LAB ASSIGNMENT – 4

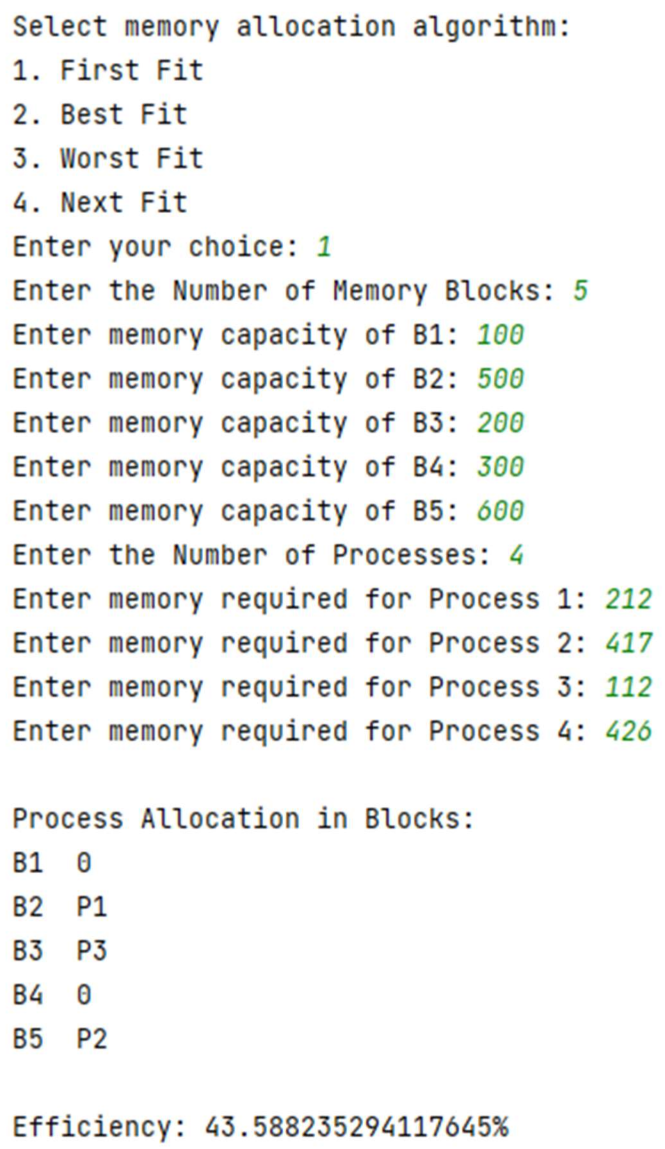
Memory Management Strategies

PROGRAM –

```java
import java.util.Arrays;
import java.util.Scanner;

public class MemoryManagement {

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

        // Display menu for memory allocation algorithms
        System.out.println("Select memory allocation algorithm:");
        System.out.println("1. First Fit\n2. Best Fit\n3. Worst Fit\n4. Next Fit");
        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();

        // Input number of memory blocks and their capacities
        System.out.print("Enter the Number of Memory Blocks: ");
        int blockNo = scanner.nextInt();
        int[] blocks = new int[blockNo];
        for (int i = 0; i < blockNo; i++) {
            System.out.print("Enter memory capacity of B" + (i + 1) + ": ");
            blocks[i] = scanner.nextInt();
        }

        // Input number of processes and their memory requirements
        System.out.print("Enter the Number of Processes: ");
        int processNo = scanner.nextInt();
        int[] processes = new int[processNo];
        for (int i = 0; i < processNo; i++) {
            System.out.print("Enter memory required for Process " + (i + 1) + ": ");
            processes[i] = scanner.nextInt();
        }

        // Array to store which process is allocated to which block (-1 indicates no
allocation)
        int[] display = new int[blockNo];
        Arrays.fill(display, -1);

        // Switch case to handle the selected memory allocation strategy
        switch (choice) {
            case 1 -> allocateMemory(processes, blocks, display, "First Fit");
            case 2 -> allocateMemory(processes, blocks, display, "Best Fit");
            case 3 -> allocateMemory(processes, blocks, display, "Worst Fit");
            case 4 -> nextFitAllocation(processes, blocks, display);  // Next Fit has a
separate method
            default -> System.out.println("Invalid choice. Exiting program.");
        }
        scanner.close();
    }

    // Method to allocate memory based on the chosen strategy (First, Best, or Worst Fit)
    private static void allocateMemory(int[] processes, int[] blocks, int[] display,
String strategy) {
        int blockNo = blocks.length, sum = 0, total = Arrays.stream(blocks).sum(); //
Total memory sum calculation
        boolean[] flag = new boolean[blockNo]; // Flag to check if a block is already
allocated
```

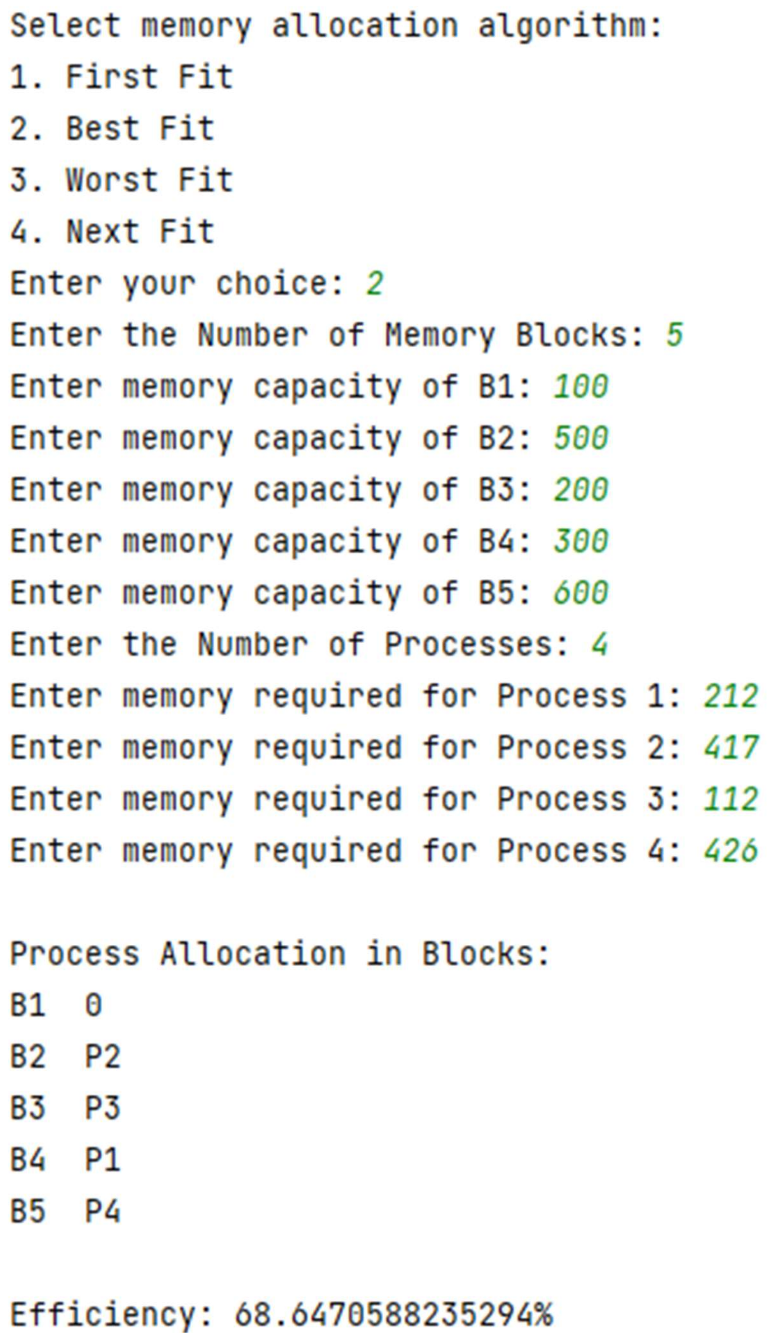
```java
        // Loop through processes to allocate them to memory blocks based on the strategy
        for (int i = 0; i < processes.length; i++) {
            int idx = switch (strategy) {
                case "First Fit" -> findBlockFirstFit(blocks, flag, processes[i]);
                case "Best Fit" -> findBlockBestFit(blocks, flag, processes[i]);
                case "Worst Fit" -> findBlockWorstFit(blocks, flag, processes[i]);
                default -> -1;
            };
            if (idx != -1) {
                flag[idx] = true; // Mark block as allocated
                sum += processes[i]; // Add process memory to sum
                display[idx] = i; // Store allocated process index in the display array
            }
        }
        printAllocation(display, blocks, sum, total); // Print allocation and efficiency
    }

    // First Fit: Find the first block that can fit the process
    private static int findBlockFirstFit(int[] blocks, boolean[] flag, int size) {
        for (int j = 0; j < blocks.length; j++)
            if (!flag[j] && blocks[j] >= size) return j; // Return the first available
block index
        return -1; // Return -1 if no suitable block found
    }

    // Best Fit: Find the smallest block that can fit the process
    private static int findBlockBestFit(int[] blocks, boolean[] flag, int size) {
        int idx = -1;
        for (int j = 0; j < blocks.length; j++)
            if (!flag[j] && blocks[j] >= size && (idx == -1 || blocks[j] < blocks[idx]))
idx = j; // Choose the smallest available block
        return idx;
    }

    // Worst Fit: Find the largest block that can fit the process
    private static int findBlockWorstFit(int[] blocks, boolean[] flag, int size) {
        int idx = -1;
        for (int j = 0; j < blocks.length; j++)
            if (!flag[j] && blocks[j] >= size && (idx == -1 || blocks[j] > blocks[idx]))
idx = j; // Choose the largest available block
        return idx;
    }

    // Next Fit: Allocate memory sequentially, starting from where the last allocation
ended
    private static void nextFitAllocation(int[] processes, int[] blocks, int[] display) {
        int total = Arrays.stream(blocks).sum(), sum = 0, idx = 0; // Initialize sum and
start index
        boolean[] flag = new boolean[blocks.length]; // To track allocated blocks

        for (int i = 0;i<processes.length;i++) {
            boolean allocated = false; // Track if process is allocated
            for (int j = 0; j < blocks.length; j++) { // Loop through blocks starting
from last index
                int blockIdx = (idx + j) % blocks.length; // Wrap around using modulus
operator
                if (!flag[blockIdx] && blocks[blockIdx] >= processes[i]) {
                    sum += processes[i]; // Add process memory to sum
                    blocks[blockIdx] -= processes[i]; // Reduce block size
                    display[blockIdx] = i; // Store allocated process
                    flag[blockIdx] = true; // Mark block as allocated
                    idx = (blockIdx + 1) % blocks.length; // Update the last allocated
block index
                    allocated = true;
                    break;
                }
            }
            if (!allocated) {
                System.out.println("No suitable block found for process with size: " +
processes[i]);
            }
        }
        printAllocation(display, blocks, sum, total); // Print allocation and efficiency
    }

    // Print the final allocation and calculate the efficiency
    private static void printAllocation(int[] display, int[] blocks, int sum, int total)
{
        System.out.println("\nProcess Allocation in Blocks:");
        for (int i = 0; i < display.length; i++) {
            System.out.println("B" + (i + 1) + "\t" + (display[i] == -1 ? "0" : "P" +
(display[i] + 1))); // Print allocated processes
        }
        System.out.println("\nEfficiency: " + (sum * 100.0) / total + "%"); // Print
efficiency as a percentage
    }
}
```

OUTPUT



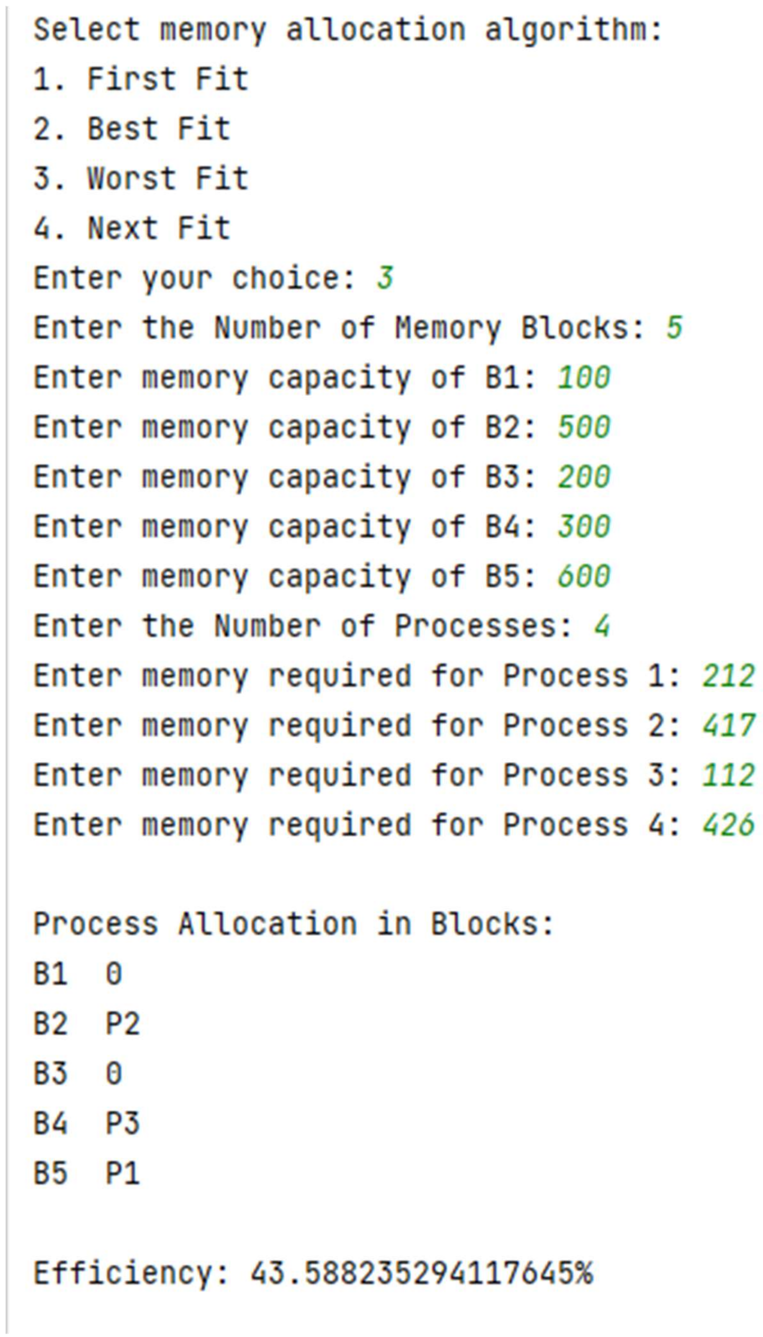
```
Select memory allocation algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
Enter your choice: 1
Enter the Number of Memory Blocks: 5
Enter memory capacity of B1: 100
Enter memory capacity of B2: 500
Enter memory capacity of B3: 200
Enter memory capacity of B4: 300
Enter memory capacity of B5: 600
Enter the Number of Processes: 4
Enter memory required for Process 1: 212
Enter memory required for Process 2: 417
Enter memory required for Process 3: 112
Enter memory required for Process 4: 426

Process Allocation in Blocks:
B1  0
B2  P1
B3  P3
B4  0
B5  P2

Efficiency: 43.588235294117645%
```

Output 1 – First – Fit



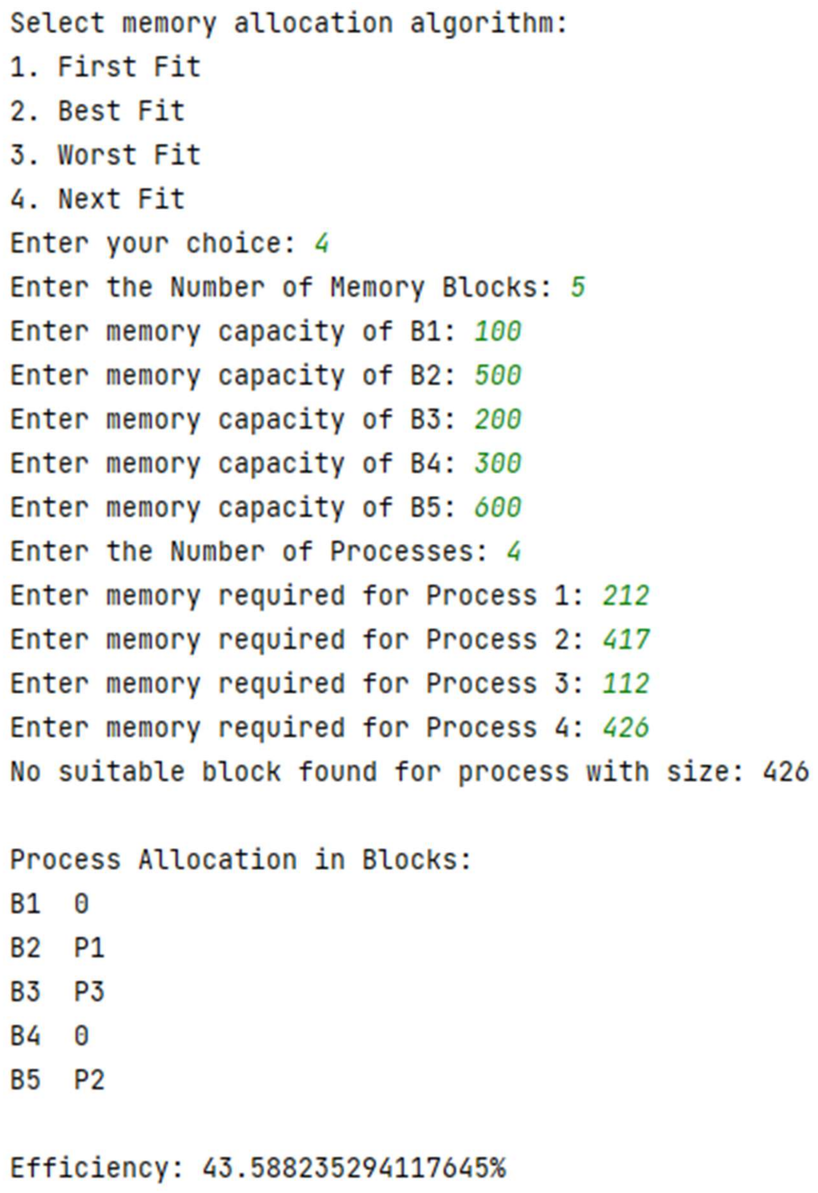
```
Select memory allocation algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
Enter your choice: 2
Enter the Number of Memory Blocks: 5
Enter memory capacity of B1: 100
Enter memory capacity of B2: 500
Enter memory capacity of B3: 200
Enter memory capacity of B4: 300
Enter memory capacity of B5: 600
Enter the Number of Processes: 4
Enter memory required for Process 1: 212
Enter memory required for Process 2: 417
Enter memory required for Process 3: 112
Enter memory required for Process 4: 426

Process Allocation in Blocks:
B1  0
B2  P2
B3  P3
B4  P1
B5  P4

Efficiency: 68.6470588235294%
```

Output 2 – Best – Fit



```
Select memory allocation algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
Enter your choice: 3
Enter the Number of Memory Blocks: 5
Enter memory capacity of B1: 100
Enter memory capacity of B2: 500
Enter memory capacity of B3: 200
Enter memory capacity of B4: 300
Enter memory capacity of B5: 600
Enter the Number of Processes: 4
Enter memory required for Process 1: 212
Enter memory required for Process 2: 417
Enter memory required for Process 3: 112
Enter memory required for Process 4: 426

Process Allocation in Blocks:
B1  0
B2  P2
B3  0
B4  P3
B5  P1

Efficiency: 43.588235294117645%
```

Output 3 – Worst – Fit



```
Select memory allocation algorithm:
1. First Fit
2. Best Fit
3. Worst Fit
4. Next Fit
Enter your choice: 4
Enter the Number of Memory Blocks: 5
Enter memory capacity of B1: 100
Enter memory capacity of B2: 500
Enter memory capacity of B3: 200
Enter memory capacity of B4: 300
Enter memory capacity of B5: 600
Enter the Number of Processes: 4
Enter memory required for Process 1: 212
Enter memory required for Process 2: 417
Enter memory required for Process 3: 112
Enter memory required for Process 4: 426
No suitable block found for process with size: 426

Process Allocation in Blocks:
B1  0
B2  P1
B3  P3
B4  0
B5  P2

Efficiency: 43.588235294117645%
```

Output 4 – Next - Fit