# Comp-2540: Lab Assignment 2: Word Count II (Sorting)

**Your Marks (For TA Use Only)**

| 5.1 | 5.2 | 5.3 | 5.4 | 5.5 | Total | Marked by |
|---|---|---|---|---|---|---|
| | | | | | | |
| 0.5 | 0.5 | 1 | 1 | 1 | | |

Check the following list before you submit:

- ✔ Have signed the document
- ✔ Have run the starter code and fill in the chart and table (5.1)
- ✔ Have predicted the runtime for selectSort (5.2).
- ✔ Have proved the complexity of mergeSort (5.3).
- ✔ Have submitted the `readTextGOOD` code on our site (5.4)
- ✔ Have tested your `countFAST` at our A2 site (5.5).

## 1 Due Date and Submission

The due date is Feb 3/5. To be submitted 10 minutes before the end of your lab section. Make sure that you add your signature below to reaffirm that you followed Senate Bylaws 31 (click here for the document). Some PDF readers may not support digital signature well. We recommend you to use Acrobat Reader that is free for downloading.

**Fill and Sign The Form**

**I**, your name Aakanksha Mandal , **verify that the submitted work is my own work.**

Date 29/01/202 Student Number 110189000 EMail ID manda

## 2 Objective

The aim of this assignment is to understand the performance difference between various sorting algorithms, and the performance of string concatenation. The hidden cost in string concatenation is also related to array resizing, a useful technique we will learn in other data structures.

## 3 The Problem Specification

The problem is similar to *A1: Word Count I*, i.e., to count the frequency of words in a text file. The output changes a little bit: instead of returning the top count, this time we return the frequency of the top 200-th most frequent word. Although the problem is similar, we start with sorting-based algorithms and compare different sorting approaches. We will try to reduce its overall run time, including the text reading part.

## 4 The algorithm

In this assignment, the counting method is based on sorting algorithms. A template of the algorithms is listed in Algorithm 1. The idea is to sort the tokens first, so that all the same tokens are located consecutively. For the sample input in A1, the sorted result can be:

```
a and are are are complicated constructing deficiencies deficiencies
design is is it it make make no no obvious obviously of one other
simple so so software that that the there there there to to two
way way ways time
```

**Input:** tokens: Array of string tokens
**Output:** The 200-th most frequent word
**begin**
  sort *tokens* alphabetically;
  **foreach** *t in tokens* **do**
    **if** *t is new* **then**
      add *t* in wordArray;
      update wordFreqArray;
    **end**
    **else**
      increment wordFreqArray until it is different
    **end**
  **end**
  Find the 200th most frequent word from wordFreqArray;
**end**

**Algorithm 1:** WordCount by sorting the tokens.

When the tokens are sorted, it is rather easy to count their duplicates–we simply increment the count until a new token is encountered. The challenge is how to sort the tokens efficiently. We give three example sorting algorithms, i.e., Selection Sort, Insertion Sort, and Merge Sort. A starter code written in Java can be downloaded by clicking **this**.

# 5 Tasks

## 5.1 Run the program and report the results (0.5 mark)

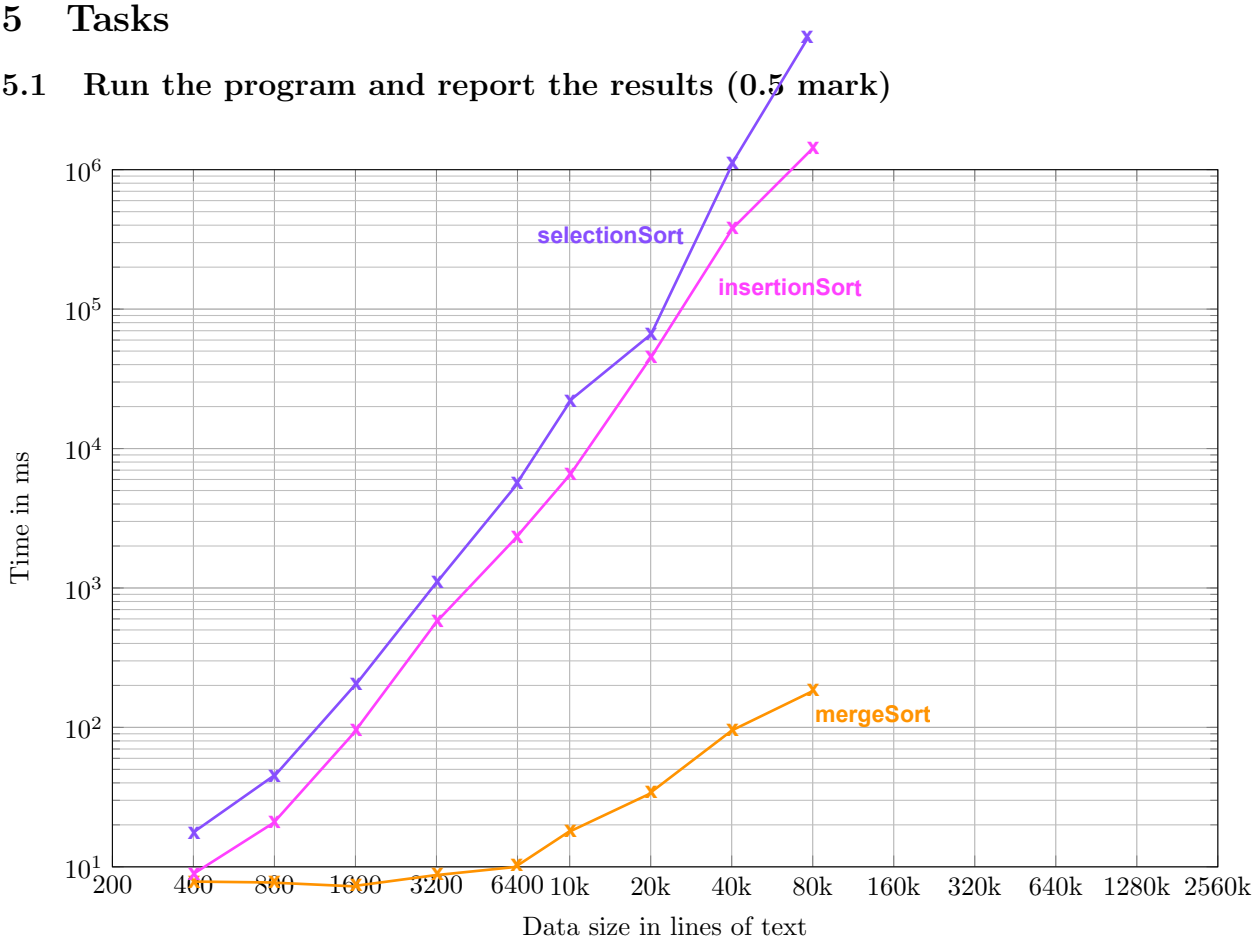


Figure 1: Plot the run time for the counting problem using three sorting algorithms. For Selection-Sort and Insertion-Sort, the program won't be able to run the full gamut of the datasets. You plot the maximum data size that you can run on your computer. Note that it is in log scale.

Read and understand the program and various sorting algorithms. You can run the starter code on your own machine and report your results below by filling in the table and plotting the chart. You can adjust the data sizes that you want to run on our sample code, so that the big data won't stall your machine. As a rule of thumb, you can

skip the data whenever that data runs more than half an hour. When there is not enough space in the form, you can replace zeros with 'k'. In the table there is no space for data sets that are bigger than 80k, but you can plot bigger data in the chart.

There are more data sets that listed below: 200, 400, 800, 1600, 3200, 6400, 10k, 20k, 40k, 80k, 160k, 320k, 640k, 1280k, 2560k.

| Data size | 400 | 800 | 1600 | 3200 | 6400 | 10k | 20k | 40k | 80k |
|---|---|---|---|---|---|---|---|---|---|
| Time for selectSort | 17 | 45 | 200 | 1185 | 5741 | 21k | 76k | 1.2m | 4.2m |
| Time for insertSort | 7 | 21 | 98 | 589 | 2385 | 6820 | 43k | 397k | 1.3m |
| Time for margeSort | 3 | 3 | 2 | 7 | 11 | 17 | 34 | 97 | 195 |

## 5.2 Predict the run-time for big data(0.5 mark)

Since it is impossible to run Select-Sort on big data, we can only predict its approximate run-time. Give your estimate of run-time for Selection-Sort on data $dblp2560k$.

Your estimation of run-time on $dblp2560k$ is 4915200 (in seconds)

## 5.3 Algorithm analysis (1 mark)

Give the complexities of these three algorithms. In particular, give the proof of the complexity of the MergeSort algorithm.

| | Select Sort | Insertion Sort | Merge Sort |
|---|---|---|---|
| Time complexity in Big O notation | O(n^2) | O(n^2) | O(n*logn) |

Step 1: Write the recurrence relation for merge sort:

T(n) = 2 T(n/2) + O(n)

Step 2: Guess its complexity

T(n) = O(n*logn)

Step 3: Prove it using mathematical induction

Base Case: When the array size is 1, there is no sorting required, so T(1) = c=O(1).

Inductive Hypothesis: Assume that the recurrence relation holds for k>=1, i.e. T(k) <= c*k*logk.

Inductive Step: Prove that T(2k) <= c*(2k)*log(2k).
T(2k) = 2*T(k) + c*(2k)
Substitute the hypothesis for T(k):
T(2k) <= 2(c*k*logk) + c*(2k)
T(2k) <= c*(2k)(logk + 1)
T(2k) <= c*(2k)*log(2k)
Therefore, T(2k) = 2k*log(2k) + O(2k) = n*logn + O(n) where n=2k. This matches the guess of T(n) = O(n*logn).
Hence by mathematical induction, the complexity of Merge Sort is O(n*logn).

Figure 2: Proof of the complexity of the merge sort algorithm using the Substitution Method

## 5.4 Read From Text (1 marks)

Text reading is an integral component in our counting algorithms, and it is also ubiquitous in daily applications. The following is a very bad algorithm for reading from a text file, although it is correct and straightforward. It reads from a file line by line, and concatenate them into `text`. Then we split it using a regular expression. The `split` part is fast in linear complexity (we will learn it in Comp-2140), hence that is not our concern. Note that `trim()` is needed to remove empty strings.

```
static String[] readTextBAD(String PATH) throws Exception {
        BufferedReader br=new BufferedReader(new FileReader(PATH));
        String text="";
        String line="";
        while ((line=br.readLine())!=null)
                text=text+" "+line.trim();
        String tokens[]=text.trim().split("[^a-zA-Z]+");
        return tokens;
}
```

### 5.4.1 Time complexity of *readTextBAD*

Write below the time complexity of `readTextBAD` in terms of the text length $n$ (number of lines), and explain your answer. Refer to *repeat*1 example in the textbook for the cause of low speed and its time complexity.

The time complexity of readTextBAD is O(n^2). This is because string concatenation creates new strings each time which would require 1+2+..+n=(n(n+1))/2=(n^2+n)/2=O(n^2). Reading lines and splitting the text both take O(n) time however that is negligible compared to concatenation.

Figure 3: The time complexity for `readTextBAD`

### 5.4.2 Write *readTextGOOD*

The program is slow because of the String data type. One improvement is to use StringBuilder that is explained in our textbook in page 269. Write *readTextGood* that uses StringBuilder. You need to test your program in our submission site here. Also plot the chart in Fig 5 to compare the BAD and GOOD text readers. I understand that the wait can be long when you run these programs, but I want you to experience this painful process so that you will not write this kind of painful programs in the future.

```
static String[] readTextGOOD(String PATH) throws Exception {
  BufferedReader br = new BufferedReader(new FileReader(PATH));
  StringBuilder text = new StringBuilder();
  String line;
  while ((line = br.readLine()) != null) {
    text.append( “  ” ).append(line.trim());
  }
  br.close();
  String[] tokens = text.toString().trim().split( “ [^a-zA-Z]+ ” );
  return tokens;
  }
```

Figure 4: Code for `readTextGOOD`. It should not be the same as in the starter code.

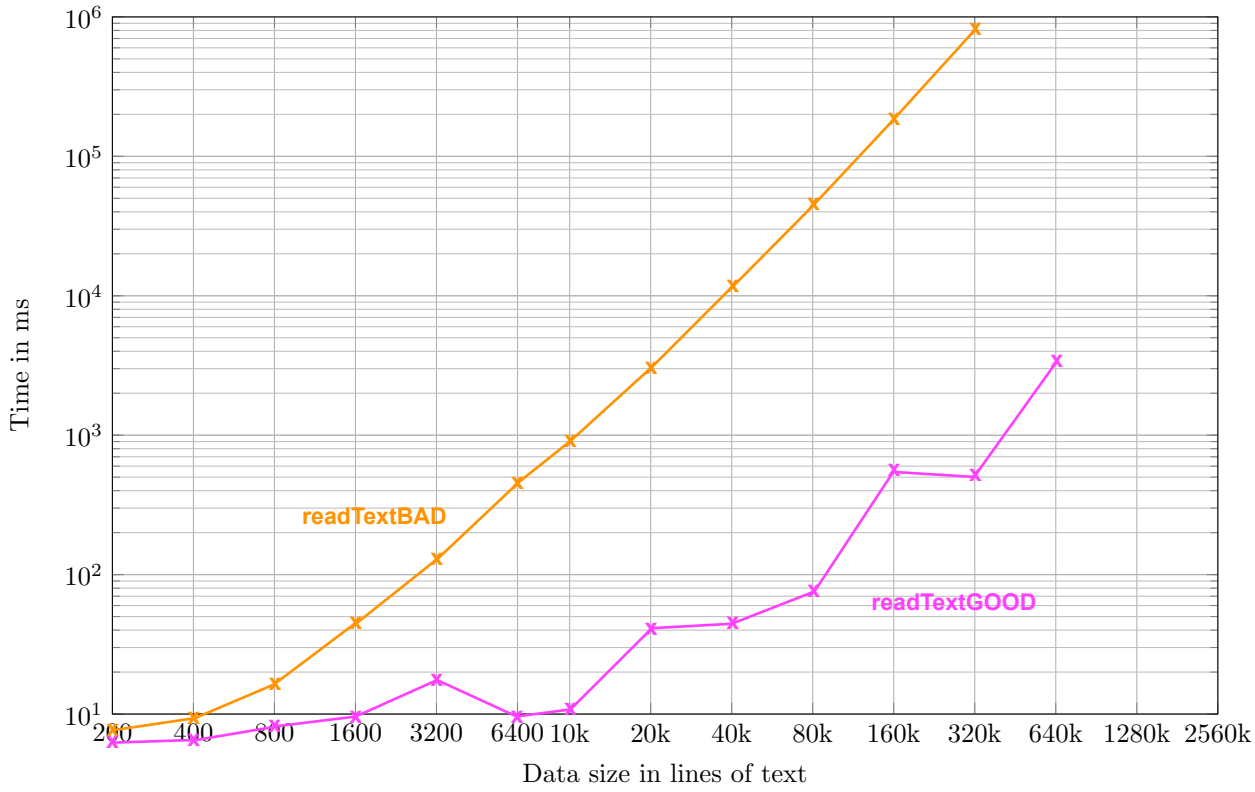


Figure 5: Plot the run time for *textReadBAD* and *textReadGOOD*. You can skip the data points that take too much time. You can run either on our site or on your machine, but you need to run at least once successfully on our site to demonstrate your program *GOOD* version is correct.

### 5.5 Overall Better Counting Program (1 mark)

Now you are ready to write a better counting program, combining the best for text reading, sorting, or other techniques. Similar to A1, you need to write a method that is called `countFAST`, but the method signature (the input and output data types) is a little different:

```
public static Integer countFAST(String fileName) throws Exception { ...}
```

The function is also a little different: it takes a text file name (not an array of strings anymore), and it returns the frequency of the 200-th most frequent word (not the top one). You need to submit this program on our site A2 site, get it run correctly, and able to process large data (at least 1280k data). You won't get any marks if our marking site does not record your successful run on 1280K or 2560k data.

Your biggest data processed: 2560k Time to process this data: 3535ms

Describe your program briefly (e.g., read text using ****, then sort using ****):

countFAST reads text using BufferedReader, counts word frequencies with a HashMap, and then uses a min-heap (PriorityQueue) to find the 200th most frequent word.

## 6 Bonus (Top 6 students will get bonus marks)

The top 6 programs (students) in task 5.5 in terms of speed will receive one bonus mark. The top programs will be selected from the scoreboard at A2 site. You can improve the text read part, or the counting part, or both. You are free to use any techniques. The top speed would be selected from the code running on the 1280k data.