# **Word Frequency Counter Project Notes**

**Architecture**

InputFile

Dictionary

Small / regular chunks

<word, frequency>

Large chunk

SwapDictionaryToFile

MergeSortTextFile

ProcessLines

FileReader

Large chunk

Small / regular chunks

Write Lines

Sort if temporary file (large file)

Temporary File (Unsorted)

OutputFile (Sorted)

BlockingCollection

**How to Run/Test the program:**

1. Copy the VS solution to a folder.
   1. *Dictionary/WordFrequencyCounter* – Main project
   2. *Dictionary/WordFrequencyCounterUnitTests* – Unit Test Project
2. Build the solution
3. Run the exe with command line arguments of input file and output filename.
   1. *WordFrequencyCounter.exe <fullpath\_inputfile.txt> <fullpath\_outputfile.txt>*
4. Log file is created in *$(ProjectDir)/Files/Log.txt*
5. Sample Input and Output files can be found in *Dictionary/WordFrequencyCounterUnitTests/WordFrequencyCounterUnitTests/TestFiles*

**Prerequisites:**

1. Visual studio installed.
2. Project is targeted for .Net 6

**Description:**

1. Design considerations
   1. For memory efficiency, at a time, a single line is read from the input file (*FileReader*), instead of reading entire file to memory.
   2. The design handles two scenarios:
      1. Smaller/medium files which can be accommodated in memory. Algorithm as follows:
         1. Line is read from the input file and the word-frequency key value pair is stored in dictionary.
         2. The dictionary is sorted by descending order of frequency and then by word. Then written to output file.
      2. Support for larger files that can’t be accommodated in memory. Algorithm as follows:
         1. A memory capacity is defined for Dictionary based on available memory.
         2. Line is read from input file and the word-frequency key value pair is stored in dictionary.
         3. When the size of the dictionary exceeds the memory capacity, the dictionary is written to temporary file.
         4. Then next batch of lines are read and word-frequency key value pair is stored in dictionary. If the dictionary size exceeds, it is then merged with the previous temporary file and the process continues.
         5. Finally, the remaining dictionary contents are merged with temporary file.
         6. The temporary file is sorted using merge sort and written to output file.
   3. Design Constraints:
      1. The work flow for smaller/medium files is straight forward with less complexity. But the support for larger files involves writing to the disk multiple times for calculating word-frequency counter and sorting.
2. There can be maximum 3 tasks running concurrently at a time.
   1. *FileReader*(..)
      1. Reads one line at a time from input file and writes it to BlockingCollection.
      2. Completes the BlockingCollection once file reading is complete.
   2. *ProcessLines*(…)
      1. Reads from the BlockingCollection, split line into words, counts the frequency of words and write the key value pair to dictionary.
      2. When the size of dictionary exceeds the specified capacity, it calls “*SwapDictionaryToFile*” task asynchronously to swap the dictionary to temp file.
      3. Clear the current dictionary and continue step i.
      4. Once reading from BlockingCollection is complete, sort the dictionary in descending order of frequency and then by word.
      5. Calls “SwapDictionaryToFile” method synchronously to write the dictionary to file.
   3. *SwapDictionaryToFile*(…)
      1. Only one instance of this task/method will be running at a time. This is to ensure we always have one swap file and have enough memory for *ProcessLines* task to continue processing.
      2. It handles the following scenarios
         1. For small/medium input files that can be accommodated in memory, it is called once and writes the sorted dictionary to output file.
         2. For larger files that can’t be accommodated in memory,
            1. Writes the dictionary to temporary file if no temporary file already exists.
            2. If temporary file already exists, merge the dictionary to existing temporary file.
         3. The memory capacity can be configured in such a way that there is sufficient memory left to copy the dictionary to file. For instance, if memory available for the program to use is 1 GB, and the input file is 2 GB, then the dictionary capacity should be 500 MB, so that when dictionary reaches 500 MB, we copy the 500 MB to a temporary file asynchronously (using *SwapDictionaryToFile* Task) and use the remaining 500 MB to continue filling the dictionary (*ProcessLines*).
3. *MergeSortTextFile*
   1. This method is used to sort the large file.
   2. Merge sort using temporary files is used since the full file cannot be accommodated in memory.
   3. Initially the bigger file is split into chunks and sorted and written to multiple small temporary files.
   4. Sorting Algorithm:
      1. The sorted temporary files are then merged using Priority Queue.
      2. Priority queue is configured to sort first based on descending order of frequency and then by word and then finally by temporary file number.
      3. First line from each temporary file is read at a time and added to priority queue.
      4. The top most element is removed from priority queue and written to output file.
      5. Then next line is read from the file from which the top element is found and pushed to priority queue
      6. The process continues until the priority queue is empty.
4. *FileLogger*
   1. *FileLogger* class is used to log lines to files. It is implemented as an interface since in future, the logger can be modified as per the requirements like console logging / Database logging.
   2. The logger is made thread safe.
5. Unit test cases
   1. Unit test cases are added to verify the project.