Banker's Algorithm Implementation Report

Name: Hashim Abdulla
Course: Operating Systems
Assignment: Homework 2 - Banker's Algorithm
Due Date: November 9, 2025

GitHub link: https://github.com/HashAbdulla/Homework2-BankersAlgo [ALL DOCUMENTATION/SOURCE CODE IN THIS REPO]

AI Tool Disclosure

AI Used: Claude (Claude Sonnet 4.5)

How I Used It:

- Had it create the initial code structure and function templates for the Banker's Algorithm
- It gave me formatting for output displays and trace logging
- It provided starter code for matrix operations (is_less_or_equal, add_vectors)
- Added comprehensive comments and documentation throughout the code
- Helped me draft this report template and then I added relevant info

Human Work (My Contributions):

- Implemented the complete safety test algorithm logic with proper tie-breaking
- Designed and coded the resource request validation with pretend-allocate/rollback mechanism
- Written all the test scenarios and verified correctness against assignment requirements
- Debugged all edge cases and ensured deterministic behavior (tie-breaking by process index)
- Performed thorough testing with provided dataset and created additional test cases
- Analyzed results and wrote all explanations and insights in Sections 1 and 2
- Structured the trace output to clearly show Work and Finish vectors at each step
- Final code review, optimization, and all edits to meet assignment specifications

---

Section 1: Outputs

Test 1: Baseline Safety Test at S0

Input Data:

- **Processes:** 5 (P1 through P5)
- **Resource Types:** 3 (A, B, C)
- **Total Resources:** A=10, B=5, C=7
- **Available at S0:** [3, 3, 2]

```
Allocation:
  P1: [0, 1, 0]
  P2: [2, 0, 0]
  P3: [3, 0, 2]
  P4: [2, 1, 1]
  P5: [0, 0, 2]

Max:
  P1: [7, 5, 3]
  P2: [3, 2, 2]
  P3: [9, 0, 2]
  P4: [2, 2, 2]
  P5: [4, 3, 3]

Need:
  P1: [7, 4, 3]
  P2: [1, 2, 2]
  P3: [6, 0, 0]
  P4: [0, 1, 1]
  P5: [4, 3, 1]
```

[System configuration, Allocation, Max, and computed Need matrices]

Safety Algorithm Execution Trace:

```
Step 1:
  Process P2 can be satisfied
  Need[P2] = [1, 2, 2] <= Work = [3, 3, 2]
  After P2 completes:
    Work = Work + Allocation[P2] = [5, 3, 2]
    Finish[P2] = True

Step 2:
  Process P4 can be satisfied
  Need[P4] = [0, 1, 1] <= Work = [5, 3, 2]
  After P4 completes:
    Work = Work + Allocation[P4] = [7, 4, 3]
    Finish[P4] = True

Step 3:
  Process P1 can be satisfied
  Need[P1] = [7, 4, 3] <= Work = [7, 4, 3]
  After P1 completes:
    Work = Work + Allocation[P1] = [7, 5, 3]
    Finish[P1] = True
```

[P2, P4, P1 selections with Work vector updates]

Output

```
Step 4:
  Process P3 can be satisfied
  Need[P3] = [6, 0, 0] <= Work = [7, 5, 3]
  After P3 completes:
    Work = Work + Allocation[P3] = [10, 5, 5]
    Finish[P3] = True

Step 5:
  Process P5 can be satisfied
  Need[P5] = [4, 3, 1] <= Work = [10, 5, 5]
  After P5 completes:
    Work = Work + Allocation[P5] = [10, 5, 7]
    Finish[P5] = True

------------------------------------------------------------
 OK All processes can finish - System is SAFE
------------------------------------------------------------

============================================================
RESULT: System is SAFE
Safe Sequence: <P2,P4,P1,P3,P5>
============================================================
```

[P3, P5 selections and the safe sequence]

We can see that the system is in a safe state. The algorithm finds the sequence <P2,P4,P1,P3,P5> by iteratively selecting processes whose needs can be satisfied with the available resources. Each completed process releases its resources, enabling the next process to finish.

Test 2: Denied Request Scenario

```
############################################################
# TEST 2: RESOURCE REQUEST - DENIED SCENARIO
############################################################

Scenario: P1 requests (0, 4, 0)
Expected: DENIED (exceeds Need or leads to unsafe state)

============================================================
RESOURCE REQUEST from P1
============================================================
Request: [0, 4, 0]
OK Check 1 passed: Request [0, 4, 0] <= Need [7, 4, 3]
DENIED: Request [0, 4, 0] exceeds Available [3, 3, 2]

============================================================
RESULT: DENIED
Reason: DENIED: Request [0, 4, 0] exceeds Available [3, 3, 2]
============================================================
```

The request is denied because P1 requests 4 units of resource B, but only 3 units are available. The request fails at Check 2 (Request <= Available) before safety testing is even performed.

Test 3: Granted Request Scenario

```
============================================================
RESOURCE REQUEST from P2
============================================================
Request: [1, 0, 2]
OK Check 1 passed: Request [1, 0, 2] <= Need [1, 2, 2]
OK Check 2 passed: Request [1, 0, 2] <= Available [3, 3, 2]

Pretending to allocate resources...
New Allocation[P2] would be: [3, 0, 2]
New Available would be: [2, 3, 0]
```

[Request details, both checks passing, pretend allocation]

```
Allocation:
  P1: [0, 1, 0]
  P2: [3, 0, 2]
  P3: [3, 0, 2]
  P4: [2, 1, 1]
  P5: [0, 0, 2]

Max:
  P1: [7, 5, 3]
  P2: [3, 2, 2]
  P3: [9, 0, 2]
  P4: [2, 2, 2]
  P5: [4, 3, 3]

Need:
  P1: [7, 4, 3]
  P2: [0, 2, 0]
  P3: [6, 0, 0]
  P4: [0, 1, 1]
  P5: [4, 3, 1]

Available: [2, 3, 0]
```

[New state matrices]

[After all steps…]

```
------------------------------------------------------------
 OK All processes can finish - System is SAFE
------------------------------------------------------------

 GRANTED: Request leads to safe state with sequence <P2,P4,P1,P3,P5>

============================================================
RESULT: GRANTED
Reason: GRANTED: Request leads to safe state with sequence <P2,P4,P1,P3,P5>
============================================================
```

The request is granted since it passes all validation checks and leads to a safe state. After pretend allocation (Available = [2,3,0]), the safety test finds sequence <P2,P4,P1,P3,P5>, confirming the system infact remains deadlock-free.

Section 2: Implementation Details

(Code inside this document was formatted using the Microsoft Word "Easy Code Formatter" extension to enhance readability and presentation before converting the file to PDF.)

Data Structures and Design Choices

I went with Python lists to handle all the matrices and vectors here, mainly since they're straightforward,no extra packages needed. The Allocation plus Max setups are basically grids made from nested lists, where allocation[i][j] tells you how much of resource j is taken by process i right now. Picked regular lists instead of arrays seeing as we aren't doing heavy math, just adding small numbers or checking which one's bigger.

The Need matrix doesn't stick around in my code. Rather, I work it out live through the calculate_need() function when it's actually required. That decision wasn't accidental because Need ties closely to Allocation and Max, any shift in Allocation (say, after approving a request) would force a recalculation regardless. Pulling it together just-in-time ensures things stay in sync without risking errors from outdated values.

I also made a few small helper functions is_less_or_equal() and add_vectors() just to keep things neat. Since the logic keeps needing item-wise checks and sums, these bits do that heavy lifting. If I skip them then I'd be stuck tossing loops all over just to verify stuff like allocation[i][0] <= work[0], which would get messy fast.

Implementing the Tie-Breaking Rule

Figuring out how to handle several running tasks wasn't easy for me especially forcing the system to choose the one with the smallest number every single time. That's key, since skipping this step might lead to varied correct orders on separate runs, messing up fair evaluation.

The approach I went with was simple: I ran through each process one by one, starting at 0 up to n- 1:

```
1. for i in range(n):
2.     if not finish[i] and is_less_or_equal(need[i], work):
3.         # Select this process
4.         break
5.
```

Because Python's range(n) counts from 0 up like 0, 1, 2, 3, 4, the very first process meeting the requirement is picked. Suppose P2 and P4 are ready; P2 runs since it comes earlier in line. That way, things unfold the same every time, just like the task wanted.

The Pretend-Allocate Mechanism

The resource request feature felt tricky to build since tweaking the state directly won't work. Instead, I went with duplicating lists in Python to simulate how things might play out.

Whenever a procedure asks for something, I generate fresh duplicates of the resource table along with the free list

```
1. new_allocation = [row.copy() for row in allocation]
2. new_available = available.copy()
3.
```

After that, I tweak those duplicates like the request was granted. What matters is the allocation and free resources aren't altered. So there's zero need for a rollback step. When the safety test fails in this new state, I simply leave it alone. Nothing breaks since I didn't touch the real state at any point.

This just seemed like the easiest fix. Instead of logging edits or undoing them later, I went with a duplicate. It's straightforward, plus it keeps the starting point untouched.

Safety Test Implementation

The safety check sticks close to the standard method. I start by copying Available into Work while marking every Finish flag as False. After that, the core loop runs nonstop , either till every process wraps up (that's safe) or things halt mid-way (which means unsafe).

The loop checks every process where Finish[i] is still false while Need[i] fits within Work. Once spotted, I act like that process ends, tossing its allocated resources into Work and flipping its status to done. Plus, I show updates each time, so you watch how Work expands as tasks wrap up.

The hard part was knowing when to quit. In case every n process gets added into the safe list just fine, then everything's under control. Yet if we check each one and none of them can move forward, then we're stuck in a deadlock stitation. I set up a found flag to see if anything shifted during every round of the main cycle.

Key Insights

How tiny requests might lead to trouble:

While trying out various test cases, I realized asking for only one or two bits could mess up the whole setup, which felt odd at first. Even if those numbers look harmless, they might still break things behind the scenes. That's because staying safe doesn't depend on how many you take, but whether there are still enough options left to finish what's needed down the line.

Imagine this: say Available is [3, 3, 2], then a process requests [2, 2, 1] - suddenly you've got only [1, 1, 1] left. Seems okay at first glance; however, when each leftover task requires 2 units or more of any resource, none can even begin. That single request messed things up by using pieces everyone else depends on. When one job stalls, its held-back resources don't get freed, so others wait endlessly, dragging everything into deadlock. The real issue is the scraps you leave blocking the next move.

Safe sequences arent unique

One more thing stood out; same system state, yet plenty of working safe sequences show up. Mine came back with , p2, p4, p1, p3, p5 though the task said that's just one of several right picks. Had I gone for a different tie-breaking rule, the result would've shifted but still been valid.

This idea clicks once you see how the algorithm operates. At each point, multiple processes may be ready to go so long as we choose one and move forward, we'll clear all in time. The check simply confirms at least one safe sequence is possible; finding all isn't necessary. On a real OS, scheduling also weighs things like urgency or balance, meaning the real run order won't hinge solely on Banker's algorithm.

Testing Approach

I checked my setup by testing it on the given S0 data, making sure it spotted the expected safe sequences. After that, I built scenarios meant to break, for instance a request going past what's available, which needs instant rejection, or one fitting on paper but leading to risk, blocked only during the safety test. Plus, I walked through the steps myself on paper for the initial scenario, just to confirm my Work vector lined up with manual math.

The hardest error to spot came down to a basic oversight where I missed copying the data rows right. Instead of proper duplication, I used new_allocation = allocation.copy(), but that's just a shallow copy meaning any tweak to new_allocation[i][j] also messed up the source. What worked was using [row.copy() for row in allocation], for a deep copy. This sort of odd behavior in Python isn't obvious until you've gone through a few mistakes.

Compilation and Execution Instructions

Language: Python

Dependencies: None (uses only standard library)

Execution:

python bankers_algo.py

Expected Output:

- Test 1: SAFE with sequence <P2, P4, P1, P3, P5>

- Test 2: DENIED (exceeds available)
- Test 3: GRANTED with new safe sequence

End of Report