step 4: Stop

Time Complexity T(n): 2mn+m+2 : O(mn)

Algorithm 5: For Function5: display

Step 1: Start

Step 2: Initializing variables i, j --1

Step 3: LOOP: for(j=0;j<nop;j++) --n+1

Step 3.1: LOOP: for(j=0;j<nor;j++) -- n (m+1)

DISPLAY: allocation ---n (m)

DISPLAY: maximum_need --n (m)

DISPLAY: need ---n (m)

Step 4: DISPLAY: Safe Sequence ---1

Step 4.1: LOOP: for(i=0;i<k;i++) ---m+1

temp = result[i]+1 --m

Step 5: DISPLAY: Unsafe Sequence ---1

Step 5.1: flg=0 ---1

Step 5.2: LOOP: for(i=0;i<nop;i++) --n+1

Step 5.2.1: LOOP: if(finish[i]==0) --n(n)

flg=1 --n(n)

Step 6: LOOP: if(flg==1) ---n

DISPLAY: Safe state ---n

else --n

DISPLAY: Unsafe state ---n

Step 7: Stop

Time Complexity T(n): 2n^2+4nm+7n+2m+7 : O(n^2)

Algorithm 6: Structure of whole algorithm/code

Step 1: START
Step 2: Make a class
class ResourceManager
Step 2.1: Declare variables and functions
Step 3: Function1: bankers
--O(n^2)
Step 4: Function2: input
--O(mn)
Step 5: Function3: method
--O(n^2)
Step 6: Function4: search
--O(mn)
Step 7: Function5: display
--O(n^2)
Step 8: Calling Functions in Main Function
Step 9: STOP

Final Time Complexity T(n): O(mn)

Note: If both array have the same size, the time complexity is O(N^2) If both array have a different size, the time complexity is O(N.M) (as in N times M, where N and M are the array sizes)

The original Banker's Algorithm has a time complexity of O(n^2 m) which is optimized and made into O(nm).

7. Conclusion

Deadlock avoidance has been successfully done by the algorithm that checks for the safety by simulating the allocation of predetermined maximum possible resources and makes the system into s-state by checking the possible deadlock conditions for all other pending processes.

This report presents one improvement to the classical Banker's algorithm for deadlock avoidance in concur-rent systems by reducing its time complexity. For its future scope; an approach for Dynamic Banker's algorithm can be proposed which allows the number of resources to be changed at runtime that prevents the system from falling into unsafe state. It will also give details about all the resources and processes that require resources and in what quantity. This will also allocate the resources automatically to the stopped processes for the execution and will always give the appropriate safe sequence for the given processes. Further it can also take into account the interest rate while any person returns the resource and then calculate the required amount for the rest of the people.

8. Appendix

Code-Link:
https://drive.google.com/file/d/1PVZHzxfYxGCZEH20dEMOlMyBdH5t0mpH/view?usp=sharing