1. Description of your code and architecture designs
   1. Tuple.java: data structure represents one input record.
   2. ProcessedTuple.java: data structure represents one output record (with computed order number and last date of order).
   3. Block.java: data structure represents one memory block containing at most 9 tuples.
   4. Reader.java: wrapper class for BufferedReader to read one or multiple blocks.
   5. Writer.java: wrapper class for BufferedWriter to write one or multiple blocks.
   6. Main.java: main part for TPMMS algorithm implementation.
      1. readAndSort: Read T1 block by block until buffer is full, and sort using quick sort algorithm. Then write to a file, and repeat until T1 is read. Then, repeat for T2.
      2. merge: Take sub-lists produced by sort phase as input files. Read the sorted lists based on memory limit and reserve one memory block as output. It will produce merged files (runs) every pass. The last pass will output only one file.
      3. processTuples: sequentially iterate the output file produced in merge phase. Merge the tuples with same client id into one and compute the number of orders and last date of order. Results are outputted into a txt file with tag “processed”.
   7. CreateData.java: randomly generate input records.
2. Highlights of your implementation features
3. Performance results and analysis

* Test Case 1: 2 files both with half million records
  + 5M
    - * 1. Sort

Disk I/O at sort phase: 222224

Sort Phase Execution Time: 356s

Number of Tuples 1000000

* + - * 1. Merge

Disk I/O at merge phase: 444864

Merge Phase Execution Time: 14770s

* + - * 1. Total
  + 10M

1. Sort

Disk I/O at sort phase: 222224

Sort Phase Execution Time: 299s

Number of Tuples 1000000

1. Total
   * 20M
2. Sort

Disk I/O at sort phase: 222224

Sort Phase Execution Time: 101s

Number of Tuples 1000000

1. Total

* Test Case 2: 2 files both with one million records
* 5M
  1. Sort
  2. Merge
* 10M
* 20M