NAME: Meith Navlakha
SAP: 60004190068
POA Practical

EXPERIMENT 3

AIM: To implement Best Fit Memory Allocation

THEORY:

Memory allocation is the process of assigning blocks of memory to process on request. Typically the allocator receives memory from the operating system and it must assign it to the appropriate process to satisfy the request. It must also make any returned blocks available for reuse. There are many common ways to perform this, with different strengths and weaknesses.

In this experiement we will implement **Best Fit Memory Allocation**.

Best-Fit Allocation

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the free block that gives the least internal fragmentation. Allocates the process to that memory block.

It is used for Partition Memory allocation where the memory is pre partitioned into equal or unequal blocks.

Advantages:

- Memory Efficient, as the operating system allocates the job to the minimum possible space in the memory.
- Minimizes internal Fragmentation
- Memory utilization is much better than first fit as it searches the smallest free partition first available

Disadvantages:

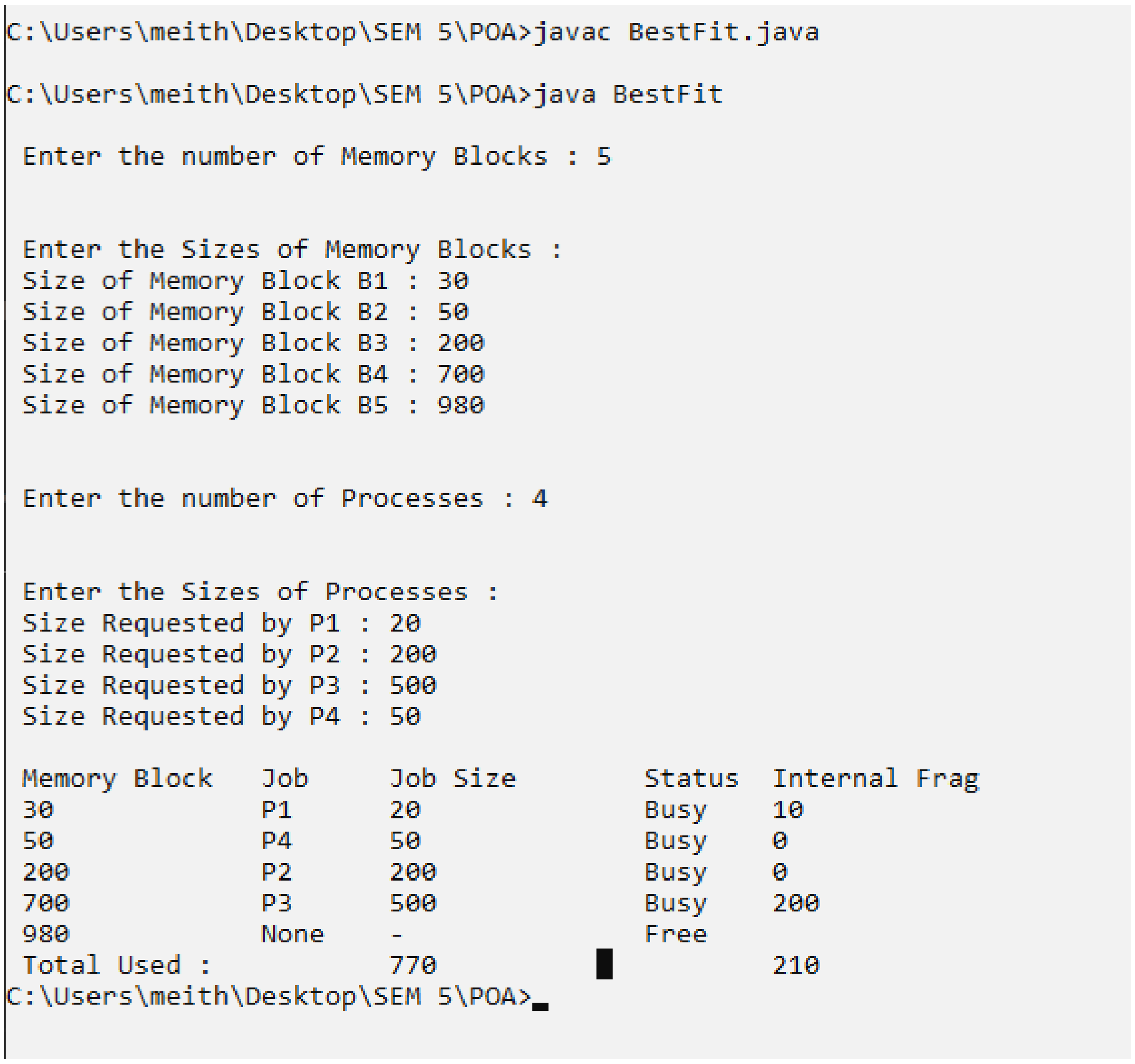
- It is slower because it scans through the entire memory list every time and tries to find out the smallest block big enough to hold the process.
- Leads to Internal fragmentation
- Cannot be used for where fast execution of processes is expected.

NAME: Meith Navlakha
SAP: 60004190068
POA Practical

CODE:

```
import java.util.*;
import java.util.Arrays;
import java.lang.Math;
import java.lang.Integer;

public class BestFit {


public static int findBlock(int curr_process, int[] curr_mem, int[] busy_mem, int
num_p, int num_b) {

    int index = -1;
    int lowest = Integer.MAX_VALUE;

    for (int i = 0; i < num_b; i++) {
      //Empty
      if (busy_mem[i] == 0 && (curr_mem[i] - curr_process) >= 0 && lowest >
(curr_mem[i] - curr_process)) {
        lowest = curr_mem[i] - curr_process;
        index = i;
      }
    }

    return index;
  }



  public static void bestFit(int[] process, int[] memory, int num_p, int num_b) {

    int[] busy_mem = new int[num_b];
    int[] internal_fragment = new int[num_b];
    int[] exe_process = new int[num_b];
    // int[]

    System.out.print("\n Memory Block\tJob\tJob Size\tStatus\tInternal Frag");
    for (int i = 0; i < num_p; i++) {
      int index = findBlock(process[i], memory, busy_mem, num_p, num_b);

      if (index == -1) {
        // No Space
      } else{
        busy_mem[index] = 1;
        exe_process[index] = i;
        internal_fragment[index] = memory[index] - process[i];
      }

    } //closes for

    // Display

    int total = 0, total_frag = 0;
    for (int i = 0; i < num_b; i++) {

      if (busy_mem[i] == 1) {
        System.out.print("\n " + memory[i] + "\t\tP" + (exe_process[i] + 1) + "\t" +
(process[exe_process[i]]) + "\t\tBusy\t" + internal_fragment[i]);
        total += process[exe_process[i]];
        total_frag += internal_fragment[i];
      } else {
        System.out.print("\n " + memory[i] + "\t\tNone" + "\t-\t" + "\tFree");

      }

    }

    System.out.print("\n Total Used : \t\t" + total + "\t\t\t" + total_frag);

  }
  public static void main(String args[]) {

    Scanner sc = new Scanner(System.in);

    System.out.print("\n Enter the number of Memory Blocks : ");
    int num_block = sc.nextInt();

    int[] memory = new int[num_block];

    System.out.print("\n\n Enter the Sizes of Memory Blocks : \n");
    for (int i = 0; i < num_block; i++) {
      System.out.print(" Size of Memory Block B" + (i + 1) + " : ");
      memory[i] = sc.nextInt();
    }

    System.out.print("\n\n Enter the number of Processes : ");
    int num_process = sc.nextInt();
    int[] process = new int[num_process];

    System.out.print("\n\n Enter the Sizes of Processes : \n");
    for (int i = 0; i < num_process; i++) {
      System.out.print(" Size Requested by P" + (i + 1) + " : ");
      process[i] = sc.nextInt();
    }

    bestFit(process, memory, num_process, num_block);

  }

}
```

OUTPUT:



```
C:\Users\meith\Desktop\SEM 5\POA>javac BestFit.java

C:\Users\meith\Desktop\SEM 5\POA>java BestFit

 Enter the number of Memory Blocks : 5


 Enter the Sizes of Memory Blocks :
 Size of Memory Block B1 : 30
 Size of Memory Block B2 : 50
 Size of Memory Block B3 : 200
 Size of Memory Block B4 : 700
 Size of Memory Block B5 : 980


 Enter the number of Processes : 4


 Enter the Sizes of Processes :
 Size Requested by P1 : 20
 Size Requested by P2 : 200
 Size Requested by P3 : 500
 Size Requested by P4 : 50

 Memory Block   Job     Job Size        Status  Internal Frag
 30             P1      20              Busy    10
 50             P4      50              Busy    0
 200            P2      200             Busy    0
 700            P3      500             Busy    200
 980            None    -               Free
 Total Used :           770                     210
C:\Users\meith\Desktop\SEM 5\POA>
```

CONCLUSION:

In this experiment I implemented Best Fit Memory Allocation in Java. Here, the OS allocates the free memory block to the process with the least internal fragmentation. The advantages of Best Fit are efficient memory utilization as it allocates the most appropriate memory block. But it is very slow as it has to scan through the entire free/allotted block list.