1. Show the execution of Basic and Advanced Linux commands. (any 5 commands from each type)

Here are examples of **Basic** and **Advanced** Linux commands along with their execution and descriptions:

Basic Linux Commands

1. **`ls`** – List directory contents.
   ```bash
   ls
   ```
   This command displays the files and directories in the current directory.

2. **`cd`** – Change directory.
   ```bash
   cd /path/to/directory
   ```
   Changes the current working directory to the specified path.

3. **`pwd`** – Print working directory.
   ```bash
   pwd
   ```
   Displays the full path of the current working directory.

4. **`mkdir`** – Make directories.
   ```bash
   mkdir new_directory
   ```
   Creates a new directory named `new_directory`.

5. **`cp`** – Copy files and directories.
   ```bash
   cp source_file destination_file
   ```
   Copies the file from `source_file` to `destination_file`.

Advanced Linux Commands

1. **`grep`** – Search text using patterns.
   ```bash
   ```

   grep "pattern" file.txt
   ~~~
   Searches for the specified pattern in `file.txt` and prints the matching lines.

2. **`find`** – Search for files in a directory hierarchy.
   ```bash
   find /home/user -name "*.txt"
   ~~~
   Finds all `.txt` files in the `/home/user` directory and its subdirectories.

3. **`tar`** – Archive files.
   ```bash
   tar -czvf archive.tar.gz /path/to/directory
   ```
   Creates a compressed archive (`archive.tar.gz`) of the specified directory.

4. **`chmod`** – Change file permissions.
   ```bash
   chmod 755 file.sh
   ```
   Changes the file permissions of `file.sh` to `755` (read, write, execute for owner; read and execute for others).

5. **`awk`** – Pattern scanning and processing language.
   ```bash
   awk '{print $1}' file.txt
   ```
   Prints the first column of data from `file.txt`.

These are a few examples of basic and advanced Linux commands that are frequently used in system administration, programming, and daily usage.

2. Write a shell Script to find the sum of digits of the entered number.

```
#!/bin/bash

# Ask the user to enter a number
echo "Enter a number: "
read number

# Initialize sum to 0
sum=0
```

```
# Loop through each digit of the number
while [ $number -gt 0 ]
do
    # Get the last digit of the number
    digit=$((number % 10))

    # Add the digit to the sum
    sum=$((sum + digit))

    # Remove the last digit from the number
    number=$((number / 10))
done

# Output the sum of digits
echo "The sum of the digits is: $sum"
```

#!/bin/bash

```
# Ask the user to enter a number
echo "Enter a number: "
read number

# Initialize sum to 0
sum=0

# Convert the number into a string to iterate through each digit
for digit in $(echo $number | sed -e 's/\(.\)/\1 /g')
do
    # Add each digit to the sum
    sum=$((sum + digit))
done

# Output the sum of digits
echo "The sum of the digits is: $sum"
```

3. Write a shell Script to sort array elements in Descending order.

```
#!/bin/bash

# Define an array of numbers
echo "Enter the number of elements in the array:"
read n

echo "Enter the elements of the array:"
for ((i=0; i<n; i++))
```

```
do
    read arr[$i]
done

# Sort the array in descending order using bubble sort
for ((i=0; i<n-1; i++))
do
    for ((j=i+1; j<n; j++))
    do
        if [ ${arr[i]} -lt ${arr[j]} ]; then
            # Swap the elements if they are in the wrong order
            temp=${arr[i]}
            arr[$i]=${arr[j]}
            arr[$j]=$temp
        fi
    done
done

# Print the sorted array
echo "Sorted array in descending order:"
for ((i=0; i<n; i++))
do
    echo -n "${arr[$i]} "
done
echo
```

4. Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the First Come and First Serve (FCFS) Scheduling. (enter suitable number of processes, CPU burst, and Arrival Time)

```
import java.util.Scanner;

public class FCFS {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input number of processes
        System.out.print("Enter the number of processes: ");
        int n = sc.nextInt();

        int[] process = new int[n];
        int[] arrivalTime = new int[n];
        int[] burstTime = new int[n];
        int[] waitingTime = new int[n];
        int[] turnaroundTime = new int[n];
        int[] completionTime = new int[n];
```

```
// Input process details
for (int i = 0; i < n; i++) {
    process[i] = i + 1; // Process IDs (P1, P2, ...)
    System.out.print("Enter arrival time of Process " + (i + 1) + ": ");
    arrivalTime[i] = sc.nextInt();
    System.out.print("Enter burst time of Process " + (i + 1) + ": ");
    burstTime[i] = sc.nextInt();
}

// Sort processes by Arrival Time
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
        if (arrivalTime[j] > arrivalTime[j + 1]) {
            // Swap Arrival Time
            int temp = arrivalTime[j];
            arrivalTime[j] = arrivalTime[j + 1];
            arrivalTime[j + 1] = temp;

            // Swap Burst Time
            temp = burstTime[j];
            burstTime[j] = burstTime[j + 1];
            burstTime[j + 1] = temp;

            // Swap Process ID
            temp = process[j];
            process[j] = process[j + 1];
            process[j + 1] = temp;
        }
    }
}

// Compute Completion Time
completionTime[0] = arrivalTime[0] + burstTime[0];
for (int i = 1; i < n; i++) {
    if (completionTime[i - 1] < arrivalTime[i]) {
        completionTime[i] = arrivalTime[i] + burstTime[i]; // Process waits for CPU to be free
    } else {
        completionTime[i] = completionTime[i - 1] + burstTime[i];
    }
}

// Compute Turnaround Time and Waiting Time
for (int i = 0; i < n; i++) {
    turnaroundTime[i] = completionTime[i] - arrivalTime[i]; // TAT = Completion Time - Arrival Time
    waitingTime[i] = turnaroundTime[i] - burstTime[i];     // WT = TAT - Burst Time
}

// Print the results
```

```
        System.out.println("\nProcess\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time");
        for (int i = 0; i < n; i++) {
            System.out.println("P" + process[i] + "\t\t" + arrivalTime[i] + "\t\t" + burstTime[i] + "\t\t" +
                    completionTime[i] + "\t\t" + turnaroundTime[i] + "\t\t" + waitingTime[i]);
        }

        // Calculate and print average TAT and WT
        double totalTAT = 0, totalWT = 0;
        for (int i = 0; i < n; i++) {
            totalTAT += turnaroundTime[i];
            totalWT += waitingTime[i];
        }
        System.out.printf("\nAverage Turnaround Time: %.2f", totalTAT / n);
        System.out.printf("\nAverage Waiting Time: %.2f", totalWT / n);

        sc.close();
    }
}
```

5. Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the Shortest Job First (Preemptive and Non-Preemptive) Scheduling. (enter suitable number of processes, CPU burst, and Arrival Time)

```
import java.util.*;

public class SJF {
    static class Process {
        int id, arrivalTime, burstTime, completionTime, turnaroundTime, waitingTime, remainingTime;

        public Process(int id, int arrivalTime, int burstTime) {
            this.id = id;
            this.arrivalTime = arrivalTime;
            this.burstTime = burstTime;
            this.remainingTime = burstTime;
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the number of processes
        System.out.print("Enter the number of processes: ");
        int n = sc.nextInt();
        List<Process> processes = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            System.out.print("Enter arrival time of Process " + (i + 1) + ": ");
            int arrival = sc.nextInt();
            System.out.print("Enter burst time of Process " + (i + 1) + ": ");
            int burst = sc.nextInt();
            processes.add(new Process(i + 1, arrival, burst));
        }

        System.out.println("\nChoose Scheduling Type:");
        System.out.println("1. Non-Preemptive");
        System.out.println("2. Preemptive");
        int choice = sc.nextInt();

        if (choice == 1) {
            sjfNonPreemptive(processes, n);
        } else if (choice == 2) {
            sjfPreemptive(processes, n);
        } else {
            System.out.println("Invalid choice!");
        }
    }

    // Non-Preemptive SJF
    private static void sjfNonPreemptive(List<Process> processes, int n) {
        int currentTime = 0, completed = 0;
        while (completed < n) {
            Process selectedProcess = null;

            // Select the shortest job that has arrived
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && p.remainingTime > 0) {
                    if (selectedProcess == null || p.burstTime < selectedProcess.burstTime) {
                        selectedProcess = p;
                    }
                }
            }

            if (selectedProcess != null) {
                // Execute the process to completion
                currentTime += selectedProcess.burstTime;
                selectedProcess.completionTime = currentTime;
                selectedProcess.turnaroundTime = selectedProcess.completionTime -
selectedProcess.arrivalTime;
                selectedProcess.waitingTime = selectedProcess.turnaroundTime - selectedProcess.burstTime;
                selectedProcess.remainingTime = 0; // Process is completed
                completed++;
            } else {
                currentTime++; // Idle time if no process is ready
            }
        }

        printResults(processes, "Non-Preemptive SJF");
    }

    // Preemptive SJF
    private static void sjfPreemptive(List<Process> processes, int n) {
        int currentTime = 0, completed = 0;

        while (completed < n) {
            Process selectedProcess = null;

            // Select the shortest job that has arrived
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && p.remainingTime > 0) {
                    if (selectedProcess == null || p.remainingTime < selectedProcess.remainingTime) {
                        selectedProcess = p;
                    }
                }
            }

            if (selectedProcess != null) {
                // Execute for one unit of time
                selectedProcess.remainingTime--;
                currentTime++;

                // If the process is completed
                if (selectedProcess.remainingTime == 0) {
                    completed++;
                    selectedProcess.completionTime = currentTime;
                    selectedProcess.turnaroundTime = selectedProcess.completionTime -
selectedProcess.arrivalTime;
                    selectedProcess.waitingTime = selectedProcess.turnaroundTime -
selectedProcess.burstTime;
                }
            } else {
                currentTime++; // Idle time if no process is ready
            }
        }

        printResults(processes, "Preemptive SJF");
    }

    // Function to print the results
    private static void printResults(List<Process> processes, String schedulingType) {
        System.out.println("\n" + schedulingType + " Results:");
        System.out.println("Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time");

        for (Process p : processes) {
            System.out.println("P" + p.id + "\t\t" + p.arrivalTime + "\t\t" + p.burstTime + "\t\t" +
                    p.completionTime + "\t\t\t" + p.turnaroundTime + "\t\t\t" + p.waitingTime);
        }

        // Calculate average TAT and WT
        double totalTAT = 0, totalWT = 0;
        for (Process p : processes) {
            totalTAT += p.turnaroundTime;
            totalWT += p.waitingTime;
        }

        System.out.println("\nAverage Turnaround Time: " + (totalTAT / processes.size()));
        System.out.println("Average Waiting Time: " + (totalWT / processes.size()));
    }
}
```

6. Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the Priority (Preemptive and Non-Preemptive) Scheduling. (enter suitable number of processes, CPU burst, and Arrival Time)

```
import java.util.ArrayList;
import java.util.Scanner;

public class PriorityScheduling {

    static class Process {
        int id, arrivalTime, burstTime, priority, completionTime, turnaroundTime, waitingTime, remainingTime;

        public Process(int id, int arrivalTime, int burstTime, int priority) {
            this.id = id;
            this.arrivalTime = arrivalTime;
            this.burstTime = burstTime;
            this.priority = priority;
            this.remainingTime = burstTime; // For preemptive scheduling
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input number of processes
        System.out.print("Enter the number of processes: ");
        int n = sc.nextInt();

        ArrayList<Process> processes = new ArrayList<>();

        // Input process details
        for (int i = 1; i <= n; i++) {
            System.out.print("Enter arrival time of Process " + i + ": ");
            int arrivalTime = sc.nextInt();
            System.out.print("Enter burst time of Process " + i + ": ");
            int burstTime = sc.nextInt();
            System.out.print("Enter priority of Process " + i + " (lower value = higher priority): ");
            int priority = sc.nextInt();
            processes.add(new Process(i, arrivalTime, burstTime, priority));
        }

        // Select scheduling mode
        System.out.println("\nSelect Scheduling Mode:");
        System.out.println("1. Preemptive Priority Scheduling");
        System.out.println("2. Non-Preemptive Priority Scheduling");
        int mode = sc.nextInt();

        if (mode == 1) {
            preemptivePriorityScheduling(processes, n);
        } else if (mode == 2) {
            nonPreemptivePriorityScheduling(processes, n);
        } else {
            System.out.println("Invalid mode selected.");
        }
    }

    // Preemptive Priority Scheduling
    public static void preemptivePriorityScheduling(ArrayList<Process> processes, int n) {
        int currentTime = 0, completed = 0;
        double totalTAT = 0, totalWT = 0;

        while (completed < n) {
            // Find the highest-priority process available at the current time
            Process currentProcess = null;
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && p.remainingTime > 0) {
                    if (currentProcess == null || p.priority < currentProcess.priority) {
                        currentProcess = p;
                    }
                }
            }

            if (currentProcess == null) {
                currentTime++;
                continue;
            }

            // Execute the process for 1 unit of time
            currentProcess.remainingTime--;
            currentTime++;

            // If the process is completed
            if (currentProcess.remainingTime == 0) {
                completed++;
                currentProcess.completionTime = currentTime;
                currentProcess.turnaroundTime = currentProcess.completionTime - currentProcess.arrivalTime;
                currentProcess.waitingTime = currentProcess.turnaroundTime - currentProcess.burstTime;

                totalTAT += currentProcess.turnaroundTime;
                totalWT += currentProcess.waitingTime;
            }
        }

        printResults(processes, n, totalTAT, totalWT);
    }

    // Non-Preemptive Priority Scheduling
    public static void nonPreemptivePriorityScheduling(ArrayList<Process> processes, int n) {
        int currentTime = 0, completed = 0;
        double totalTAT = 0, totalWT = 0;

        while (completed < n) {
            // Find the highest-priority process available at the current time
            Process currentProcess = null;
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && p.remainingTime > 0) {
                    if (currentProcess == null || p.priority < currentProcess.priority) {
                        currentProcess = p;
                    }
                }
            }

            if (currentProcess == null) {
                currentTime++;
                continue;
            }

            // Execute the process to completion
            currentTime += currentProcess.remainingTime;
            currentProcess.remainingTime = 0;
            currentProcess.completionTime = currentTime;

            // Calculate TAT and WT
```

```
            currentProcess.turnaroundTime = currentProcess.completionTime - currentProcess.arrivalTime;
            currentProcess.waitingTime = currentProcess.turnaroundTime - currentProcess.burstTime;

            totalTAT += currentProcess.turnaroundTime;
            totalWT += currentProcess.waitingTime;

            completed++;
        }

        printResults(processes, n, totalTAT, totalWT);
    }

    // Print the results
    public static void printResults(ArrayList<Process> processes, int n, double totalTAT, double totalWT) {
        System.out.println("\nPriority Scheduling Results:");
        System.out.println("Process\tArrival Time\tBurst Time\tPriority\tCompletion Time\tTurnaround
Time\tWaiting Time");

        for (Process p : processes) {
            System.out.println("P" + p.id + "\t\t" + p.arrivalTime + "\t\t" + p.burstTime + "\t\t" + p.priority +
                    "\t\t" + p.completionTime + "\t\t" + p.turnaroundTime + "\t\t" + p.waitingTime);
        }

        System.out.println("\nAverage Turnaround Time: " + (totalTAT / n));
        System.out.println("Average Waiting Time: " + (totalWT / n));
    }
}
```

7. Write a program to compute the Turnaround Time (TAT) and Waiting Time (WT) using the Round Robin Scheduling. (enter suitable number of processes, CPU burst, and Arrival Time)

```
import java.util.ArrayList;
import java.util.Scanner;

public class RoundRobinScheduling {

    static class Process {
        int id, arrivalTime, burstTime, remainingTime, completionTime, turnaroundTime, waitingTime;

        public Process(int id, int arrivalTime, int burstTime) {
            this.id = id;
            this.arrivalTime = arrivalTime;
            this.burstTime = burstTime;
            this.remainingTime = burstTime; // Initially, remaining time is the burst time
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input number of processes
        System.out.print("Enter the number of processes: ");
        int n = sc.nextInt();

        ArrayList<Process> processes = new ArrayList<>();

        // Input process details
        for (int i = 1; i <= n; i++) {
            System.out.print("Enter arrival time of Process " + i + ": ");
            int arrivalTime = sc.nextInt();
            System.out.print("Enter burst time of Process " + i + ": ");
            int burstTime = sc.nextInt();
            processes.add(new Process(i, arrivalTime, burstTime));
        }

        // Input time quantum
        System.out.print("Enter the time quantum: ");
        int timeQuantum = sc.nextInt();

        // Call the Round Robin scheduling function
        roundRobinScheduling(processes, n, timeQuantum);
    }

    public static void roundRobinScheduling(ArrayList<Process> processes, int n, int timeQuantum) {
        int currentTime = 0, completed = 0;
        double totalTAT = 0, totalWT = 0;

        ArrayList<Process> queue = new ArrayList<>(); // Process queue
        int index = 0; // Index for adding processes to the queue

        while (completed < n) {
            // Add processes that have arrived at the current time to the queue
            for (Process p : processes) {
                if (p.arrivalTime <= currentTime && !queue.contains(p) && p.remainingTime > 0) {
                    queue.add(p);
                }
            }

            // If queue is empty, advance time
            if (queue.isEmpty()) {
                currentTime++;
                continue;
            }
            // Get the next process from the queue
            Process currentProcess = queue.remove(0);

            // Execute the process for the time quantum or until completion
            int executionTime = Math.min(timeQuantum, currentProcess.remainingTime);
            currentTime += executionTime;
            currentProcess.remainingTime -= executionTime;

            // If the process is completed
            if (currentProcess.remainingTime == 0) {
                completed++;
                currentProcess.completionTime = currentTime;
                currentProcess.turnaroundTime = currentProcess.completionTime - currentProcess.arrivalTime;
                currentProcess.waitingTime = currentProcess.turnaroundTime - currentProcess.burstTime;

                totalTAT += currentProcess.turnaroundTime;
                totalWT += currentProcess.waitingTime;
            } else {
                // Re-add the process to the queue if not completed
                queue.add(currentProcess);
            }
        }

        // Print results
        printResults(processes, n, totalTAT, totalWT);
    }

    // Print the results
    public static void printResults(ArrayList<Process> processes, int n, double totalTAT, double totalWT) {
        System.out.println("\nRound Robin Scheduling Results:");
        System.out.println("Process\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting
Time");

        for (Process p : processes) {
            System.out.println("P" + p.id + "\t\t" + p.arrivalTime + "\t\t" + p.burstTime + "\t\t" +
                    p.completionTime + "\t\t" + p.turnaroundTime + "\t\t" + p.waitingTime);
        }

        System.out.println("\nAverage Turnaround Time: " + (totalTAT / n));
        System.out.println("Average Waiting Time: " + (totalWT / n));
    }
}
```

8. Write a program to demonstrate any 5 system calls.

```java
import java.io.File;
import java.io.FileWriter;
import java.util.Scanner;

public class SystemCallsDemo {
    public static void main(String[] args) {
        try {
            // 1. Create a file
            File file = new File("example.txt");
            if (file.createNewFile()) {
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }

            // 2. Write to the file
            FileWriter writer = new FileWriter(file);
            writer.write("Hello, this is a demonstration of system calls in Java.");
            writer.close();
            System.out.println("Data written to file.");

            // 3. Read from the file
            Scanner reader = new Scanner(file);
            System.out.println("File content:");
            while (reader.hasNextLine()) {
                System.out.println(reader.nextLine());
            }
            reader.close();

            // 4. Get file properties
            System.out.println("File Path: " + file.getAbsolutePath());
            System.out.println("File Size: " + file.length() + " bytes");

            // 5. Delete the file
            if (file.delete()) {
                System.out.println("File deleted successfully.");
            } else {
                System.out.println("Failed to delete the file.");
            }
        } catch (Exception e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}


import java.io.*;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SystemCallDemo {
    public static void main(String[] args) {
        try {
            // 1. Open: Create a file and open it for writing
            File file = new File("example.txt");
            if (!file.exists()) {
                file.createNewFile();
                System.out.println("File created: " + file.getName());
            } else {
                System.out.println("File already exists.");
            }

            // 2. Write: Write some data to the file
            FileWriter writer = new FileWriter(file);
            writer.write("This is an example demonstrating system calls.\n");
            writer.write("Java can simulate system calls like open, read, write, and close.\n");
            writer.close();
            System.out.println("Data written to the file.");

            // 3. Read: Read the data back from the file
            BufferedReader reader = new BufferedReader(new FileReader(file));
            System.out.println("\nContents of the file:");
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();

            // 4. Close: The writer and reader are explicitly closed after use

            // 5. Fork (Simulated using Threads)
            System.out.println("\nSimulating 'fork' using threads:");
            ExecutorService executorService = Executors.newFixedThreadPool(2);

            Runnable task1 = () -> System.out.println("Child process 1: Running task.");
            Runnable task2 = () -> System.out.println("Child process 2: Running task.");

            executorService.execute(task1);
            executorService.execute(task2);

            executorService.shutdown();

        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

9. Write a program to Implement multithreading for Matrix Operations using Pthreads (any one operation).

```
class MatrixMultiplier implements Runnable {
    private final int[][] matA;
    private final int[][] matB;
    private final int[][] result;
    private final int row; // Row of the result matrix to compute

    public MatrixMultiplier(int[][] matA, int[][] matB, int[][] result, int row) {
        this.matA = matA;
        this.matB = matB;
        this.result = result;
        this.row = row;
    }

    @Override
    public void run() {
        int colsB = matB[0].length;
        int colsA = matA[0].length;
        for (int j = 0; j < colsB; j++) { // For each column in matB
            result[row][j] = 0; // Initialize to zero
            for (int k = 0; k < colsA; k++) { // Perform dot product
                result[row][j] += matA[row][k] * matB[k][j];
            }
        }
    }
}

public class MatrixMultiplicationMultithreading {
    public static void main(String[] args) {
        // Define matrices
        int[][] matA = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };
        int[][] matB = {
            {1, 4, 7},
            {2, 5, 8},
            {3, 6, 9}
        };

        int rowsA = matA.length;
        int colsB = matB[0].length;
        int[][] result = new int[rowsA][colsB];

        Thread[] threads = new Thread[rowsA];

        // Create and start threads for each row of the result matrix
        for (int i = 0; i < rowsA; i++) {
            threads[i] = new Thread(new MatrixMultiplier(matA, matB, result, i));
            threads[i].start();
        }

        // Wait for all threads to complete
        for (int i = 0; i < rowsA; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Print the resulting matrix
        System.out.println("Resultant Matrix:");
        for (int[] row : result) {
            for (int elem : row) {
                System.out.print(elem + " ");
            }
            System.out.println();
        }
    }
}
```

10. Write a program to check whether a given system is in a safe state or not using Banker's Deadlock Avoidance algorithm (assume suitable data).

```
import java.util.Scanner;

public class BankersAlgorithm {
    private int numberOfProcesses, numberOfResources;
    private int[][] max, allocation, need;
    private int[] available;

    public void initialize() {
        Scanner sc = new Scanner(System.in);

        // Input number of processes and resources
        System.out.print("Enter the number of processes: ");
        numberOfProcesses = sc.nextInt();
        System.out.print("Enter the number of resources: ");
        numberOfResources = sc.nextInt();

        max = new int[numberOfProcesses][numberOfResources];
        allocation = new int[numberOfProcesses][numberOfResources];
        need = new int[numberOfProcesses][numberOfResources];
        available = new int[numberOfResources];

        // Input Max Matrix
        System.out.println("Enter the Max Matrix:");
        for (int i = 0; i < numberOfProcesses; i++) {
            for (int j = 0; j < numberOfResources; j++) {
                max[i][j] = sc.nextInt();
            }
        }

        // Input Allocation Matrix
        System.out.println("Enter the Allocation Matrix:");
        for (int i = 0; i < numberOfProcesses; i++) {
            for (int j = 0; j < numberOfResources; j++) {
                allocation[i][j] = sc.nextInt();
            }
        }

        // Input Available Vector
        System.out.println("Enter the Available Vector:");
        for (int j = 0; j < numberOfResources; j++) {
            available[j] = sc.nextInt();
        }

        // Calculate the Need Matrix
        for (int i = 0; i < numberOfProcesses; i++) {
            for (int j = 0; j < numberOfResources; j++) {
                need[i][j] = max[i][j] - allocation[i][j];
            }
        }
    }

    public boolean isSafeState() {
        boolean[] finished = new boolean[numberOfProcesses];
        int[] work = available.clone();
        int[] safeSequence = new int[numberOfProcesses];
        int count = 0;

        while (count < numberOfProcesses) {
            boolean found = false;
            for (int i = 0; i < numberOfProcesses; i++) {
                if (!finished[i]) {
                    int j;
                    for (j = 0; j < numberOfResources; j++) {
                        if (need[i][j] > work[j]) {
                            break;
                        }
                    }

                    if (j == numberOfResources) {
                        for (int k = 0; k < numberOfResources; k++) {
                            work[k] += allocation[i][k];
                        }
                        safeSequence[count++] = i;
                        finished[i] = true;
                        found = true;
                    }
                }
            }

            if (!found) {
                System.out.println("The system is not in a safe state.");
                return false;
            }
        }

        System.out.println("The system is in a safe state.");
        System.out.println("Safe sequence: ");
        for (int i = 0; i < numberOfProcesses; i++) {
            System.out.print("P" + safeSequence[i] + " ");
        }
        System.out.println();

        return true;
    }

    public static void main(String[] args) {
        BankersAlgorithm ba = new BankersAlgorithm();
        ba.initialize();
        ba.isSafeState();
    }
}
```

```
\* Enter the number of processes: 5
Enter the number of resources: 3
Enter the Max Matrix:
7 5 3
3 2 2
9 0 2
```

```
2 2 2
4 3 3
 Enter the Allocation Matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter the Available Vector:
3 3 2

The system is in a safe state. Safe sequence: P1 P3 P4 P0 P2 */
```

11. Write a program to calculate the number of page faults for a reference string (input any suitable reference string) using First In First Out (FIFO) page replacement algorithms.

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class FIFOPageReplacement {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the number of frames
        System.out.print("Enter the number of frames: ");
        int frames = sc.nextInt();

        // Input the reference string
        System.out.print("Enter the length of the reference string: ");
        int n = sc.nextInt();
        int[] referenceString = new int[n];

        System.out.println("Enter the reference string:");
        for (int i = 0; i < n; i++) {
            referenceString[i] = sc.nextInt();
        }

        // FIFO Page Replacement
        Queue<Integer> pageQueue = new LinkedList<>();
        int pageFaults = 0;

        for (int page : referenceString) {
            if (!pageQueue.contains(page)) {
                // If the page is not in the queue, a page fault occurs
                if (pageQueue.size() == frames) {
```

```java
                pageQueue.poll(); // Remove the oldest page (FIFO)
            }
            pageQueue.add(page); // Add the new page
            pageFaults++;
        }
    }

    System.out.println("Total Page Faults: " + pageFaults);
    sc.close();
  }
}

//3 12
//7 0 1 2 0 3 0 4 2 3 0 3
//page fault 10
```

12. Write a program to calculate the number of page faults for a reference string (input any suitable reference string) using the Least Recently Used (LRU) page replacement algorithms.

```java
import java.util.*;

public class LRUPageReplacement {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    // Input the number of frames
    System.out.print("Enter the number of frames: ");
    int frames = sc.nextInt();

    // Input the reference string
    System.out.print("Enter the length of the reference string: ");
    int n = sc.nextInt();
    int[] referenceString = new int[n];

    System.out.println("Enter the reference string:");
    for (int i = 0; i < n; i++) {
      referenceString[i] = sc.nextInt();
    }

    // LRU Page Replacement Algorithm
    List<Integer> pageFrames = new ArrayList<>();
    int pageFaults = 0;

    for (int page : referenceString) {
      if (!pageFrames.contains(page)) {
        // Page Fault
```

```
                if (pageFrames.size() == frames) {
                    // Remove the least recently used page
                    pageFrames.remove(0);
                }
                pageFrames.add(page);
                pageFaults++;
            } else {
                // If the page is in memory, move it to the most recently used position
                pageFrames.remove((Integer) page);
                pageFrames.add(page);
            }
        }

        System.out.println("Total Page Faults: " + pageFaults);
        sc.close();
    }
}

//Page fault 9
```

13. Write a program to calculate the number of page faults for a reference string (input any suitable reference string) using the Optimal page replacement algorithms.

```
import java.util.*;

public class OptimalPageReplacement {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        // Input the number of frames
        System.out.print("Enter the number of frames: ");
        int frames = sc.nextInt();

        // Input the reference string
        System.out.print("Enter the length of the reference string: ");
        int n = sc.nextInt();
        int[] referenceString = new int[n];

        System.out.println("Enter the reference string:");
        for (int i = 0; i < n; i++) {
            referenceString[i] = sc.nextInt();
        }

        // Optimal Page Replacement Algorithm
        List<Integer> pageFrames = new ArrayList<>();
        int pageFaults = 0;
        for (int i = 0; i < n; i++) {
            int page = referenceString[i];
            if (!pageFrames.contains(page)) {
                // Page Fault
                if (pageFrames.size() == frames) {
                    // Find the page to replace using the Optimal Algorithm
                    int farthestIndex = -1;
                    int pageToReplace = -1;
                    for (int p : pageFrames) {
                        int nextUse = Integer.MAX_VALUE;
                        for (int j = i + 1; j < n; j++) {
                            if (referenceString[j] == p) {
                                nextUse = j;
                                break;
                            }
                        }
                        if (nextUse > farthestIndex) {
                            farthestIndex = nextUse;
                            pageToReplace = p;
                        }
                    }
                    pageFrames.remove((Integer) pageToReplace);
                }
                pageFrames.add(page);
                pageFaults++;
            }
        }

        System.out.println("Total Page Faults: " + pageFaults);
        sc.close();
    }
}

//page fault 7
```