EE 602

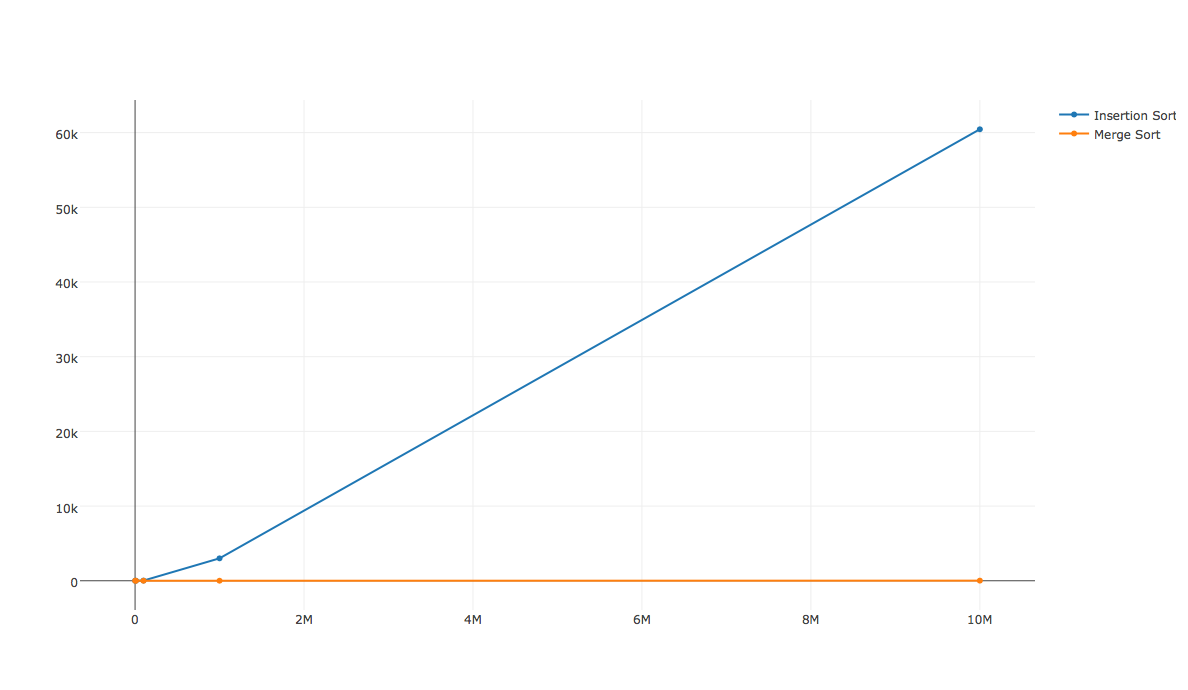
Java Assignment #1

Hualiang Li

Task 1:

To run mysortGUI code, move mysortGUI.java to the test driver folder you provide to us, it is same directory with sortGUI.java. Then just run it. Then it will pop up a GUI. Then, you can generate different data sizes and different sorting algorithms. The elapsed time will be printed in the console within eclipse.

See attached source code in the email for detailed implementation. Basically speaking, I just reused the sortGUI class and added the time measurement. I also change the data type from float to integer. Below is my plot of different data size vs. elapsed time for merge sort and insertion sort:



Here is the table:



From the result, I can see that for small data size, the insertion sort is as fast as merge sort. However, as data size increasing, the merge sort is much efficient then the insertion sort. This is because the insertion sort is O(n2), while the merge sort is O(nlogn).

Task 2:

To run ExternalSort.java, move my ExternalSort.java to the test driver folder you provided, it is the same folder with sortGUI.java. Then just run it. The console will print out:

Size of the input file to be sorted;

Memory capacity we can use to sort for a run;

Number of blocks we are sorting for each pass;

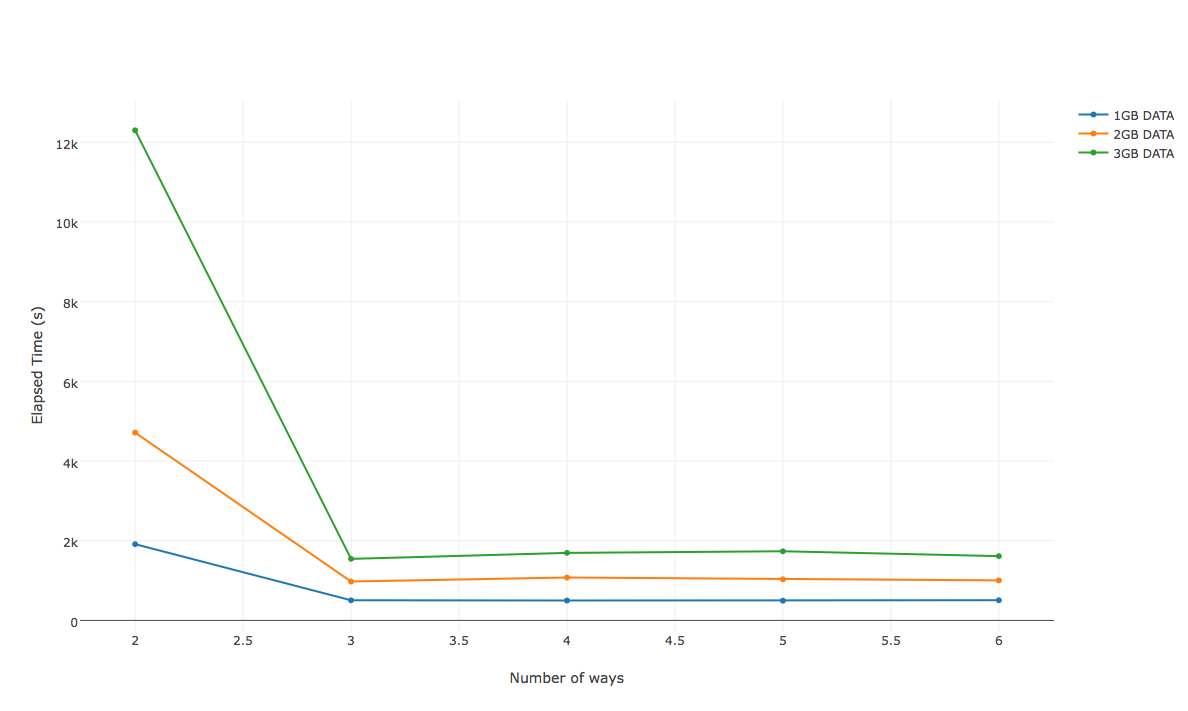
The total elapsed time;

To change the number of ways, say you want to implement 2-way external sort, go to main function and change variable “numWays” to 2. For 3-way, change it to 3, and so on so forth.

To change the input file size, go to main function in ExternalSort class, assign variable “inputSize” to the size you like. The size unit is in GB.

One thing worth to mention is I use priority queue to merge files. This is useful when we running B-way external sort.

Below are my plot and the table of different data size and different ways external sorting. I was using 30MB memory to sort.



From the plot, I can see that for file size 2GB, B=3 is my best system setup. This can be seen from the plot. This is because with more way merge, we reduce the height of the merge tree. The reading and writing is 2P(log(P)+1), reduce P will reduce I/O. However, this does not keep decrease the elapsed time because after a certain B size, the sort will dominate the elapsed time, instead of file I/O.

Appendix A

Task 2: two way external sort and B way external sort:

**package** testjava;

**import** java.util.\*;

**import** java.io.\*;

// External Sort

// Adjust parameter in the main function

// Parameter includes target file size, which is the file to be sorted.

// Number of ways to sort, typically range in 2-6.

//

**public** **class** ExternalSort {

**public** **void** generateFile(String fileName, **double** size) **throws** FileNotFoundException, UnsupportedEncodingException

{

//Size in Gbs of my file that I want

**double** wantedSize = Double.*parseDouble*(System.*getProperty*("size", Double.*toString*(size)));

Random random = **new** Random();

File file = **new** File(fileName);

PrintWriter writer = **new** PrintWriter(**new** BufferedWriter(**new** OutputStreamWriter(**new** FileOutputStream(file), "UTF-8")), **false**);

**int** counter = 0;

**while** (**true**) {

String sep = "";

**for** (**int** i = 0; i < 100; i++) {

**int** number = random.nextInt(1000000000) + 1;

writer.print(sep);

writer.print(number);

sep = "\n";

}

writer.println();

//Check to see if the current size is what we want it to be

**if** (++counter == 50000) {

System.***out***.printf("Now Size: %.3f GB%n", file.length() / 1e9);

**if** (file.length() >= wantedSize \* 1e9) {

writer.close();

**break**;

} **else** {

counter = 0;

}

}

}

System.***out***.printf("Created a file of %.3f GB\n", file.length() / 1e9);

}

// Divide the file into small blocks. If the blocks

// are too small, we shall create too many temporary files.

// If they are too big, we shall be using too much memory.

**public** **static** **long** estimateBestSizeOfBlocks(File filetobesorted) {

**long** sizeoffile = filetobesorted.length();

// we don't want to open up much more than 1024 temporary files, better run

// out of memory first. (Even 1024 is stretching it.)

**final** **int** MAXTEMPFILES = 1024;

**long** blocksize = sizeoffile / MAXTEMPFILES ;

// on the other hand, we don't want to create many temporary files

// for naught. If blocksize is smaller than half the free memory, grow it.

// long freemem = Runtime.getRuntime().freeMemory();

**long** freemem = 1024\*1024\*30; //1kb \* 1024 \* 30 = 30MB

System.***out***.println("File size= " + sizeoffile/1000000+"MB");

System.***out***.println("Using memory = " + freemem/1000000 +" MB");

**if**( blocksize < freemem/2)

blocksize = freemem/2;

**else** {

**if**(blocksize >= freemem)

System.***err***.println("We expect to run out of memory. ");

}

**return** blocksize;

}

// This will simply load the file by blocks of x rows, then

// sort them in-memory, and write the result to a bunch of

// temporary files that have to be merged later.

//

// @param file some flat file

// @return a list of temporary flat files

**public** **static** List<File> sortInBatch(File file, Comparator<String> cmp) **throws** IOException {

List<File> files = **new** ArrayList<File>();

BufferedReader fbr = **new** BufferedReader(**new** FileReader(file));

**long** blocksize = *estimateBestSizeOfBlocks*(file);// in bytes

**try**{

List<String> tmplist = **new** ArrayList<String>();

String line = "";

**try** {

**while**(line != **null**) {

**long** currentblocksize = 0;// in bytes

**while**((currentblocksize < blocksize)

&&( (line = fbr.readLine()) != **null**) ){ // as long as you have 2MB

tmplist.add(line);

currentblocksize += line.length(); // 2 + 40; // java uses 16 bits per character + 40 bytes of overhead (estimated)

}

files.add(*sortAndSave*(tmplist,cmp));

tmplist.clear();

}

} **catch**(EOFException oef) {

**if**(tmplist.size()>0) {

files.add(*sortAndSave*(tmplist,cmp));

tmplist.clear();

}

}

} **finally** {

fbr.close();

}

**return** files;

}

**public** **static** File sortAndSave(List<String> tmplist, Comparator<String> cmp) **throws** IOException {

Collections.*sort*(tmplist,cmp); //

File newtmpfile = File.*createTempFile*("sortInBatch", "flatfile");

newtmpfile.deleteOnExit();

BufferedWriter fbw = **new** BufferedWriter(**new** FileWriter(newtmpfile));

**try** {

**for**(String r : tmplist) {

fbw.write(r);

fbw.newLine();

}

} **finally** {

fbw.close();

}

**return** newtmpfile;

}

// This merges a bunch of temporary flat files

// @param files

// @param output file

// @return The number of lines sorted.

**public** **static** File mergeSortedFiles(List<File> files, **final** Comparator<String> cmp) **throws** IOException {

PriorityQueue<BinaryFileBuffer> pq = **new** PriorityQueue<BinaryFileBuffer>

(11, **new** Comparator<BinaryFileBuffer>() {

**public** **int** compare(BinaryFileBuffer i, BinaryFileBuffer j) {

**return** cmp.compare(i.peek(), j.peek());

}

});

**int** total = 0;

**for** (File f : files) {

total += f.length();

//System.out.println("file size = " + f.length()/1000000);

BinaryFileBuffer bfb = **new** BinaryFileBuffer(f);

pq.add(bfb);

}

**final** File folder = **new** File("./");

File newtmpfile = File.*createTempFile*("merge", "flatfile", folder);

newtmpfile.deleteOnExit();

BufferedWriter fbw = **new** BufferedWriter(**new** FileWriter(newtmpfile));

**try** {

**while**(pq.size()>0) {

BinaryFileBuffer bfb = pq.poll(); //poll block that contains smallest in the priority queue

String r = bfb.pop(); //pop out the block head

fbw.write(r);

fbw.newLine();

**if**(bfb.empty()) {

bfb.fbr.close();

bfb.originalfile.delete();// we don't need you anymore

} **else** {

pq.add(bfb); // add it back

}

}

} **finally** {

fbw.close();

**for**(BinaryFileBuffer bfb : pq ) bfb.close();

}

//System.out.println("It should be: " + total + ". The real size is: " + newtmpfile.length());

**return** newtmpfile;

}

**public** **static** **void** copy(File inputfile, String output) **throws** IOException{

BufferedReader fbr = **new** BufferedReader(**new** FileReader(inputfile));

String line = "";

PrintWriter writer = **new** PrintWriter(output, "UTF-8");

**try**{writer.println(line);

**while**((line = fbr.readLine()) != **null**){

writer.println(line);

}

}**finally**{

fbr.close();

writer.close();

}

}

**public** **static** **void** main(String[] args) **throws** IOException {

// if(args.length<2) {

// System.out.println("please provide input and output file names");

// return;

// }

// String inputfile = args[0];

// String outputfile = args[1];

String inputfile = "myinput";

String outputfile = "myoutput";

**double** inputSize = 2; //Unit: GB

**int** numWays = 6;

//new ExternalSort().generateFile(inputfile, inputSize);

Comparator<String> comparator = **new** Comparator<String>() {

**public** **int** compare(String r1, String r2){

**int** val1, val2;

val1 = Integer.*parseInt*(r1);

val2 = Integer.*parseInt*(r2);

**return** val1 - val2;

}

};

**long** tStart = System.*currentTimeMillis*(); // starting a timer couting in milliseconds //DO SORTING

List<File> inputl = *sortInBatch*(**new** File(inputfile), comparator);

List<File> outputl = **new** ArrayList<File>();

Boolean end = **false**;

**int** pass = 0;

**while**(!end){

System.***out***.println("pass " + pass + ": number of blocks = " + inputl.size());

**if**(inputl.size() <= 1) **break**;

outputl = **new** ArrayList<File>();

**for**(**int** i=0; i<inputl.size(); i+=numWays){

List<File> group = **new** ArrayList<File>();

**for**(**int** j=i; j<i+numWays && j<inputl.size(); j++){

group.add(inputl.get(j));

}

outputl.add(*mergeSortedFiles*(group, comparator));

}

pass ++;

inputl = outputl;

}

*copy*(outputl.get(0), outputfile);

**long** tEnd = System.*currentTimeMillis*();

**long** tDelta = tEnd - tStart;

**double** elapsedSeconds = tDelta / 1000.0;

System.***out***.println(elapsedSeconds);

}

}

**class** BinaryFileBuffer {

**public** **static** **int** *BUFFERSIZE* = 2048;

**public** BufferedReader fbr;

**public** File originalfile;

**private** String cache;

**private** **boolean** empty;

**public** BinaryFileBuffer(File f) **throws** IOException {

originalfile = f;

fbr = **new** BufferedReader(**new** FileReader(f), *BUFFERSIZE*);

reload();

}

**public** **boolean** empty() {

**return** empty;

}

**private** **void** reload() **throws** IOException {

**try** {

**if**((**this**.cache = fbr.readLine()) == **null**){

empty = **true**;

cache = **null**;

}

**else**{

//System.out.println(cache);

empty = **false**;

}

} **catch**(EOFException eof) {

empty = **true**;

cache = **null**;

}

}

**public** **void** close() **throws** IOException {

fbr.close();

}

**public** String peek() {

**if**(empty()) **return** **null**;

**return** cache.toString();

}

**public** String pop() **throws** IOException {

String answer = peek();

reload();

**return** answer;

}

}