Q2. Design suitable data structures and implement Pass-I of a two-pass macro-processor.

```
import java.io.*;
import java.util.*;

class MacroProcessorPass1{
  static class MNTEntry {
    String name;
    int mdtIndex;

    MNTEntry(String name, int mdtIndex) {
      this.name = name;
      this.mdtIndex = mdtIndex;
    }
  }

  public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    List<MNTEntry> MNT = new ArrayList<>();
    List<String> MDT = new ArrayList<>();

    System.out.println("Enter number of lines in input program:");
    int n = Integer.parseInt(br.readLine());

    String[] program = new String[n];
    System.out.println("Enter the program (line by line):");
```

```
for (int i = 0; i < n; i++) {
  program[i] = br.readLine();
}

boolean inMacro = false;
int mdtIndex = 0;

for (int i = 0; i < n; i++) {
  String line = program[i].trim();

  if (line.equalsIgnoreCase("MACRO")) {
    inMacro = true;
    String defLine = program[++i].trim(); // Macro header line
    String[] parts = defLine.split("\\s+");
    String macroName = parts[0];
    MNT.add(new MNTEntry(macroName, mdtIndex));
    continue;
  }

  if (line.equalsIgnoreCase("MEND")) {
    MDT.add("MEND");
    mdtIndex++;
    inMacro = false;
    continue;
  }

  if (inMacro) {
    MDT.add(line);
```

```
            mdtIndex++;
          }
        }

        System.out.println("\n----- MNT (Macro Name Table) -----");
        System.out.printf("%-10s %-10s\n", "Name", "MDT Index");
        for (MNTEntry e : MNT) {
          System.out.printf("%-10s %-10d\n", e.name, e.mdtIndex);
        }

        System.out.println("\n----- MDT (Macro Definition Table) -----");
        for (int i = 0; i < MDT.size(); i++) {
          System.out.println(i + "  " + MDT.get(i));
        }
      }
    }

    //8
    //MACRO
    //INCR &A,&B
    //MOVER AREG,&A
    //ADD AREG,&B
    //      MOVEM AREG,RESULT
    //MEND
    //      START
    //      INCR DATA1,DATA2
    //END
```

Q11.Write a program to simulate Memory placement strategies – best fit, and worst fit.

```
import java.util.Scanner;
public class MemoryPlacement {
  static void bestFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[] = new int[n]; // Stores block assigned to each process

    for (int i = 0; i < n; i++)
      allocation[i] = -1;

    for (int i = 0; i < n; i++) {
      int bestIdx = -1;
      for (int j = 0; j < m; j++) {
        if (blockSize[j] >= processSize[i]) {
          if (bestIdx == -1 || blockSize[j] < blockSize[bestIdx])
            bestIdx = j;
        }
      }
      if (bestIdx != -1) {
        allocation[i] = bestIdx;
        blockSize[bestIdx] -= processSize[i];
      }
    }
    System.out.println("\n===== Best Fit Allocation =====");
    System.out.println("Process No.\tProcess Size\tBlock No.");
    for (int i = 0; i < n; i++) {
      System.out.print((i + 1) + "\t\t" + processSize[i] + "\t\t");
      if (allocation[i] != -1)
```

```java
            System.out.println(allocation[i] + 1);
        else
            System.out.println("Not Allocated");
    }
}
static void worstFit(int blockSize[], int m, int processSize[], int n) {
    int allocation[] = new int[n]; // Stores block assigned to each process

    // Initially, no process is allocated
    for (int i = 0; i < n; i++)
        allocation[i] = -1;

    for (int i = 0; i < n; i++) {
        int worstIdx = -1;
        for (int j = 0; j < m; j++) {
            if (blockSize[j] >= processSize[i]) {
                if (worstIdx == -1 || blockSize[j] > blockSize[worstIdx])
                    worstIdx = j;
            }
        }
        if (worstIdx != -1) {
            allocation[i] = worstIdx;
            blockSize[worstIdx] -= processSize[i];
        }
    }
    System.out.println("\n===== Worst Fit Allocation =====");
    System.out.println("Process No.\tProcess Size\tBlock No.");
    for (int i = 0; i < n; i++) {
```

```java
            System.out.print((i + 1) + "\t\t" + processSize[i] + "\t\t");
            if (allocation[i] != -1)
                System.out.println(allocation[i] + 1);
            else
                System.out.println("Not Allocated");
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter number of memory blocks: ");
        int m = sc.nextInt();
        int blockSize[] = new int[m];
        System.out.println("Enter size of each memory block:");
        for (int i = 0; i < m; i++)
            blockSize[i] = sc.nextInt();
        System.out.print("Enter number of processes: ");
        int n = sc.nextInt();
        int processSize[] = new int[n];

        System.out.println("Enter size of each process:");
        for (int i = 0; i < n; i++)
          processSize[i] = sc.nextInt();
        int[] blockCopy1 = blockSize.clone();
        int[] blockCopy2 = blockSize.clone();
        bestFit(blockCopy1, m, processSize, n);
        worstFit(blockCopy2, m, processSize, n);
    }
}
```

Q6. Program to simulate CPU Scheduling Algorithms: SJF.

```
import java.util.Scanner;

class Process {
   int pid;
   int burstTime;
   int waitingTime;
   int turnaroundTime;
   boolean completed;
}

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

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

      Process[] p = new Process[n];

      for (int i = 0; i < n; i++) {
         p[i] = new Process();
         p[i].pid = i + 1;
         System.out.print("Enter burst time for Process P" + p[i].pid + ": ");
         p[i].burstTime = sc.nextInt();
         p[i].completed = false;
      }
```

```java
int time = 0, completed = 0;
System.out.println("\nGantt Chart:");

while (completed < n) {
   int idx = -1;
   int minBT = Integer.MAX_VALUE;

   for (int i = 0; i < n; i++) {
      if (!p[i].completed && p[i].burstTime < minBT) {
         minBT = p[i].burstTime;
         idx = i;
      }
   }

   System.out.print("| P" + p[idx].pid + " ");
   p[idx].waitingTime = time;
   time += p[idx].burstTime;
   p[idx].turnaroundTime = time;
   p[idx].completed = true;
   completed++;
}
System.out.println("|");

System.out.println("\nSJF Scheduling Results:");
System.out.println("--------------------------------------------------");
System.out.println("Process\tBurst Time\tWaiting Time\tTurnaround Time");
System.out.println("--------------------------------------------------");
```

```
        float totalWT = 0, totalTAT = 0;

        for (int i = 0; i < n; i++) {
            System.out.println("P" + p[i].pid + "\t\t" + p[i].burstTime + "\t\t" +
                    p[i].waitingTime + "\t\t" + p[i].turnaroundTime);

            totalWT += p[i].waitingTime;
            totalTAT += p[i].turnaroundTime;
        }

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