

Balanced Sort Merge


You are to write java program(s) to carry out an external balanced sort merge, following the specification given below. You may work individually or as a team of two. Solo solutions require a balanced 2-way sort merge while pairs must complete a balanced 4-way sort merge.

Specification

1. You are sorting text files, and each line is one datum. That is, if you input two entire lines of some book then they are compared using a Java string compare function, treating each line as if it was [say] a single entry in a dictionary. That is, you are **not** sorting just words or letters or numbers you are sorting entire lines, comparing one line with another. This is just like the Linux "sort" command.
2. Be careful when reading text files using a function like `readLine()` because different operating systems and applications may use different control character combinations to indicate end-of-line, and the reading method may throw out the actual end-of-line marker(s) in the input file and substitute its own preferred one when outputting. You need to guard against this so that the size of the sorted output is the same as the original input no lost characters. (see <https://stackoverflow.com/questions/42769110/keep-new-lines-when-reading-in-a-file> as an example starting place to achieve this).
3. You must write code to take plain text input and produce initial runs, and you must write code to subsequently perform a balanced k-way sort merge on those initial runs (where k is either 2 or 4). You may either write separate programs and combine their execution with a batch/script file, or just write one program to do it all, but for testing we will want to see the output of each part. So ...
4. Your program must be called XSort. It must create initial runs first and then merge them in a separate step. Initial runs can be between 64 and 1024 in length, and this is controlled with a commandline argument. If the argument provided is outside of this range then your program can abort (probably good to print a usage message so the user knows what went wrong).
5. Input text (to be sorted) should be read as standard input, not directly from a file, so that your sorting program could be part of any process pipe. And all output should be to standard output for the same reason. In Linux, you can pipe the output of any program into another program using the "|" operator, and you can redirect output

into a file using the ">" operator. So, if we had a text file called MobyDick.txt and wanted initial runs of length 512 and wanted to store those runs into a file called Result.txt then we would use the following java incantation (if the program and data are in the same directory) "cat MobyDick.txt | java XSort 512 > Result.txt"

6. Initial runs must be created using heapsort, implemented from scratch by the student(s).
7. Merging: if a second commandline argument is provided then it must be either the integer 2 or the integer 4, indicating whether a balanced two-way or four-way sort merge is happening. That is, solo solutions will only accept a 2, and pair solutions will only accept a 4, and any other value should abort with an error message. This should make it easy for the marker to tell which program type they are testing.
8. When merging is asked for, the initial runs should be distributed into two (or four) temporary files and then merging passes are carried out (back and forth, two temporary input files and two temporary output files) until just one sorted run is produced (which goes to standard output). To carry out a balanced 2-way sort merge on the aforementioned MobyDick.txt using initial runs of length 64 and then put the result in Moby.sorted then we would use "cat MobyDick.txt | java XSort 64 2 > Moby.sorted"
9. Modest documentation should be included in your source code, and any contributors should have their names and student ID numbers in the header documentation. If any external help was obtained or any AI used then a note to this effect should be included in the header documentation as well, indicating where the help came from and what it was. Note that it is okay to get some help, provided it is credited to the correct source. Obviously you can't get someone else to do your

assignment entirely 

Submission: Submit a compressed folder with all your code and a README.txt file with instructions on how to use your code and any other information you want the marker to know about. Make sure you include student names and ID numbers in the documentation of each code segment, and in the README.

Your folder name should be your student ID number if you are a solo solution, or both ID numbers separated by an underscore if you are a pair.

Procedure: create an empty folder whose name is your student ID. Copy only source code and README into it (i.e. no compiled code or sample data or anything else). Compress it

and submit via Moodle. Only one person in a pair needs to submit for a pair. The other person in the pair should probably submit a note saying "I'm with so-and-so".

E.g. on Linux, have a look at the "tar" command as one way to make a compressed folder (called a tarball, if you didn't know).

Some sample text files will be provided on the course Moodle page.

We're looking for nice, well-formatted code with a little documentation that compiles and runs on the Lab linux machines in R Block (make sure you compile and test in the lab before submitting!!!) and which executes the correct algorithms.