Computer science problems.

About the problems. For this year's problem set you will explore sorting. Although sorting is a well-known problem in computer science, there is no one perfect sorting algorithm that gives the best results for all kinds of data.

What you need to do. For these problems we ask you to write a program (or programs), as well as answer some more theoretical questions. You may answer these questions in comments in your programs or in separate files (pdf preferred).

You may use any programming language you want for your programs, as long as its full implementation is available at no cost and with an easy installation for both Mac and Windows. Since you need to be timing the sorting part of your programs, you need a language that has a timing function/method.

It is best to implement each sorting problem as a separate program so that we can run them separately. We will be looking for the following in your submissions:

- Correct code that we can run. You need to send us all your code files, including the header files for languages like C++. If you are using standard libraries, make sure to include all "import" statements, as required by the language you are using. Make sure to send the files under the correct names, including the file extension (.java, .c, etc). Make sure that the file names do not contain any identifying information about you, such as your first or last name.

- Test data for your code that you have used (you can write it in comment or in a separate file). Make sure to test your code well. For the sorting programs you may have functions that generate test data - include these functions as well, with clear instructions on how to use them.

- Code documentation and instructions. If you are submitting your answers to non-code problems in a separate file, also make sure that it does not have your name in the contents or in the file name. The only place where you specify your name is the zip file with your solutions which must be of the form `yourlastname-CS-solution.zip` (replace `yourlastname` by your actual last name). **Make sure that you use zip compression, and not any other one, such as tar**. In the beginning of each file specify, in comments:

  1. Problem number(s) in the file. If you have a file with "helper" functions, mark it as such.
  2. The *programming language*, including the *version* (Java 1.8 or 1.9, for instance), the *development framework* (such as Visual Studio) that you used, unless you were using just a plaintext editor (notepad, emacs, etc), and the *platform* (such as Windows, Mac, Linux)
  3. Instructions for running your program (how to call individual functions, pass the input (if any), etc), either in comments in your program file or as a separate file, clearly named. Your program may get

input from the user (i.e. it asks to enter some data and then reads it) or you may store the data in specific variables within your program. You need to clearly explain how to input or set the data.

4. Some of your code may be commented out if it is not used in the final run of your program. Make sure it is clear what needs to be uncommented to run code for each of the problems.
5. All of your test data.
6. If you were using sources other than the ones listed here (i.e. textbooks, online resources, etc) for ideas for your solutions, please clearly credit these contributions. This is a courtesy to work of others and a part of ethics code for scholars.

- Clear, understandable, and well-organized code. This includes:

  1. Clear separation between problems; comments that help find individual problems and explain how to run the corresponding functions.
  2. Breaking down code into functions that are clearly named and described (in comments), using meaningful names for variables and function parameters. Your code should be as self-explanatory as possible. While using comments helps, naming a variable `average` is better that naming it `x` and writing a comment "x represents the average".
  3. Minimization of code repetition. Rather than using a copy-paste approach, use functions for repeated code and reuse these functions.
  4. Using well-chosen storage structures (use an array or a list instead of ten variables, for instance) and well-chosen programming constructs (use loops or recursion when you can, rather than repeated code).
  5. For this problem set we will be comparing the output of your programs to correctly sorted output file that we generate. Make sure to follow the data format exactly, otherwise it will not match. Specifically pay attention to line ending: there should be no spaces at the end of data items. If a data item consists of several parts, pay attention to how these parts are separated in the examples: that's how it needs to be separated in the output.

For this problem set you need to make your code as fast as possible. While it's tricky to compare programs written in different programming languages, we will be comparing programs written in the same language and use these as expected benchmarks for other languages. We will be testing your programs for edge cases in data and see how the execution times change.

Note that the sorting time is just one factor in our evaluation: additionally we are looking at approaches that you have tried, eliminating known inefficiencies, data representation choices, and explanations of your decisions.

For this problem set you will explore different sorting algorithms and the conditions in which they produce the fastest solution. While sorting is a widely-explored problem, the fastest algorithms tend to be data-specific. In this problem set you will get a chance to study sorting algorithms in practice, rather than from purely theoretical standpoint. The speed of the program will be measured by the elapsed time (for sorting only, not for reading and printing data). Some sorting algorithms that you might want to review before, or in the process of, working on this problem set:

- Insertion sort,
- Quick sort,
- Merge sort,
- Timsort [2],
- Heap sort,
- Counting sort,
- Radix sort.

The classical textbook that describes all of the above sorting algorithms is [1] except Timsort; other textbooks and resources also provide a good overview.

General requirements. Your program must be written so that it starts by reading the names of two files: input and output (each on a separate line, with a prompt to enter a file name). Then it reads the data from the input file. The first line in the data file is the number of data items. Each data item appears on its own line.

Reading data. You may store the data into any data type you want (for instance, as a string or as a custom-made object). However, you may not compute any properties of the data outside of the timed part of the program. For instance, if your custom object has a field for the length of the string, you may not compute the length and fill in this field as you are reading the data.

Timing. After the data has been read, the sorting starts. For all sorting problems that require **timing**, the timing starts at this point. Note that any data preprocessing that you would like to do is also included into the timed portion.

For timing use the timing functions in the programming language that you are using: start the timer before the sorting, sort the data, and stop the timer after you have sorted the data.

Printing the results. After the data has been sorted, the timing stops. Then the data is written out to the output file, also one item on a line, in a sorted order. After that a **single** number is printed: **the number of milliseconds** that the sorting took.

Please make sure that the data in the output file is exactly the same as in the input file, only sorted. Also make sure that there are no prints other than the time the sorting took.

Other considerations. Your sorting must work on a single processor and not use (or depend on) parallel processing.

You may use any data structures and sorting algorithms that are implemented in your programming language of choice and its commonly used libraries. However, note that you would need to specialize these algorithms to your data and sorting criteria. Also note that you need to include the efficiency of the data structures and algorithms that you are using into your efficiency analysis.

Problem 1: counting sort. In this problem you will explore a non-comparison based sorting using *counting sort* (see [1]). Your goal is to implement counting sort to sort student records based on expected year of graduation of current students, in increasing order. A student record consists of:

- Last name (a string of no more than 30 English letters and symbols, such as a hyphen, single quotation mark, and space), followed by a comma
- First name (a string of no more than 30 English letters and symbols, such as a hyphen, single quotation mark, and space),
- GPA (a number between 0.00 and 4.00 with two decimal places),
- The year of graduation (a number between 2018 and 2022, inclusive).

For instance, a (small) valid input data set is:

```
5
Lee, Riley 3.34 2020
Lopez, Maria 3.41 2019
Martin, Claire-Marie 2.78 2020
O'Neil, Benjamin 3.62 2021
Smith, Jordan 3.07 2018
```

Note that sorting by graduation year must preserve the original order for students with the same graduation year. Thus after sorting by graduation year your program must output:

```
Smith, Jordan 3.07 2018
Lopez, Maria 3.41 2019
Lee, Riley 3.34 2020
Martin, Claire-Marie 2.78 2020
O'Neil, Benjamin 3.62 2021
```

Run your program on a dataset of at least 100,000 elements (randomly generate nonsensical names and GPAs) to get the baseline for subsequent modifications.

Problem 2: sorting by year of graduation and GPA. Now suppose you are given the same kind of data, but you need to sort it by GPA in decreasing order within each group. If two students have the same graduation year and GPA, their order in the input file must be preserved. You may use a combination of two passes of sorting: one by year of graduation, and one by GPA. Alternatively you can use a one-pass approach using whatever sorting algorithm you like to sort by both fields at once.

Your goal is to make your sorting fast; use the timer in the program to measure elapsed time (see general requirements for details). Please answer the following about your program:

- What approaches have you considered or tried, and why did you pick the approach that you are submitting?
- What is the worst case, the best case, and (approximately) the average case as Big-O of the size of the data set? Explain your answer, use the known efficiencies of the sorting that you are using. [1] provides the definition of Big-O efficiency.
- Assuming that you are sorting N elements, how much memory does your approach use? Please explain what the memory is used for.

Problem 3: sorting by student IDs. Suppose the data set is changed to add unique 9-digit student IDs:

```
5
345678916 Lee, Riley 3.34 2020
342368905 Lopez, Maria 3.41 2019
467231450 Martin, Claire-Marie 2.78 2020
398765468 O'Neil, Benjamin 3.62 2021
602345671 Smith, Jordan 3.07 2018
```

1. Is it possible to use counting sort to sort students by IDs? Please clearly explain your answer.
2. Describe how you would use radix sort to sort students by IDs by having a separate sorting pass through the data set for each digit. Would this be a good approach? Please explain your answer; address both memory and time costs. What is the Big-O efficiency of this approach?
3. Now suppose that we use radix perform only three passes through the data set, one for each group of three digits. Each group of three digits has 1,000 possible values. What are the tradeoffs between this method and the one in the previous question? What is the big-O efficiency of this method? Which of the two would be more efficient in practice? Please explain your answer.
4. Write a program to sort the data by IDs. You may use radix sort as described above or in a different way; you may also use a different method altogether if you think it would run faster than radix sort.
5. Describe what you have tried and explain your choice.

Problem 4: sorting strings. Your goal for this problem is to sort strings of upper-case English letters (no spaces, no other symbols). The strings can be anywhere between 1 and 40 letters long (inclusive); each length has an equally likely probability of appearing. Strings are filled in by randomly choosing each letter with the probability of 1/26. Below is a possible (small) data set:

```
7
BSSIQQOJXPHGBBLBHWNO
OAW
GIFMTLWXBAWPTAVAMKJYNDBGBPLUBTYOQZNLYQVS
NLVEB
VYQQ
VZXPUOAGVCXWNVWSXMB
DNQYROSIAAYSLPWAAGAMDUXFRYMJOGHSLHVBM
```

You goal is to sort the data set in decreasing order by the following:

1. The number of occurrences of the most frequent letter in the string, regardless of where it appears.
2. If two strings have the same number of occurrences of the most frequent letter, then by the alphabetical order of such letter. Be careful: a string may have equal number of occurrences of two different letters, in which case the one that's alphabetically first is counted.
3. If two strings are equal by the first two criteria, they are compared alphabetically. If one string is a prefix of another, it is considered to be first, so `APP` precedes `APPLE`.

The sorted data from above is:

```
DNQYROSIAAYSLPWAAGAMDUXFRYMJOGHSLHVBM
BSSIQQOJXPHGBBLBHWNO
GIFMTLWXBAWPTAVAMKJYNDBGBPLUBTYOQZNLYQVS
VZXPUOAGVCXWNVWSXMB
VYQQ
OAW
NLVEB
```

Your goal is to write the fastest program to sort strings according to these comparison criteria. Use timing functions to time the sorting portion of your program, including any necessary pre-processing that you do after you have read the data. Keep in mind that you might need data sets of millions of strings to measure the effects of improvements. Write a program to generate sample data, submit it with your sorting.

As in the previous questions, explain the decisions that you made. Also give a theoretical worst case of your program as Big-O and explain what the expected efficiency is, if different.

Problem 5: sorting sequences of numbers. For the final problem each data item consists of 5 positive integers of no more than 17 decimal digits. Of these five numbers exactly one is a prime number. The sorting is based on the position of the prime number in the sequence. For instance, `4 17 1000000 55 12467568` is smaller than `333 33 3 55555 1020202020202020` because `17` occurs at the position 2, and 3 at the position 3.

If two sequences have their prime numbers at the same position, the one with the smaller value of the prime number is smaller: `4 17 1000000 55 12467568` is smaller than `15 97 8 14142 121`.

If the two primes are at the same position and are the same, all numbers in the sequence are compared one-by-one left-to-right until the first difference, which determines the order. For instance, `4 17 1000000 55 12467568` is smaller than `4 17 100000000000 555555 21`.

Your goal is to write a proigram that sorts the sequences in increasing order based on the comparison described above.

Note that there may be large primes and products of large primes occurring in the data. Your goal is to write a program that runs as fast as possible *even if the data has such cases.*

Explain your decisions and show your test data. Note that your test data cannot be generated fully randomly since large primes are quite rare; you would have to add them in as special cases.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[2] T. Peters. Timsort description. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`. [Online; last accessed 16-9-2018].