

Runsor

Thore Husfeldt

Revision 7cca766 , Tue Feb 13 22:16:05 2018 +0100

Sorting by runs

This exercise asks you to implement a sorting algorithm that we call *runsor*. The main goal of the exercise is to expose you to the details of correctly implementing a nontrivial sorting algorithm. This includes careful index manipulation, special cases, and memory management. The algorithm shares many features with *bottom-up mergesort*, and you are welcome to re-use as much of the code in our textbook¹ as you deem useful, in particular from Section 2.2.

¹ R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011.

A *run* in a subsequence is a maximal sequence of nondecreasing, consecutive elements. For instance, the runs in RUNSORTEXAMPLE are RU, NS, ORT, EX, AMP, L, and E. There are two runs in Z00, namely Z and 00. A sequence is sorted if and only if it consists of a single run.

Runsor works in rounds. Each round merges pairs of neighbouring runs. (If the number of runs is odd, the last run is not merged.) Each round divides the number of runs by 2, so after $\sim \lg N$ rounds, the process stops. See Fig. 1.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	R	U	N	S	O	R	T	E	X	A	M	P	L	E
first round:														
merge RU and NS	N	R	S	U	O	R	T	E	X	A	M	P	L	E
merge ORT and EX	N	R	S	U	E	O	R	T	X	A	M	P	L	E
merge AMP and L	N	R	S	U	E	O	R	T	X	A	L	M	P	E
“merge” E (with the empty run)	N	R	S	U	E	O	R	T	X	A	L	M	P	E
second round:														
merge NRSU with EORTX	E	N	O	R	R	S	T	U	X	A	L	M	P	E
merge ALMP with E	E	N	O	R	R	S	T	U	X	A	E	L	M	P
final round:														
merge ENORRSTUX with AELMP	A	E	E	L	M	N	O	P	R	R	S	T	U	X

Figure 1: Trace of runsor

Requirements

Write an implementation of runsor in Java or Python. Follow the conventions used in the textbook:

1. Write a class `Runsort` that includes a method `public static void sort(Comparable[] a)`.
2. You must implement the above algorithm, in particular its way of manipulating arrays as arrays and not as lists.
3. You are not allowed to use more than linear extra space.
4. Your implementation should sort the lines of `stdin` lexicographically. For your convenience, there are some test cases on codeJudge. If your code does not pass the tiny test cases, it is considered wrong.

Tips and comments

1. As always, do not reinvent the wheel, for your own sake as much as ours. For instance, I used the merge method from Section 2.2 in the textbook without modification, including the neat trick with a static `aux` array for auxiliary storage. When reusing code, remember to be open about it (for instance, add a comment like “from Sedgewick and Wayne, Section 2.2.” to your code).²
2. I solved this exercise using 30 lines of my own code, and maybe 30 lines from `MergeBU`. Most of the time I spent on hunting annoying index mistakes.
3. Test your code on your own machine using the client in Example in Section 2.1. Make sure you’ve run your sorting algorithm on `tiny.txt` and `words3.txt`, but also on some pathological cases: empty input, 1-letter input, 2-letter input, etc. Also, sort something really big. If everything looks good, give it a try on codeJudge.

² Indeed, one of the goals of this exercise is to motivate you to understand what’s happening in the book’s merge-sort code.

Bells and whistles

Implement at least one of the following fancy extensions.

1. *Insertions sort to avoid short runs.* Never merge runs that are shorter than 8 elements. If the current run length is less than 8, use insertion sort to sort the next 8 positions.³
2. *Mirror down-runs.* When scanning for runs, also check *decreasing* runs (such as `ZMFCBA`). Whenever you find a decreasing run, revert it in linear time. This trick makes the algorithm run much faster on almost-reverse-sorted input.
3. *Galloping.* Read Peters⁴ or Wikipedia⁵. Good luck!
4. Draw a visual trace of runsort for each round in the style of the book, e.g., like the “Visual trace of shellsort” figure in Section 2.1. (`StdStats.plotBars` does most the work for you.)

³ There’s nothing magical about 8. In fact, larger values like 32 or 64 are probably better. But then the whole algorithm becomes hard to debug, because most of your test inputs will be too short to ever use the merge step.

⁴ Tim Peters, *timsort.txt*, bugs.python.org/file4451/timsort.txt. Retrieved 11 March 2014.

⁵ Timsort, *Wikipedia*, *The Free Encyclopedia*. Wikimedia Foundation, Inc. Retrieved 11 March 2014.

Deliverables

1. The source code for Runsort in a file called `Runsort.*`. Make sure it's perfect, following the book's conventions for clarity, brevity, indentation, variable names, comments, etc.
2. The source code for your extension, in a file called `FancyRunsort.*`.
3. A brief report in PDF, telling us if the code seems to sort, what you've tested it on, and a brief description of your extension. You may use the report skeleton on the next page.
4. Any input files that gave your sorting algorithm trouble.

Further reading

Runsort sometimes called of polyphase merge sort⁶ or natural merge sort⁷. With most of the ideas mentioned above, and a few more due to Tim Peters, it is used in Python (since version 2.3) and Java (since SE 7), often called Timsort.

⁶ Polyphase merge sort, *Wikipedia, The Free Encyclopedia*. Wikimedia Foundation, Inc. Retrieved 11 March 2014.

⁷ Sedgewick and Wayne, exercise 2.2.16

Runsor Report

by Alice Cooper and Bob Marley⁸

Tests

We have run our sorting algorithm on the following inputs from the book; all were sorted: `tiny.txt`, `words3.txt`, [...]. The `TestTiny` tests on codeJudge were solved correctly. Our implementation exceeds the timelimit on codeJudge for the test case `Test06rnd102400`. On codeJudge, our algorithm is twice as fast on the `*run*` in comparison to the `*rnd*` instances.

We also ran our sorting algorithm on `cannotsort.txt` (included). The algorithm fails on this input, but we cannot find the error in our code.⁹

Extensions

¹⁰

Our implementation switches to insertion sort for sequences of length at most 8. We performed test with and without this extension for random inputs of size 1,000,000, but we were not able to detect any improvement in running time. ¹¹

Our implementation mirrors down-runs. This gives a significant improvement on inputs that [...]. For instance, the improved code sorts an input consisting of [...] in time [...], whereas the original implementation used [...]. ¹²

We attempted to understand *galloping*, and have implemented the following idea: [...]. The result was great on `somefile.txt` (included), but we weren't able to detect any improvement on other inputs. ¹³

The figure below shows a visual trace of runsort: [...].

⁸ Complete the report by filling in your names and the parts marked [...]. Remove the sidenotes in your final hand-in.

⁹ Edit or remove this paragraph if you were unable to find an input where your sorting algorithm failed.

¹⁰ In this section, briefly state which extension you chose to implement, and (if it makes sense) report on a single, simple experiment.

¹¹ Remove or edit as necessary. Perform the empirical test and report the result. If you want, try with various sequence lengths and various input lengths. Maybe report the running times in the form of a graph or table. Or write "Unfortunately, with this extension, our code no longer sorts correctly and we are unable to find the error."

¹² Remove or edit as necessary.

¹³ Remove or edit as necessary.