due Sunday, 4 February 2018, at 11:45 PM

Summary. *You will implement and do computational experiments with insertion sort, merge sort, Heapsort, and the sorting utility provided by your language of choice. Your programs will output the input, sorted in ascending order, plus runtime and number of comparisons. You will also write a report detailing the results of your experiments.*

Grading

Your actual program, as submitted to the programming assignment locker `Program 1`, will determine 75% of your grade, as follows.

- 20% based on code inspection – is your code readable, well-documented, and easily identifiable as a faithful implementation of the algorithm it implements?
- 12% for compilation – do your programs compile without errors?
- 20% for correctness – do your programs correctly sort the input?
- 15% for correct reports of number of comparisons for the three sorting algorithms you implemented yourself.[1]
- 8% for reports of runtime that are within reason.

The other 25% will be based on the quality of your report as submitted to the separate assignment locker for it, `Program 1 Report`. In particular,

- 5% for coherent description of your experimental setup
- 5% for judicious choice of inputs
- 10% for informative tables and charts
- 5% for summary and conclusions

The purpose of this assignment is to get a feel for how the sorting algorithms you've learned perform "in real life" and how they stack up against the sorting utility provided by your programming language of choice. There are two parts to this assignment, to be submitted to two distinct GradeScope lockers.

- `Program 1.` This is a GradeScope programming assignment locker to which you will submit the required programs and scripts following instructions detailed below *to the letter.*
- `Program 1 Report.` This is posted like a GradeScope homework assignment. You are expected to write a (roughly) two-page report with several tables and charts, following instructions given below.

Instructions for the sorting algorithm implementations

You will implement insertion sort, merge sort, heapsort, and a program that calls on the sorting utility provided by your programming language. All of these must be implemented in the same programming language and *use the same internal data structure* for elements to be sorted (integers). In particular, you may not use a list-based version of merge sort while using an array-based version of heapsort. And you have to use whatever data structure is expected of the sorting utility of the programming language: an array for C/C++, an `ArrayList` for Java, and a list for Python. Those four languages are the ones we will support.

Each of your programs should

- read input from *standard input*,
- run the sorting algorithm,
- determine the runtime in milliseconds and number of key comparisons done by the algorithm,
- write the output (sorted) to *standard output*, one integer per line

[1]There's only one correct answer for insertion sort. When there are equal elements in merge sort, the one in the first half should be favored. In case of Heapsort, you should always swap with the *left child* if there are two children with equal keys.

- write a two lines to *standard **error***:

  ```
  runtime,MILLISECONDS
  comparisons,NUMBER_OF_COMPARISONS
  ```

  `MILLISECONDS` should be the number of milliseconds expressed as an *integer* – runs that are less than one millisecond are not worth the bother in an experimental study, and the runtime should be measured for the sorting only (not including input/output). `NUMBER_OF_COMPARISONS` should count the number of *key comparisons*, ones that compare input integers.[2]

Input (from `stdin`) will have the following format

`n` *number_of_integers* (length of array/list)
$A[1]$
...
$A[n]$

where n is *number_of_integers*. The first line is there primarily for C implementations, to make it easy to allocate an array of the right size initially.

Your programs should all use the same standard comparison method used by the language's sort utility. In C, C++, and Python, this means passing a comparison function to the sorting function. In Java, it means overloading the `compareTo` method or defining a `Comparator`. The comparison function or comparator should increment a global counter each time a comparison takes place.

What to submit

Follow the GradeScope submission instructions, posted separately in a file called
`gradescope-program-submission-instructions.pdf`,
and submit the following files

- all source code and any supporting files (e.g., a `makefile`),
- `compile_programs.sh` – a (bash) script that compiles all of your programs, if necessary; for Python, it might be empty; for Java, it would probably contain the line `javac *.java`; and for C/C++, the easiest would be `make`, assuming you submit a `makefile`.
- the following scripts, each of which runs a different sorting algorithm implementation (which one should be obvious from the name).
  - `run_insertion_sort.sh`
  - `run_merge_sort.sh`
  - `run_heapsort.sh`
  - `run_sort_utility.sh`

  Be sure to make all programs and scripts executable – the GradeScope virtual machine will not do this for you.

What to include in the report

The report should describe experiments you did, apart from the GradeScope environment, on, for example, a reserved VCL platform or your own laptop/desktop with no other applications running. You should report the machine architecture (processor type and speed, size of cache, size of memory), operating system, language, and compiler (version). Inputs to your experiments should include random lists of integers, sorted lists, and lists in reverse order. If you're having fun and have time on your hands, you can play around with other types of inputs (nearly sorted, "organ pipe", etc.).

The smallest lists in your experiments should be large enough for all of your programs to run for at least 10 milliseconds. You can use smaller lists to see where merge sort starts outperforming insertion sort with respect to number of comparisons. At least eight sizes should be included and

[2] Do *not use the merge sort implementation described in the textbook. It will lead to an incorrect count of the number of* key *comparisons because of its use of "sentinel values". Keep in mind that, in any future homework or test questions about the number of merge sort comparisons,* **only the true key comparisons should be counted.**

each size should be twice the previous one. For example, the sizes n might be 0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.4, and 12.8 million, i.e., range from one hundred thousand to roughly 13 million.

You should do some preliminary experiments to determine a good range. On my MacBook Air with
2.2 GHz Intel dual-core i7, 8 GB 1600 MHz DDR3 memory, 256 KB L2 cache, 4 MB L3 cache,
I checked out two extremes to give an idea.

- a list-based Python implementation of merge sort takes about 50 milliseconds to sort 10,000 integers; and about 10 seconds to sort 1,280,000.[3]
- the C qsort utility takes 14 milliseconds to sort 100,000 integers and 2.5 seconds to sort 12,800,000.

Obviously, you won't be able to run insertion sort on the larger inputs – it belongs in a separate category and may require a sequence of smaller sizes to compare with merge sort alone. So runtimes will range between roughly 10 milliseconds and a few seconds, perhaps a few minutes.

Since times and numbers of comparisons will vary from one input to another, you should do at least 10 runs for each implementation and size and report averages (and standard deviations if these are large). The total number of runs you do, after preliminary experiments, would be at least 10 (number of runs) times 8 (number of sizes) times 3 (number of implementations other than insertion sort) times 3 (types of inputs), or 720. Additional 16 runs will be needed to compare insertion sort and merge sort on smaller inputs. You should be able to write one or more scripts that do these in batch. When you are gathering runtime and comparison statistics and have already verified correctness you can ignore the output. On unix-based systems you can do, for example:

```
$ ./random_integers.py 10000 2 | ./merge_sort.py > /dev/null
runtime,8694
comparisons,120405
```

Here `random_integers.py`, which will be posted as a resource, is a script that produces input in the required format. The first argument is the number of integers, the second is a seed for the random number generator (you can use seeds 1–10 for simplicity).

The report should address the following issues:

1. the growth rate in number of comparisons and how accurately this is predicted by the worst-case theoretical analysis;
2. the growth rate in runtime;
3. the relationship between runtime and number of comparisons;
4. the impact of different types of inputs on runtime and number of comparisons

For (1) you could have a table or chart (or both) that shows the *ratio* between actual and predicted comparisons. For (2) you might want to show the ratio between runtime and $n \lg n$. For (3) you can show time per comparison. And for (4) you can have a table/chart with rows/lines representing each type of input. A PowerPoint presentation illustrating the good, the bad, and the ugly when it comes to tables and charts is posted as

`tables_and_charts.pptx`

Pay attention to the comments on each slide. You will lose points for charts and tables that are not of high quality.

The report should end with a summary of your observations. Were there any surprises? How well, in your opinion, does theory reflect reality? How good is the sorting utility of the programming language? What did you learn in the process of creating these implementations and running these experiments.

[3]A version that counts comparisons takes about 90 milliseconds and 15 seconds, respectively. The extra time appears to be mostly due to calling a comparison function.