CS201 - Assignment 4, Part 2
Due: Wednesday March 12, at the beginning of class

https://xkcd.com/1185/

All of the sorting algorithms we've looked at and that you've probably seen before assume that the data is in memory and that we can swap data elements efficiently. Sometimes, because of the size of the data, you cannot fit all of it into memory. In these situations, many of the traditional sorting algorithms fail miserably; the algorithms do not preserve data locality and end up accessing the disk/hard-drive frequently resulting in very, very slow running times.

For this assignment, you will be implementing an on-disk sorting algorithm that is designed to use the disk efficiently to sort large data sets. The sorting algorithm will work in two phases:

- First, your sorting algorithm breaks the data into reasonable sized chunks and sorts each of these individual chunks. This is accomplished by reading a chunk of data, sorting it, writing it to a file, then reading more data, etc. At the end of this phase, you will have a number of files on disk that are all sorted.
- Second, you will need to merge all of these files into one large file. This is accomplished by pair-wise merging of the files (very similar to the merge of MergeSort) and then writing out the result to a new, larger merged file. Eventually, all of the files will be merged to one large file. Note, this can be done very memory efficiently.

Before starting, read through this whole handout!

1 An Example

Memory (aka random access memory; RAM) allows you to access any portion of memory with roughly the same time cost. Memory is where you programs normally store data when they are running. The problem with memory is that it's expensive to make, so we have a relatively limited amount of it. Hard-drives, on the other hand, can store lots and lots of data. Hard-drives, however, are not designed to be randomly read, instead, they're meant to have large sequential parts read (for example, a file).

Let's say it's 1960 and you have a computer that can only hold 5 characters in memory at once. You have a file that contains the alphabet (26) characters that you'd like to sort. Modern operating systems use what is called "virtual memory". They can make it appear to programs like they have more memory by using the hard-drive to emulate more memory. The problem is that, as we mentioned above, because hard-drives are designed for randomly accessing memory, virtual memory can be very, very slow (some measurements show memory is 100,000 times faster than hard-drives for randomly accessing data).

If we were just to sort our 26 characters in memory/virtual memory, we will incur a huge slowdown. Instead, we can use an on-disk sorting technique. Let's say our file is the following.

`s n b y o i f u m v l t a h j e k w r d q z p g x c`

With on-disk sorting we read in the data 5 characters at a time, sort the data and *then write them back to a file*. We can fully utilize our fast memory without having to rely on the hard drive. If we did that with this data we'd end up with 6 sorted files:

1. `b n o s y`
2. `f i m u v`
3. `a h j l t`

4. `d e k r w`
5. `g p q x z`
6. `c`

We've avoided using more memory than we have, unfortunately, we want to have the whole file sorted and all we have are a bunch of files with part of the data sorted. However, what we can do now is merge each of these files into a single sorted file.

There are many ways this could be done (some more efficient then others). One easy way is to merge the first with the second, then merge that file with the third, etc.

If we did it this way it would involve 5 merges:

1. merging 1 and 2: `b f i m n o s u v y`
2. merging prev with 3: `a b f h i j l m n o s t u v y`
3. merging prev with 4: `a b d e f h i j k l m n o r s t u v w y`
4. merging prev with 5: `a b d e f g h i j k l m n o p q r s t u v w x y z`
5. merging prev with 6: `a b c d e f g h i j k l m n o p q r s t u v w x y z`

Although this is clearly more expensive than a normal sort, if we are constrained by memory, this can be much faster. Notice that the *file merge* function really only needs to have two characters in memory at a time (the next characters in the files it is merging) and therefore is very memory efficient.

An aside: Some of you may realize that there are many different ways that you could merge these files. For example, merge 1 and 2, then merge 3 and 4, then merge these two files together, etc. The approach above is by no means the most efficient, but it's easy to implement and is the one we'll do for this assignment[1].

2 Getting started

Create a new project in `Eclipse` and put the starter class files from:

`http://www.cs.middlebury.edu/~dkauchak/classes/cs201/assignments/assign4/part2/`

into your project (see the part 1 handout for two ways of doing this).

I've also included a `data` directory here with some examples for you to play with:

- `alphabet.shuffled.txt` contains the letters a-z randomly shuffled. This is a great example to use while you're debugging since you can easily look at the output.

[1] Unless you tackle the extra credit!

– `english.shuffled.txt`. Once you think you have it working, try it out on some more realistic data. This file has around 50K different English words in it. Although with modern computers we could still easily sort this in memory, it still serves as a more realistic example. Don't forget to increase your `maxSize` parameter when you run on this data, otherwise you will generate lots and lots of temporary files. Something in between 1,000 and 10,000 would be reasonable.

3 On-disk sorting

We will be implementing an on-disk sorter that sorts the lines (Strings) in a file alphabetically. The only place you will be adding code is in the `OnDiskSort` class. The other three classes are the same we looked at for part 1 of this assignment.

I have provided you with the method headers for the methods that you need to implement. I encourage you to add additional private methods, but **do not** change the names or parameters of the methods I have provided you. This will make our life much easier when we grade the assignment. I have made some of the methods `protected` where normally I would have made them `private` to assist us in grading.

I have provided you with a few methods already to help you get started:

- `OnDiskSort`: the constructor. Make sure that you understand what all of the parameters do. `maxSize` is the maximum number of Strings that can be read in to memory at any one time. We will be creating temporary files along the way (for example, to store the sorted chunks). This will be done in the `workingDirectory` directory (which is also just a `File`). `outputFile` will contain the final result of your sorting.

  The constructor just saves away the input parameters, creates the working directory if it doesn't already exist and then clears out any temporary files in the working directory.

- `clearTempFiles`: Clears all the temporary files. This should be called before and after sorting. However, for debugging purposes, you can leave these calls out to see the intermediary results.

- `getTempFilename`: you shouldn't need to use this method.

- `generateIntermediarySortFiles`: This method performs the first step of the on-disk sorting, which is to read the input file, break it into `maxSize` pieces, sort them and then write these sorted files to temporary files in the working directory. The method returns an array of these files.

You will need to fill in the following methods:

- `writeToDisk`: Writes the data in the array out to the file. This used in the `generateIntermediarySortFiles` method, so it won't work until you write it.

- `copyFile`: Read the data from one file and copy it to a new file. This can be useful in your `mergeFiles` method.
- `merge`: takes two sorted files and merges them into one sorted file. This is very similar to the `merge` method of MergeSort. The main difference is that rather than merging from two arrays you are merging two files. You **should not simply read in the data from both of these files and then use the merge method from MergeSort**. We are trying to be memory efficient and this would defeat the purpose. Instead, you should open `BufferedReader`s to both of the files and then, reading one line at a time, read either from the first file or the second and write that directly out to the output file, depending on the appropriate ordering. Besides the variables for doing the file I/0, you should only need **two** String variables to keep track of the data. Note this is very hard to do using `Scanner`s, so I strongly suggest using `BufferedReader`s.
- `mergeFiles`: takes an array of `File`s, each of which should contain sorted data and then uses the `merge` method to merge them into one large sorted file. Notice that the `merge` method only merges two files at a time. The easiest way to merge all of the n sorted files is to merge the first two files, then merge the third file with the result of merging the first two files, then the fourth in, etc. This is *not* the most efficient way of doing it, however, it will make your life easy (see the extra credit for doing it a better way). **NOTE:** you cannot read and write to a file at the same time, so you will need to use another temporary file to store your temporary results as you merge the data.
- `sort`: this is the public method that will be called when you want to sort new data. For this assignment, we will only be sorting String data. This method shouldn't be very long and will make use of the methods above.

4 Advice on implementation

- Make sure you understand at a high level what we're doing. If you're still fuzzy, reread the example section above, discuss with your classmates or come talk to me.
- Write and test this assignment incrementally. Pick one method (or even a part of a method), code it up and then test to make sure it works. I'd advise tackling them in the order listed above, but you may do them however you'd like. Use the alphabet data to test your methods.
- Make sure you understand the file I/O basics. I've included an Appendix below that has a few simple examples, but also look at the examples from class.
- If you're trying to debug your program, put in lots of System.out.println() statements to help you understand what's happening. Before submitting, make sure to remove these!

5 Extra Credit

As I mention above, this is not an efficient way to merge all of the sorted chunks. Once you have things working above, for extra credit, implement a more efficient `mergeFiles` method. This is

optional and you do not have to do it.

If you do this, I strongly suggest making a new method (i.e. don't delete your original `mergeFiles` method, just rename is to something like `mergeFilesLinear`). If you do the extra credit, put the phrase "EXTRA CREDIT DONE" in a line by itself in the class header under the @date tag, so the TAs know to look more closely at your method.

6 When You're Done

Make sure you have comments at the top of any file/class that you wrote with with your name and the assignment number.

To group these files into a single file, you need to export your project. To do this:

1. Right-click on the project (on Mac, ctrl+click) and select `Export`.
2. You'll see a number of options. Open up the `Java` folder and select `JAR file` and click `Next`.
3. You should just see your project selected. Below, make sure **only** the following two options are checked:
   - "Export Java source files and resources"
   - "Compress the contents of the JAR File"
4. Click on the `Browse...` button and pick a location to save the output file. Give the file a name like "kauchak1.jar", where "kauchak" is your last name and "1" is the assignment number.
5. Click `Finish`.

If all went well this will generate a single `.jar` file wherever you picked to save it.

Submit this JAR file as assignment number "4.2" via the online submission mechanism on the course web page.

Appendix

File locations

For this assignment, you'll need to be specifying files and directories. There are two ways that you can reference a file, using the "relative" path or using the "absolute" path. Is you want to just specify the name of the file/directory and no other information, then you need to put the file in the project folder inside your workspace folder. For example, if you just say "grades.txt" as the filename, it will look there by default.

The other option is to specify the full path. On macs or on the lab machines that is something that starts with '/' and on windows computers it will be something like "C:\". Figuring out the full path of a file will depend on which operating system you're using:

- **On Mac:** press `command + i` on the file. This brings up the file information. You will see a field called `where` that tells you the full path to the file. For example, if the file is called "grades.txt" and the full location is something like "/Users/dkauchak/temp/", then the file can read/written at: "/Users/dkauchak/temp/grades.txt"
- **On Windows:** right-click on the file and select "Copy Address". This will give you the full path name of the file. If you then paste this into a string and add the name of the file, you can then use this to access the file. On a windows computer, you'll need to change all of the backslashes (\) into double backslashes (\\), otherwise, your program won't compile.
- **Linux lab:** If you're browsing to the file via the windows interface, just right-click on the file and select "Properties". Inside this menu you'll find the "Location". If you're using the command-line terminal, just type `pwd`.

Read/Write Examples

Below are three more examples of reading from and writing to files. The first two methods read a file a line at a time and print out the contents to the screen. The first uses the `Scanner` class and the second the `BufferedReader` class. The third function generates random numbers and writes them to a file.

Reading with the `Scanner` class

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadWrite {
    public void readAndPrintScanner(String filename){
        try {
            Scanner scan = new Scanner(new File(filename));

            while( scan.hasNext() ){
                String next = scan.nextLine();
                System.out.println(next);
            }
        } catch (FileNotFoundException e) {
            System.out.println("Couldn't find file: " + filename);
        }

    }
}
```

Reading with the `BufferedReader` class

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadWrite {
    public void readAndPrintBufferedReader(String filename){
        try {
            BufferedReader reader = new BufferedReader(new FileReader(filename));

            String next = reader.readLine();

            while( next != null ){
                System.out.println(next);
                next = reader.readLine();
            }
        } catch (IOException e) {
            System.out.println("Problems with file: " + filename);
        }

    }
}
```

Writing a file with random numbers in it using a `PrintWriter`

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Random;

public class ReadWrite {
    public void writeRandomNums(int num, String filename){
        Random rand = new Random();

        try {
            PrintWriter writer = new PrintWriter(new FileOutputStream(filename));

            for( int i = 0; i < num; i++ ){
                writer.println(rand.nextInt());
            }

            writer.close();
        } catch (IOException e) {
            System.out.println("Problems with file: " + filename);
        }
    }
}
```

Testing your code

Testing writeToDisk

```
public static void main(String[] args){
    OnDiskSort diskSorter = new OnDiskSort(5, new File("working"), new MergeSort());

    String[] writeMe = {"I", "love", "CS"};
    File outFile = new File("output.txt");

    diskSorter.writeToDisk(outFile, writeMe);
}
```

Testing copyFile

```
public static void main(String[] args){
    OnDiskSort diskSorter = new OnDiskSort(5, new File("working"), new MergeSort());

    // the output from the last test
    // this file MUST exist!
    File inFile = new File("output.txt");
    File outFile = new File("output.copy.txt");

    diskSorter.copyFile(inFile, outFile);
}
```