CSc 345 — Analysis of Discrete Structures
Fall 2024 (McCann)

http://cs.arizona.edu/classes/cs345/fall24/

Program #3: Columnsort

Due Date: October 31st, 2024, at the beginning of class

Overview: Columnsort is an algorithm originally designed to be implemented as a parallel algorithm (that is, as an algorithm whose effort can be spread across multiple processors). For this assignment, we will treat Columnsort as if it were a normal, garden–variety internal sorting method.

Rather than provide an algorithm description in my own words, I'm attaching a page from a paper from Dartmouth that gives a pretty good description of the algorithm. No, it's not a nice, clear, processed description; it's an algorithm description from a research paper, but a pretty clear one, as such things go. If you spend a little time creating a small example and working it through by hand, I'm confident that you will be able to figure out how to perform its steps.

As you examine the algorithm, you will note that Columnsort spends a lot of time sorting sections of the sequence (the columns). You will need to employ an auxiliary sorting method for Columnsort to use to sort the columns. You may use any sorting algorithm you wish to sort the columns, so long as you implement it yourself. (That is, you can't use any of Java's API sorting methods, you can't download a sorting class from the 'net, you can't get one from a friend or ChatGPT, etc.)

You will also notice that Columnsort needs to select its own quantities of rows and columns based on some simple constraints. You'll need to implement an algorithm to select appropriate values for r and s based on the number of items to be sorted. Note that Columnsort will work with $r = N$ and $s = 1$, but that there would be a lot of wasted work.

There are other decisions that you will have to make as you create your implementation. We will not suggest any solutions; you'll have to decide on your own how to handle things.

Assignment: Implement Columnsort, and measure its performance on a collection of `Integer` objects whose content (elements of a sequence of four–byte integers) will be supplied in a file named on the command line. The goal is to get your implementation of Columnsort to sort the given collection of `Integer` objects in as little time as possible. To measure the elapsed time, use `System.nanoTime()`. Note that you want to time only the sort itself, not the reading of the file, nor the production of the output. Place your 'start the stopwatch' and 'stop the stopwatch' actions just before and just after the call to Columnsort.

Input: Write your program to accept the name of a data file on the command line: `java Prog3 prog3a.dat` is a sample invocation.

I have linked to the class web page two files of integers, named `prog3a.dat` and `prog3b.dat`. These will be the only sample data files we will supply; we're assuming that you can make your own files of integers for additional testing without too much trouble. The format of a data file is straight–forward: One integer per line. The integers will be standard four–byte integers, and can be negative, zero, or positive. If you detect that the data file contains anything other than integers, terminate the program after displaying a helpful error message.

The two sample data files are of significantly different sizes: The first is very small, and the second is much larger (though not huge by any measure). When we grade, we'll use files of integers with a variety of other characteristics as well; please test your code thoroughly. Exchanging sample data files with other students is acceptable; exchanging source code is, of course, a no–no.

(Continued ...)

Output: Have your program output (to the screen) the quantity of values (n) being sorted, the program's choices of r and s, the elapsed time required to complete the sort (in seconds, to three decimal places), and the sorted sequence of values, one per line. Use this fancy output format:

```
$ java Prog3 somedata.dat
n = 20
r = 10
s = 2
Elapsed time = 0.001 seconds.
1000
1000
1001
[...]
```

Note that the output can be easily redirected to a file; e.g., `java Prog3 prog3a.dat > results`

Hand In: On the due date, submit your completed, well–documented and well–tested program file(s) to the class submission directory, using the `turnin` command on lectura. The turnin location for this assignment is `cs345p3`. Name your main program source file `Prog3.java` so that we don't have to guess which file to compile/translate, but feel free to split up your code over multiple files.

Want to Learn More?

- Keen to see the paper in which Columnsort first appeared? Here's the reference: Leighton, Tom. "Tight Bounds on the Complexity of Parallel Sorting." Proceedings of the 16th Annual ACM Symposium on Theory of Computing (STOC), 1984, pp. 71-80. It's available from the ACM digital library: http://doi.acm.org/10.1145/800057.808667.
- The attached description of Columnsort is a page from an early version of this paper: Chaudhry, G., Cormen, T., and Wisniewski, L.F. "Columnsort Lives! An Efficient Out–of–Core Sorting Program." Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, 2001, pp. 169 - 178. It's also available from the ACM digital library: http://doi.acm.org/10.1145/378580.378627. http://doi.acm.org/10.1145/378580.378627.

Other Requirements, Hints, and Miscellany:

- The use of `nanoTime()` is demonstrated in the demo program `Timing2.java`, which is available on the class web page.
- You may NOT use smaller integer types to try to gain a speed advantage; use Java's four–byte integers in the standard `Integer` wrapper class.
- We insist that your implementation NOT create separate processes, or threads, or farm out work to other machines, etc. All of that would be fun and educational, but parallel/distributed processing is in the domain of other courses. We have plenty of topics of our own to cover, and we're keen to see what you can accomplish without throwing hardware at the problem.
- An expected question: "Can we change Columnsort to improve its performance?" So long as you remain true to the philosophy of the algorithm, yes. Think of Bubblesort; the 'check for swaps' modification creates a variant of Bubblesort, but it's still Bubblesort at the core; the modification is just a tweak. Deciding to implement a different sorting algorithm instead of Columnsort would be far more than a tweak!
- After the assignments are graded, we'll let you know whose Columnsort implementation proved to be the fastest in the testing we performed for grading. No prizes, just intra–class fame, assuming that the 'winner' wants fame. We'll at least report anonymous times.
- Seems like I'm forgetting ... oh, yes! Start early! `:-)`

(Continued ...)

The following is a page from: Chaudry, et. al. "Columnsort Lives! An Efficient Out–of–Core Sorting Program." Proc. of the 13th Annual ACM Symposium on Parallel Algorithms and Architecture, July 2001.

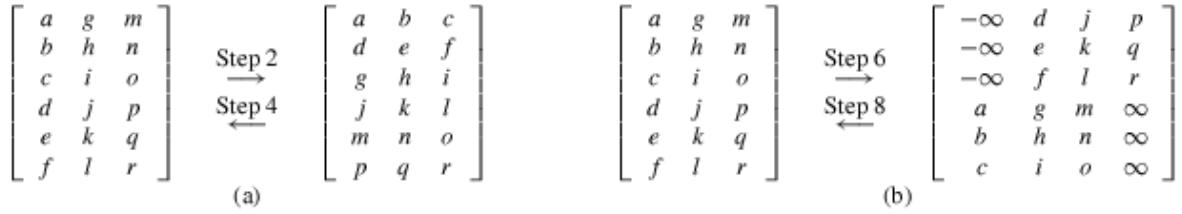

Figure 2: The operations of even-numbered steps of columnsort. This figure is taken from [Lei85]. For simplicity, this small 6×3 matrix is chosen to illustrate the steps, even though its dimensions fail to obey the columnsort restrictions on r and s. **(a)** The operations of steps 2 and 4. **(b)** The operations of steps 6 and 8.

2 The basic columnsort algorithm

In this section, we review Leighton's columnsort algorithm from [Lei85]. Along the way, we make some observations that will improve the out-of-core implementation.

Columnsort sorts N numbers, which are treated as an $r \times s$ matrix, where $N = rs$, s is a divisor of r, and $r \geq 2(s-1)^2$. When columnsort completes, the matrix is sorted in column-major order. That is, each column is sorted, and the keys in each column are no larger than the keys in columns to the right.

Columnsort proceeds in eight steps. Steps 1, 3, 5, and 7 are all the same: sort each column individually. Each of steps 2, 4, 6, and 8 permutes the matrix entries.

Step 2: Transpose and reshape

As Figure 2(a) shows, we first transpose the $r \times s$ matrix into an $s \times r$ matrix. Then we "reshape" it back into an $r \times s$ matrix by taking each row of r entries and rewriting it as an $r/s \times s$ submatrix. In Figure 2(a), for example, the column with $r = 6$ entries $a\ b\ c\ d\ e\ f$ is transposed into a 6-entry row with entries $a\ b\ c\ d\ e\ f$ and then reshaped into the 2×3 submatrix $\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$.

Step 4: Reshape and transpose

As Figure 2(a) shows, we first reshape each set of r/s rows into a single r-element row and then transpose the matrix. In other words, step 4 is the inverse of the permutation performed in step 2.

Step 6: Shift down by $r/2$

As Figure 2(b) shows, we shift each column down by $r/2$ positions, wrapping around into the next column as necessary. This shift operation vacates the first $r/2$ entries of the leftmost column, which are filled with keys of $-\infty$, and it creates a new column on the right, the bottommost $r/2$ entries of which are filled with keys of ∞. Looked at another way, we shift the top half of each column into the bottom half of that column, and we shift the bottom half of each column into the top half of the next column.

Step 8: Shift up by $r/2$

As Figure 2(b) shows, we shift each column up by $r/2$ positions, wrapping around into the previous column as necessary. In other words, we perform the inverse permutation of step 6.

We omit the proof of correctness and refer the reader to [Lei85].

The [Lei85] reference: Tom Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4):344–354, April 1985. `https://ieeexplore.ieee.org/document/5009385`