Definition:

The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra.

Banker's Algorithm working principle:

It tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes a "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

Available: A vector of length m. It shows number of available resources of each type. If Available[i] = k, then k instances of resource R_i are available.

Max: An n×m matrix that contain maximum demand of each process. If Max[i,j] = k, then process P_i can request maximum k instances of resource type R_j.

Allocation: An n×m matrix that contain number of resources of each type currently allocated to each process. If Allocation[i,j] = k, then P_i is currently allocated k instances of resource type R_j.

Need: An n×m matrix that shows the remaining resource need of each process. If Need[i,j] = k, then process P_i may need k more instances of resource type R_j to complete the task.

```
EXPERIMENT NO: 9 Banker's Algorithm

#include <stdio.h>
int curr[5][5], maxclaim[5][5], avl[5];
int alloc[5] = {0, 0, 0, 0, 0};
int maxres[5], running[5], safe=0;
int count = 0, i, j, exec, r, p, k = 1;

int main()
```

```
{
    printf("nEnter the number of processes: ");
    scanf("%d", &p);

    for (i = 0; i < p; i++) {
        running[i] = 1;
        count++;
    }

    printf("nEnter the number of resources: ");
    scanf("%d", &r);

    for (i = 0; i < r; i++) {
        printf("nEnter the resource for instance %d: ", k++);
        scanf("%d", &maxres[i]);
    }

    printf("nEnter maximum resource table:n");
    for (i = 0; i < p; i++) {
        for(j = 0; j < r; j++) {
            scanf("%d", &maxclaim[i][j]);
        }
    }

    printf("nEnter allocated resource table:n");
    for (i = 0; i < p; i++) {
        for(j = 0; j < r; j++) {
            scanf("%d", &curr[i][j]);
        }
    }

    printf("nThe resource of instances: ");
    for (i = 0; i < r; i++) {
        printf("t%d", maxres[i]);
    }

    printf("nThe allocated resource table:n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            printf("t%d", curr[i][j]);
        }

        printf("n");
    }

    printf("nThe maximum resource table:n");
    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
```

```
            printf("t%d", maxclaim[i][j]);
        }

        printf("n");
    }

    for (i = 0; i < p; i++) {
        for (j = 0; j < r; j++) {
            alloc[j] += curr[i][j];
        }
    }

    printf("nAllocated resources:");
    for (i = 0; i < r; i++) {
        printf("t%d", alloc[i]);
    }

    for (i = 0; i < r; i++) {
        avl[i] = maxres[i] - alloc[i];
    }

    printf("nAvailable resources:");
    for (i = 0; i < r; i++) {
        printf("t%d", avl[i]);
    }
    printf("n");

    //Main procedure goes below to check for unsafe state.
    while (count != 0) {
        safe = 0;
        for (i = 0; i < p; i++) {
            if (running[i]) {
                exec = 1;
                for (j = 0; j < r; j++) {
                    if (maxclaim[i][j] - curr[i][j] > avl[j]) {
                        exec = 0;
                        break;
                    }
                }
                if (exec) {
                    printf("nProcess%d is executingn", i + 1);
                    running[i] = 0;
                    count--;
                    safe = 1;

                    for (j = 0; j < r; j++) {
                        avl[j] += curr[i][j];
                    }
```

```
                    break;
                }
            }
        }
        if (!safe) {
            printf("nThe processes are in unsafe state.n");
            break;
        } else {
            printf("nThe process is in safe state");
            printf("nSafe sequence is:");

            for (i = 0; i < r; i++) {
                printf("t%d", avl[i]);
            }

            printf("n");
        }
    }
}
```

OUTPUT:
Enter the number of resources:4

Enter the number of processes:5

Enter Claim Vector:8 5 9 7

Enter Allocated Resource Table:
2 0 1 1
0 1 2 1
4 0 0 3
0 2 1 0
1 0 3 0

Enter Maximum Claim table:
3 2 1 4
0 2 5 2
5 1 0 5
1 5 3 0
3 0 3 3

The Claim Vector is: 8 5 9 7
The Allocated Resource Table:

```
2 0 1 1
0 1 2 1
4 0 0 3
0 2 1 0
1 0 3 0

The Maximum Claim Table:
3 2 1 4
0 2 5 2
5 1 0 5
1 5 3 0
3 0 3 3

Allocated resources: 7 3 7 5
Available resources: 1 2 2 2

Process3 is executing

The process is in safe state
Available vector: 5 2 2 5
Process1 is executing

The process is in safe state
Available vector: 7 2 3 6
Process2 is executing

The process is in safe state
Available vector: 7 3 5 7
Process4 is executing

The process is in safe state
Available vector: 7 5 6 7
Process5 is executing

The process is in safe state
Available vector: 8 5 9 7
```