NAME: **Meet Raut**

DIV: **S2-1**

ROLL.NO: **2201084**

Experiment 7:

- **AIM: To study and implement the concept of deadlock avoidance through Banker's Algorithm.**

- **THEORY:**

a) BANKER'S ALGORITHM:

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Why Banker's Algorithm is Named So?

The banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that the bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.
It also helps the OS to successfully share the resources between all the processes. It is called the banker's algorithm because bankers need a similar algorithm- they admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments. The banker's algorithm uses the notation of a safe allocation state to ensure that granting a resource request cannot lead to a deadlock either immediately or in the future.
In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in a safe state always.

The following **Data structures** are used to implement the Banker's Algorithm:

Let **'n'** be the number of processes in the system and **'m'** be the number of resource types.

Available

- It is a 1-d array of size **‘m’** indicating the number of available resources of each type.
- Available[j] = k means there are **‘k’** instances of resource type **Rj**

Max

- It is a 2-d array of size ‘**n*m’** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process **Pi** may request at most **‘k’** instances of resource type **Rj.**

Allocation

- It is a 2-d array of size **‘n*m’** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process **Pi** is currently allocated **‘k’** instances of resource type **Rj**

Need

- It is a 2-d array of size **‘n*m’** that indicates the remaining resource need of each process.
- Need [i,   j] = k means process **Pi** currently needs **‘k’** instances of resource type **Rj**
- Need [i,   j] = Max [i,   j] – Allocation [i,   j]

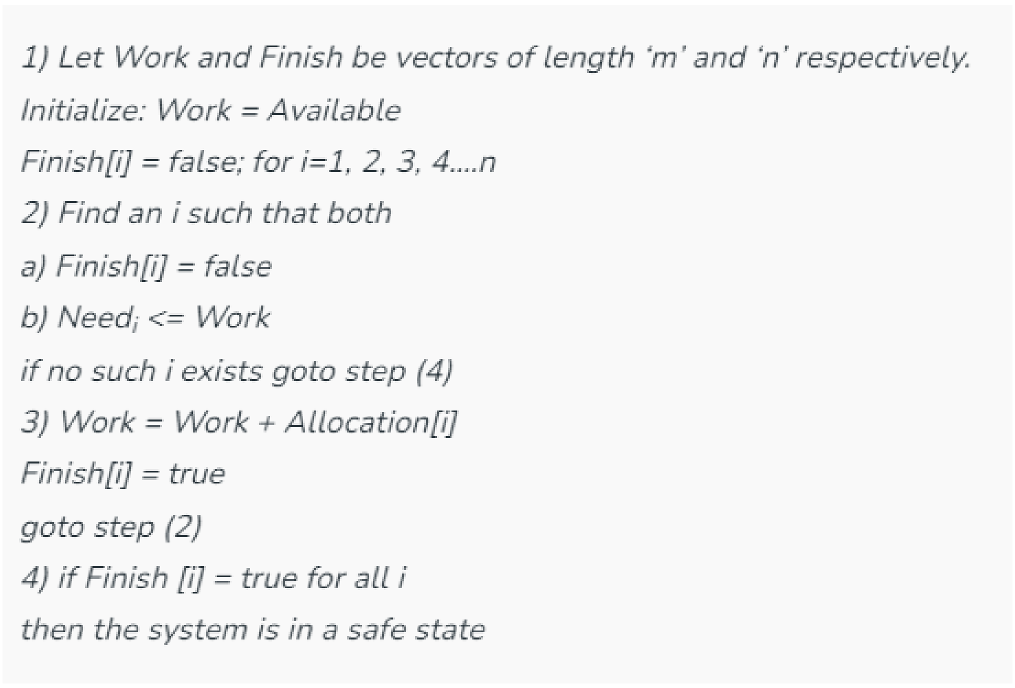
Allocation specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.
Banker’s algorithm consists of a Safety algorithm and a Resource request algorithm.

Banker’s Algorithm:-

```
1. Active:= Running U Blocked;
   for k=1…r
   New_ request[k]:= Requested_ resources[requesting_ process, k];
2. Simulated_ allocation:= Allocated_ resources;
   for k=1…..r //Compute projected allocation state
   Simulated_ allocation [requesting _process, k]:= Simulated_ allocation
[requesting _process, k] + New_ request[k];
3. feasible:= true;
   for k=1….r // Check whether projected allocation state is feasible
```

```
    if Total_ resources[k]< Simulated_ total_ alloc [k] then feasible:= false;
4. if feasible= true
    then // Check whether projected allocation state is a safe allocation state
    while set Active contains a process P1 such that
    For all k, Total _resources[k] – Simulated_ total_ alloc[k]>= Max_ need [l ,k]-Simulated_ allocation[l, k]
    Delete Pl from Active;
    for k=1…..r
        Simulated_ total_ alloc[k]:= Simulated_ total_ alloc[k]- Simulated_ allocation[l, k];
5. If set Active is empty
    then // Projected allocation state is a safe allocation state
    for k=1….r // Delete the request from pending requests
        Requested_ resources[requesting_ process, k]:=0;
    for k=1….r // Grant the request
        Allocated_ resources[requesting_ process, k]:= Allocated_ resources[requesting_ process, k] + New_ request[k];
    Total_ alloc[k]:= Total_ alloc[k] + New_ request[k];
```

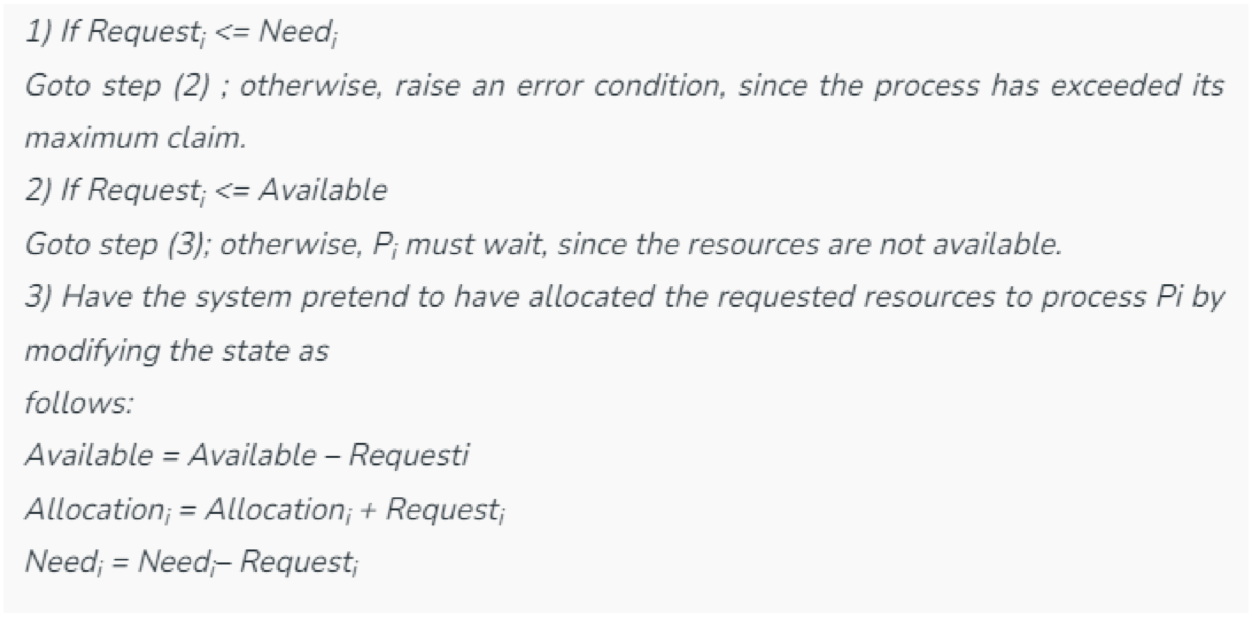
Safety Algorithm: The algorithm for finding out whether or not a system is in a safe state can be described as follows:



```
1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4….n
2) Find an i such that both
a) Finish[i] = false
b) $Need_i$ <= Work
if no such i exists goto step (4)
3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)
4) if Finish [i] = true for all i
then the system is in a safe state
```

Resource-Request Algorithm:

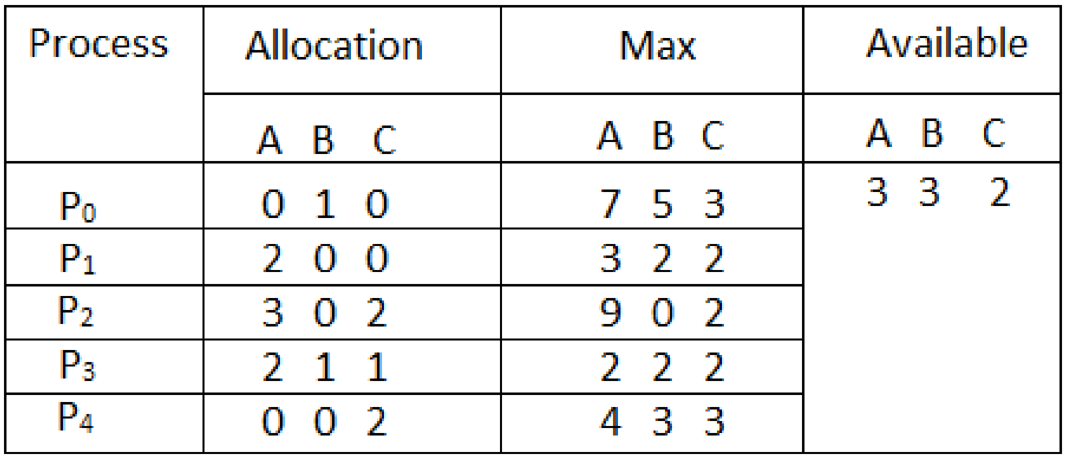
Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:



```
1) If $Request_i$ <= $Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its
maximum claim.
2) If $Request_i$ <= Available
Goto step (3); otherwise, $P_i$ must wait, since the resources are not available.
3) Have the system pretend to have allocated the requested resources to process Pi by
modifying the state as
follows:
Available = Available – Requesti
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$– $Request_i$
```

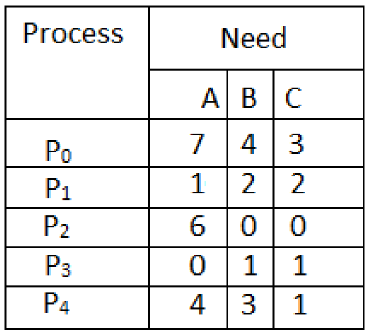
Example:

Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:



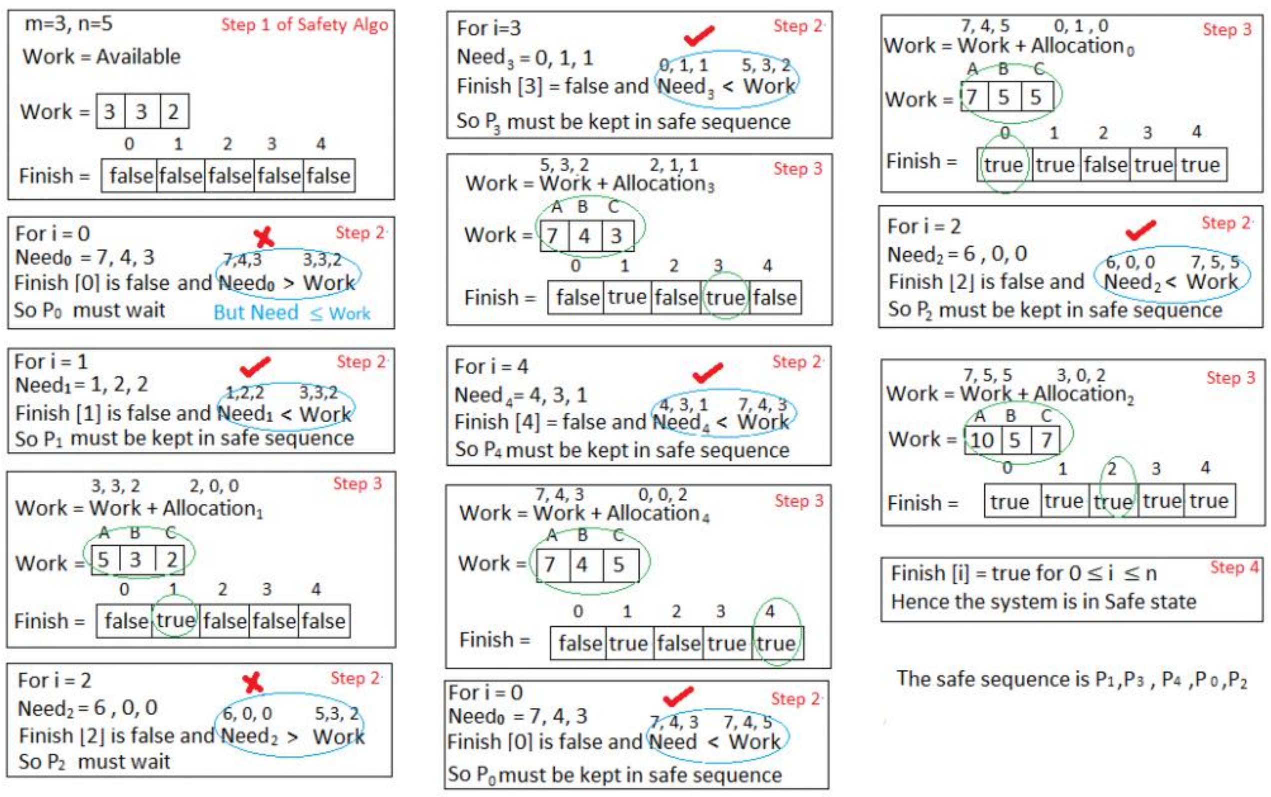
Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Q.1: What will be the content of the Need matrix?
Need [i, j] = Max [i, j] – Allocation [i, j]
So, the content of Need Matrix is:



Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Q.2: Is the system in a safe state? If Yes, then what is the safe sequence?
Applying the Safety algorithm on the given system,



m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 ✗ — Step 2

$Need_0 = 7, 4, 3$

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$

So P_0 must wait   But Need ≤ Work

For i = 1 ✓ — Step 2

$Need_1 = 1, 2, 2$

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$

So P_1 must be kept in safe sequence

Step 3

$Work = \overset{3,3,2}{Work} + \overset{2,0,0}{Allocation_1}$

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 ✗ — Step 2

$Need_2 = 6, 0, 0$

Finish [2] is false and $\overset{6,0,0}{Need_2} > \overset{5,3,2}{Work}$

So P_2 must wait

For i=3 ✓ — Step 2

$Need_3 = 0, 1, 1$

Finish [3] = false and $\overset{0,1,1}{Need_3} < \overset{5,3,2}{Work}$

So P_3 must be kept in safe sequence

Step 3

$Work = \overset{5,3,2}{Work} + \overset{2,1,1}{Allocation_3}$

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 ✓ — Step 2

$Need_4 = 4, 3, 1$

Finish [4] = false and $\overset{4,3,1}{Need_4} < \overset{7,4,3}{Work}$

So P_4 must be kept in safe sequence

Step 3

$Work = \overset{7,4,3}{Work} + \overset{0,0,2}{Allocation_4}$

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 ✓ — Step 2

$Need_0 = 7, 4, 3$

Finish [0] is false and $\overset{7,4,3}{Need} < \overset{7,4,5}{Work}$

So P_0 must be kept in safe sequence

Step 3

$Work = \overset{7,4,5}{Work} + \overset{0,1,0}{Allocation_0}$

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 ✓ — Step 2

$Need_2 = 6, 0, 0$

Finish [2] is false and $\overset{6,0,0}{Need_2} < \overset{7,5,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

$Work = \overset{7,5,5}{Work} + \overset{3,0,2}{Allocation_2}$

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for $0 \le i \le n$

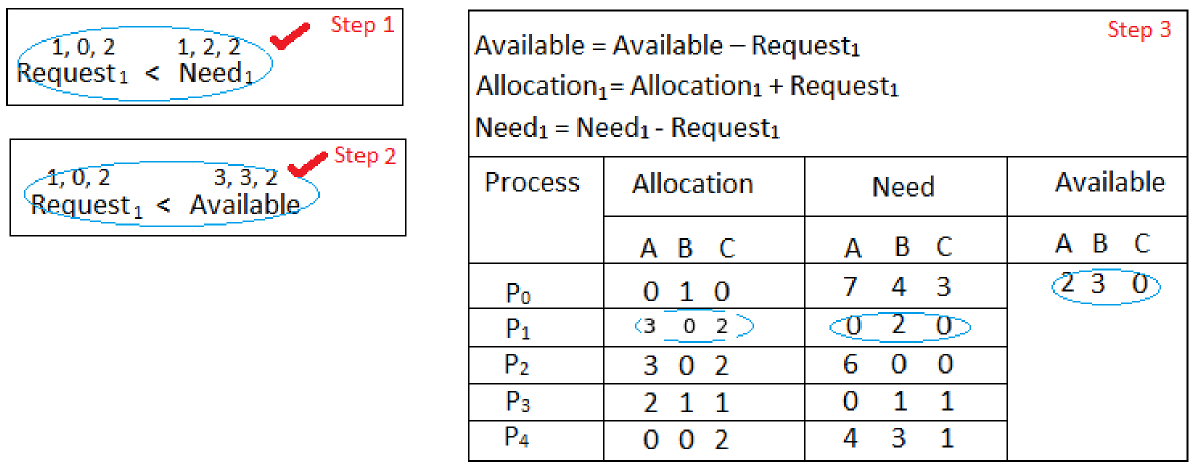
Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Q.3: What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?

$Request_1 = \overset{A\ B\ C}{1, 0, 2}$

To decide whether the request is granted we use Resource Request algorithm



Step 1 ✓

$\overset{1,0,2}{Request_1} < \overset{1,2,2}{Need_1}$

Step 2 ✓

$\overset{1,0,2}{Request_1} < \overset{3,3,2}{Available}$

Step 3

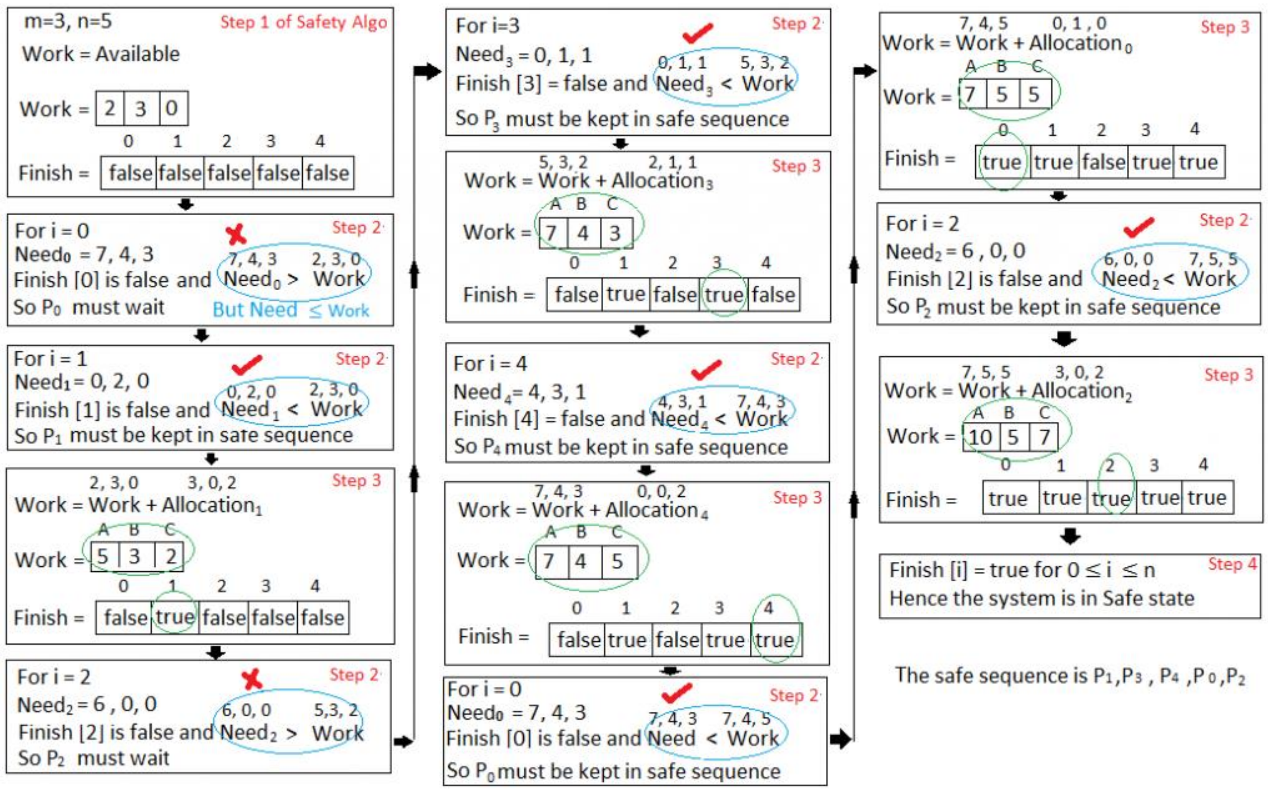
$Available = Available – Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.



Step 1 of Safety Algo

m=3, n=5

Work = Available

A	B	C
2	3	0

Finish =

0	1	2	3	4
false	false	false	false	false

Step 2

For i = 0

$Need_0 = 7, 4, 3$

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{2,3,0}{Work}$

So P_0 must wait

But Need $\leq$ Work

Step 2

For i = 1

$Need_1 = 0, 2, 0$

Finish [1] is false and $\overset{0,2,0}{Need_1} < \overset{2,3,0}{Work}$

So P_1 must be kept in safe sequence

Step 3

$\overset{2,3,0}{Work} = Work + \overset{3,0,2}{Allocation_1}$

Work =

A	B	C
5	3	2

Finish =

0	1	2	3	4
false	true	false	false	false

Step 2

For i = 2

$Need_2 = 6, 0, 0$

Finish [2] is false and $\overset{6,0,0}{Need_2} > \overset{5,3,2}{Work}$

So P_2 must wait

Step 2

For i=3

$Need_3 = 0, 1, 1$

Finish [3] = false and $\overset{0,1,1}{Need_3} < \overset{5,3,2}{Work}$

So P_3 must be kept in safe sequence

Step 3

$\overset{5,3,2}{Work} = Work + \overset{2,1,1}{Allocation_3}$

Work =

A	B	C
7	4	3

Finish =

0	1	2	3	4
false	true	false	true	false

Step 2

For i = 4

$Need_4 = 4, 3, 1$

Finish [4] = false and $\overset{4,3,1}{Need_4} < \overset{7,4,3}{Work}$

So P_4 must be kept in safe sequence

Step 3

$\overset{7,4,3}{Work} = Work + \overset{0,0,2}{Allocation_4}$

Work =

A	B	C
7	4	5

Finish =

0	1	2	3	4
false	true	false	true	true

Step 2

For i = 0

$Need_0 = 7, 4, 3$

Finish [0] is false and $\overset{7,4,3}{Need} < \overset{7,4,5}{Work}$

So P_0 must be kept in safe sequence

Step 3

$\overset{7,4,5}{Work} = Work + \overset{0,1,0}{Allocation_0}$

Work =

A	B	C
7	5	5

Finish =

0	1	2	3	4
true	true	false	true	true

Step 2

For i = 2

$Need_2 = 6, 0, 0$

Finish [2] is false and $\overset{6,0,0}{Need_2} < \overset{7,5,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

$\overset{7,5,5}{Work} = Work + \overset{3,0,2}{Allocation_2}$

Work =

A	B	C
10	5	7

Finish =

0	1	2	3	4
true	true	true	true	true

Step 4

Finish [i] = true for $0 \leq i \leq n$

Hence the system is in Safe state

The safe sequence is P_1,P_3 , P_4 ,P_0 ,P_2

Hence the new system state is safe, so we can immediately grant the request for process P1 .

- As the processes enter the system, they must predict the maximum number of resources needed which is impractical to determine.
- In this algorithm, the number of processes remain fixed which is not possible in interactive systems.
- This algorithm requires that there should be a fixed number of resources to allocate. If a device breaks and becomes suddenly unavailable the algorithm would not work.
- Overhead cost incurred by the algorithm can be high when there are many processes and resources because it has to be invoked for every processes.

- **C PROGRAM:**

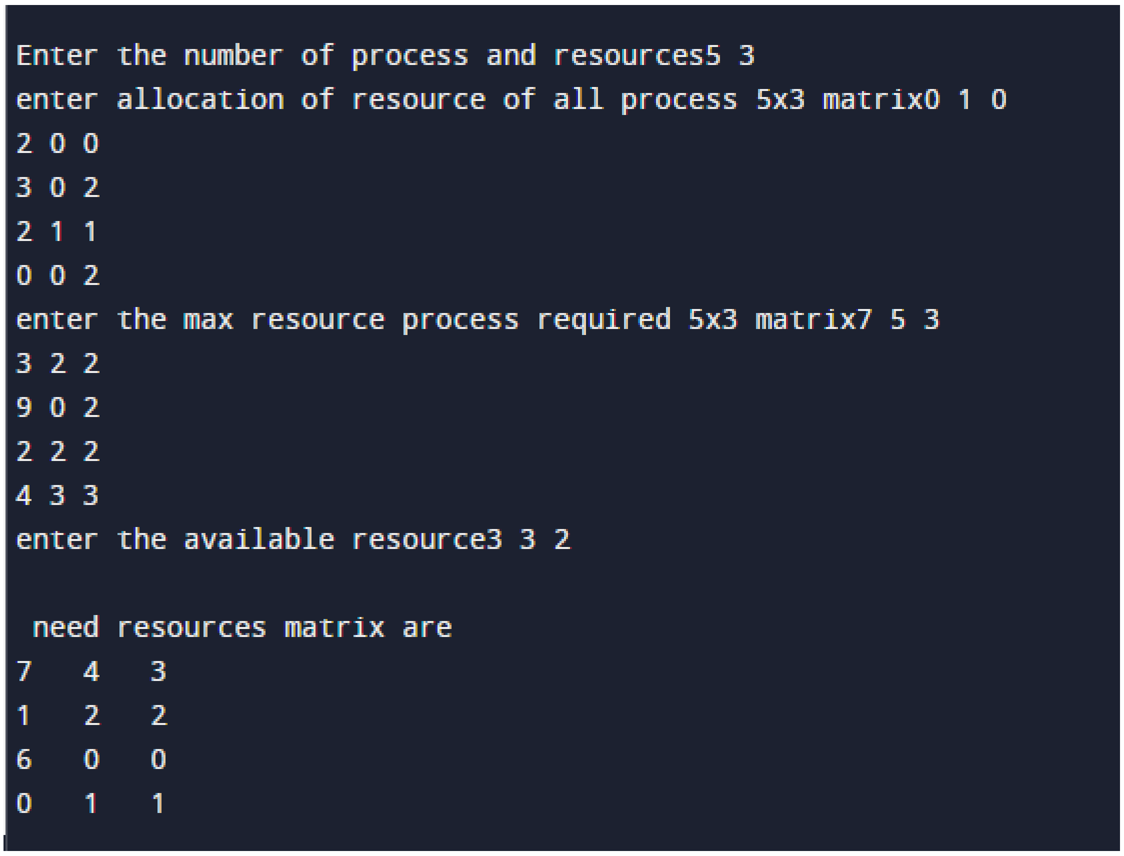
```
#include<stdio.h>

int main()
{
int p, c, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3],
finish[5], terminate = 0;

printf("Enter the number of process and resources");
scanf("%d %d", &p, &c);

printf("enter allocation of resource of all process %dx%d matrix",p,c);
for (i = 0; i < p; i++)
{
for (j = 0; j < c; j++)
{
scanf("%d", &alc[i][j]);
}
}

printf("enter the max resource process required %dx%d matrix", p, c);
for (i = 0; i < p; i++)
{
for (j = 0; j < c; j++)
{
scanf("%d", &max[i][j]);
}
}
```

```
printf("enter the available resource");
for (i = 0; i < c; i++)
scanf("%d", &available[i]);

printf("\n need resources matrix are\n");
for (i = 0; i < p; i++)
{
for (j = 0; j < c; j++)
{
need[i][j] = max[i][j] - alc[i][j];
printf("%d\t", need[i][j]);
}
printf("\n");
}

for (i = 0; i < p; i++)
{
finish[i] = 0;
}

while (count < p)
{
for (i = 0; i < p; i++)
{
if (finish[i] == 0)
{
for (j = 0; j < c; j++)
{
```

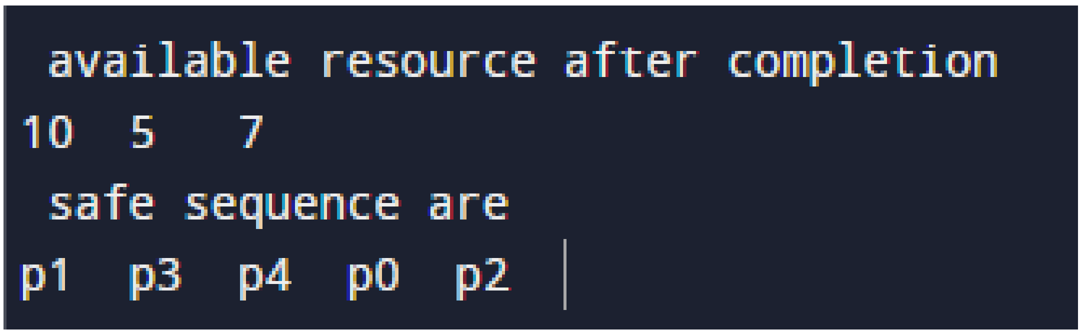
```
if (need[i][j] > available[j])
break;
}

if (j == c) {
safe[count] = i;
finish[i] = 1;
for (j = 0; j < c; j++) {
available[j] += alc[i][j];
}
count++;
terminate = 0;
}
else {
terminate++;
}
}
}

if (terminate == (p - 1))
{
printf("safe sequence does not exist");
break;
}
}

if (terminate != (p - 1))
{
printf("\n available resource after completion\n");
```

```
for (i = 0; i < c; i++)
{
printf("%d\t", available[i]);
}
printf("\n safe sequence are\n");
for (i = 0; i < p; i++)
{
printf("p%d\t", safe[i]);
}
}
return 0;
}
```

- **OUTPUT:**



```
Enter the number of process and resources5 3
enter allocation of resource of all process 5x3 matrix0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
enter the max resource process required 5x3 matrix7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
enter the available resource3 3 2

 need resources matrix are
7   4   3
1   2   2
6   0   0
0   1   1
```



```
 available resource after completion
10   5    7
 safe sequence are
p1   p3   p4   p0   p2
```

<u>**CONCLUSION:**</u> **Hence, we have successfully implemented the concept of deadlock avoidance through Banker’s Algorithm.**