Workbook 0

The hard drive in a computer is capable of storing several orders of magnitude more data than RAM. In this workbook you will write a program to sort a large list of integers stored in a file which may not fit in the memory available to the Java Virtual Machine (JVM). Algorithms which sort data in this way are sometimes called *external sorting algorithms*.

Important

The deadline for this exercise is 5pm on Monday 8th October 2018.

An on-line version of this guide is available at: https://www.cl.cam.ac.uk/teaching/1819/FJava and you should check this page regularly for announcements and errata. You might find it useful to refer to the on-line version of this guide in order to follow any provided web links or to cut 'n' paste example code.

In order to complete the task described in this workbook, you will need to use Chime and Git as a development environment for Java programs. You can create a new git repository on Chime via https://www.cl.cam.ac.uk/teaching/1819/FJava/ticklet0. There is an introductory guide to these tools on the course website if you need help.

In the Programming in Java course, you used input and output streams to read and write data to the file system. Some programs, such as external sort, benefit from the ability to read and write data at an arbitrary position in the file. The offset at which a read or write occurs is often recorded in a *file pointer* or *file descriptor*. The class `java.io.RandomAccessFile` supports this feature in Java and an instance of this class keeps track of a specific byte offset from the start of the file which will be read or written to. When a new instance of `RandomAccessFile` is created, the file pointer will typically be located before the first byte of the file, and a successful call to a `read` (or `write`) method will move the location of the file pointer by the number of bytes read (or written). In addition to read and write methods, `RandomAccessFile` objects support a `seek` method, which the programmer can use to move the file pointer to an arbitrary location in the file, and a `length` method which returns the length of the file in bytes. Here is a simple example:

```
RandomAccessFile f = new RandomAccessFile("/home/arb33/example","rw");
f.writeInt(1); //write a "1" into the first four bytes of the file
f.writeInt(2); //write the value "2" after the value "1"
f.writeInt(3); //write the value "3" after the value "2"
f.seek(4); //file pointer now between fourth and fifth byte
System.out.println("Read four bytes as an int value "+f.readInt());
System.out.println("The file is "+f.length()+" bytes long");
```

1. Compile and execute the code snippet above. What does it print?

Writing integer values using a `RandomAccessFile` object is not efficient as each call to `writeInt` results in a system call to the operating system to write the data to the hard disk. One way of making writes more efficient is to cache a number of writes in memory and then coalesce these write operations together into a single, larger, write operation. A convenient way to coalesce the `writeInt` method calls in the sample code above is to create an *output stream* from the file descriptor held by an instance of `RandomAccessFile`. This output stream can then be wrapped in a `BufferedOutputStream` object to create a small in-memory buffer. Finally, the buffered output stream can be wrapped in a `DataOutputStream` object to add support for writing primitive integers. In other words:

```
RandomAccessFile f = new RandomAccessFile("/home/arb33/example","rw");
DataOutputStream d = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(f.getFD())));
d.writeInt(1); //write calls now only store primitive ints in memory
d.writeInt(2);
d.writeInt(3);
d.flush(); //force the contents to be written to the disk. Important!
f.seek(4);
System.out.println("Read four bytes as an int value "+f.readInt());
System.out.println("The file is "+f.length()+" bytes long");
```

In the previous example, only the calls to `writeInt` are buffered through the stream; the call to `readInt` on the penultimate line is still made directly on the `RandomAccessFile` object. There is an additional method call `d.flush()` which is used to write any data which might be held in the memory cache to the hard disk; this is crucial since otherwise the subsequent method call `f.readInt()` may not produce the correct answer as the data may not have been written to the hard disk yet. As you might expect, the classes `FileInputStream`, `BufferedInputStream` and `DataInputStream` can be used in an analogous way to create an input stream to a file if you need to read multiple data items from a file in an efficient manner.

Input streams support a `skipBytes` method to move the file pointer forwards through the stream. They do not support a `seek` method, so you can only progress forwards through a file. (Why do you think this is?) If you need to make multiple passes through a file in an efficient manner, you can create a new input for each pass.

External Sort

To gain Ticklet 0 you will need to implement an external sorting algorithm. Your sorting algorithm will be provided with two file names:

1. file *A*, containing a list of Java primitive integers written in binary format which can read and written using the `readInt` and `writeInt` methods as shown above; and

2. file *B*, which is the same length as file *A*, and whose initial contents are not defined, but the space can be used as an additional place to store data whilst your program executes.

After your program has finished executing, file *A* should contain the same set of integers it initially contained, except that they should be sorted into numerical order, from smallest to largest. The contents of file *B* can be anything you like. You *must not* create any additional temporary files on the hard disk whilst sorting data, and you must not increase the size of file *A* or file *B*. You can use any stack and heap space available to the JVM for storing copies of (portions of) the data stored in the files, but remember that the amount of data held in the files may exceed the memory available to the JVM! *You cannot make use of any machine RAM except via the JVM; for example, use of the* `java.nio.channels.FileChannel` *to memory-map a file directly into RAM is not allowed.*

There are many ways to perform an external sort, and the final choice of algorithm is up to you. One simple solution is to use merge sort as follows. Imagine dividing the integers stored in the input file into *n* sorted blocks, each of size *k*, where *k* is initially equal to one and *n* is equal to the number of integers stored in the file. On each pass through the file, data held in odd blocks can be merged with data held in even blocks to create *n/2* blocks each of length *2k*. On each pass, data will need to be read from one file and written to the other. For example, on the first pass, data in file *A* will be used as input, and the results of the merge will be written into file *B*; on the second pass, data in file *B* will be used as input, and the results of the merge will be written into file *A*. In this way, after log *n* passes, a sorted copy of the data will reside in either file *A* or file *B*.

You can implement external merge sort by creating two `RandomAccessFile` objects for file *A* called `a1` and `a2`, and another two `RandomAccessFile` objects for file *B* called `b1` and `b2`. On each pass, one of the files, let's say *A*, will contain the input data, and the other file, let's say *B*, will be used for output. At the start of the pass, file pointers `a1` and `a2` should be located at the start of the first and

second blocks of (sorted) data and file pointer `b1` should be located at the start of file *B*. The integers in the first block, accessible from `a1`, can be merged with the integers in the second block, accessible from `a2`, and the results written to `b1`; when the first two blocks have been merged, then `a1` and `a2` can seek ahead in file *A* to point to the start of the third and fourth blocks of file *A*, and the process repeated until all pairs of blocks in file *A* have been merged into blocks of twice the size in file *B*.

There are some additional subtleties with the above approach which will also need to be addressed. Firstly, you may find that on some passes you have an odd number of blocks. Secondly, the final block in the file may be shorter than *k* integers in length, so you will need to make use of the `length` method on the `RandomAccessFile` object to ensure that you copy only valid elements from the final block. Finally, as mentioned in the previous section, calling `readInt` or `writeInt` directly on a `RandomAccessFile` object is simply too slow to be practical, so on each pass through the file you will need to create appropriate buffered input and output streams; you can use the `skipBytes` method on such streams to advance to new blocks on the input file as necessary. Figure 1, “External merge sort”, shows an execution trace of external merge sort for a file containing six integers; note the last copy stage, which is required here to ensure the sorted list exists in file *A* when the program terminates.

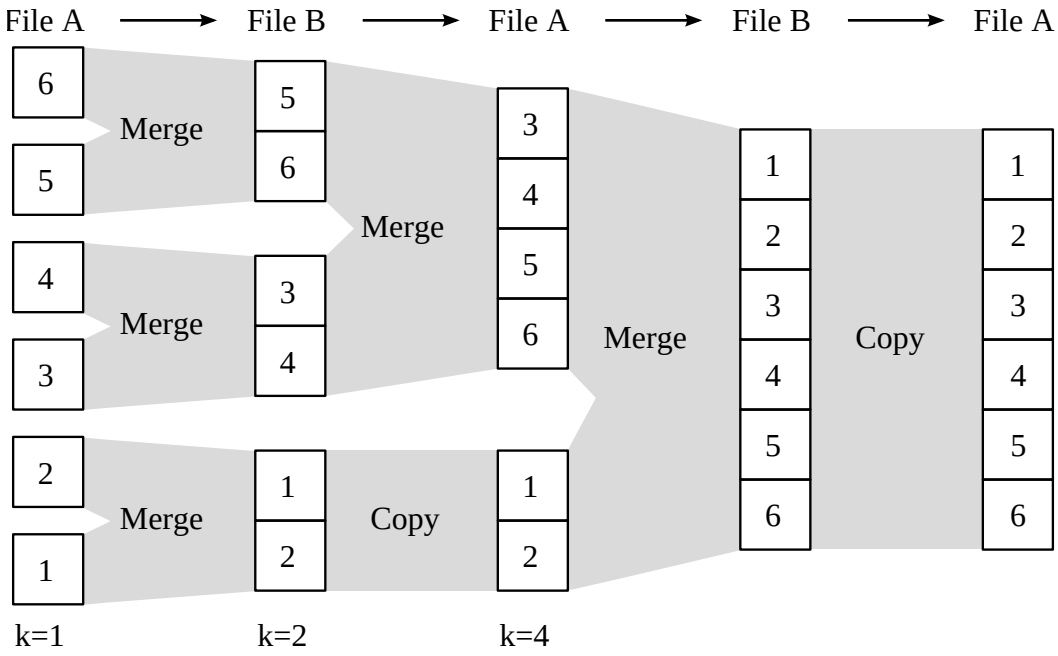

Figure 1. External merge sort

Create a class called `ExternalSort` in the package `uk.ac.cam.crsid.fjava.tick0`. The class must contain a `sort` method with the following prototype:

```
public static void sort(String filenameA, String filenameB)
                        throws FileNotFoundException, IOException
```

The `sort` method must sort the integers found in `filenameA` into numerical order from smallest to largest. Your implementation may use the space provided in `filenameB` as temporary storage.

In order to help you test your program, a skeleton implementation of `ExternalSort` is available for download from the course website, together with a ZIP file which contains a set of input files to test your program. When you have completed your implementation you can use the `main` method provided in `ExternalSort` to test your program. For example, given two files, `test1a.dat` and `test1b.dat` from the ZIP file, you can test your program as follows:

```
crsid@machine~:> java -Xmx10m uk.ac.cam.crsid.fjava.tick0.ExternalSort \
                 test1a.dat test1b.dat
The checksum is XXXXXX
```

You should check that the value of the checksum printed by your program is the same as that printed in `checksum.txt` (a file also available in the ZIP file) to double-check that you have sorted the list of integers correctly. *If you do not get the correct checksum value, your implementation is incorrect, and there is no point submitting your code as it will fail!* In the above example, the option `-Xmx10m` limits the amount of memory available to the JVM to 10 MB; you may like to alter this parameter in order to see how the performance of your solution varies.

You are, of course, welcome (and encouraged!) to devise your own external sorting algorithm, however your own implementation of `ExternalSort` *must* retain the `sort` method with the prototype as described above so that the testing framework can check your implementation is correct. Remember that the amount of memory available to the JVM may be less than the size of the input files to be sorted.

Tick 0 and Tick 0*

You have now completed all the necessary code to gain Ticklet 0. All submitted implementations will be checked for correctness against all the provided test datasets, together with several additional datasets which we do not make available to students. You can submit your work via Chime by starting at this URL: https://chime.cl.cam.ac.uk/start/fjava/tick0.

This year the tickers will be checking your code for good programming style. We understand that this is a subjective area but would like you to comply with the following guidelines. We will use (with permission) a slight adaption of the BAE Systems Applied Intelligence coding standard. You should comply with everything described on the three-page summary except where we have marked it as optional ('Opt'). The following documents are available from the course website:

- Three-page summary of guidelines as annotated code [BAE-Java-standard-summary.pdf]
- Complete document with full explanations and examples [BAE-Java-standard.pdf]

We will award a star to the ten implementations which are submitted before the deadline, and process all our datasets correctly in the shortest total time. You may resubmit your solution to improve your time as often as you like. Once we have released the testing framework, we will update a leaderboard of the time taken by each submission on the course website, so you will be able to see how well your implementation is doing compared with the others. Good luck!

We encourage you to discuss your ideas on external sorting algorithms with each other and to seek help from other students in order to understand and clarify the exercise requirements. However, your final submission must be the result of your own work and should only contain code you have written yourself.