What is Banker's Algorithm?

Banker's Algorithm is used in Operating Systems **to avoid deadlock**.
It checks whether resource allocation to a process will keep the system in a **safe state**.
A *safe state* means there exists at least one sequence in which all processes can finish.

✅ Code Breakdown and Explanation

🔹 1. Global Variables and Definitions

```
#define MAX 10


int alloc[MAX][MAX], maxNeed[MAX][MAX], need[MAX][MAX];

int avail[MAX];

int pCount, rCount;
```

- MAX defines the maximum limit for processes/resources.
- alloc[][] → Allocation Matrix (currently allocated resources)
- maxNeed[][] → Maximum Matrix (maximum resources required by each process)
- need[][] → Need Matrix (resources still required by each process)
- avail[] → Available resources vector
- pCount & rCount → total processes and resources

🔹 2. Function: isSafe()

Checks whether the system is in **SAFE or UNSAFE** state.

Step-by-step:

```
int work[MAX], finish[MAX] = {0}, safeSeq[MAX];

int i, j, count = 0;
```

- work[] acts as a temporary copy of available resources.
- finish[] keeps track if a process can complete (0 = not finished, 1 = finished)
- safeSeq[] stores the safe sequence if it exists.

```
for (i = 0; i < rCount; i++)

  work[i] = avail[i];
```

Copies available resources into work.

Main logic loop:

```
while (count < pCount) {
  int found = 0;
  for (i = 0; i < pCount; i++) {
    if (!finish[i]) {
      int canRun = 1;
```

- Loops through each process and checks if it can run with current available resources.

```
for (j = 0; j < rCount; j++) {
  if (need[i][j] > work[j]) {
    canRun = 0;
    break;
  }
}
```

- Compares the **Need Matrix** with currently available resources.
- If need <= work, process can run.

```
if (canRun) {
  for (j = 0; j < rCount; j++)
    work[j] += alloc[i][j];

  safeSeq[count++] = i;
  finish[i] = 1;
  found = 1;
}
```

- After a process completes, its allocated resources are returned to work.
- Process index added to safeSeq.

```
if (!found) {
  printf("\nSystem is NOT in SAFE state!\n");
  return 0;
}
```

- If no runnable process is found → **unsafe state**, return 0 (false)

✅ If loop successfully ends:

```
printf("\nSystem is in SAFE state.\nSafe Sequence: ");
```

Prints safe sequence.

🔹 3. Function: request()

Handles **resource request from a process**.

```
for (i = 0; i < rCount; i++) {
    if (req[i] > need[p][i]) { ... }
    if (req[i] > avail[i]) { ... }
}
```

- Check 1: Request should not exceed **maximum need**
- Check 2: Resources must be **available**

Temporarily allocate

```
avail[i] -= req[i];
alloc[p][i] += req[i];
need[p][i] -= req[i];
```

Assume request is granted and modify tables.

✅ Run safety check:

```
if (isSafe()) {
    printf("Request can be granted to P%d.\n", p);
}
```

❌ If unsafe → rollback changes:

```
avail[i] += req[i];
alloc[p][i] -= req[i];
need[p][i] += req[i];
printf("Request CANNOT be granted as it leads to UNSAFE state.\n");
```

🔹 4. main() Function

Handles **input & initial safety check**

```
scanf("%d", &pCount);
scanf("%d", &rCount);
```

Get process & resource count.

```
printf("Enter Allocation Matrix:\n");
```

Takes input for:

Allocation Maximum Available

Then computes Need matrix:

need[i][j] = maxNeed[i][j] - alloc[i][j];

✅ Calls initial isSafe()
✅ Takes a request and calls request()

✅ What To Write in Exam (Theory)

◆ Banker's Algorithm Steps

1. Compute Need = Max – Allocation
2. Check if a process can complete (need <= available)
3. If yes, mark it finish and release its resources
4. Repeat until all processes finish
5. If all can finish → **Safe State**
6. If no safe sequence → **Unsafe State (deadlock possible)**

◆ Why It Is Called "Banker’s Algorithm"?

Because like a banker gives loans only if he can still satisfy all customers later, OS allocates resources only if it remains safe.