Applied Algorithms 2024: Final Project

Martin Aumüller and Riko Jacob

Published: November 26, 2024
Strict hand-in date: Friday, 20 December 2024, 13:59
This assignment has 6 pages.

Format of Hand-in This assignment can be solved in groups of up to three people. (You can choose a new group if you want.) Please submit a .zip file on learnIT, containing:

1. your code, including all necessary non-standard libraries and classes required to compile your code, and a script (e.g., `run.sh` or `run.py`) that we can use to re-run your experiments and recreate your plots in the report (or at least to understand how to do this).

2. a single `report.pdf` file containing a report of at most 4 pages written text (**11pt** font, 1in margins, a4paper, normal line spacing, single column — ideally use the template from the course). The sections **must** be called "Part 1", "Part 2", and "Part 3" with subsections for each task. Segments of code **do count** towards the page limit, plots and tables do not, but there must be at most 6 plots or tables in total.

3. a file `authors.txt` containing a list of the group members and email addresses in the following format (and nothing else), using your .itu.dk email address:

```
Martin Aumüller <maau@itu.dk>
Riko Jacob <rikj@itu.dk>
```

Oral exam. There will be an oral group exam on January 7–8, 2025. If you have requests for a specific slot for the oral exam, write email to Riko (rikj@itu.dk). We will announce the precise schedule on January 6. Please be at the exam room at least 15 minutes before your scheduled time. You will be examined by Riko and an external censor.
Each student will be assessed for 20 minutes. At the beginning of the exam, the group presents their project together for roughly 5 minutes. We will have your report in front of us on paper as well as your code on our computer. You can use a whiteboard or paper, or you can show your code in a terminal on our computer. However, you cannot use any kind of slides. After the 5-minute presentation, we will assess each student individually, for 10-15 minutes per student, while their fellow group members (if any) wait outside. We will ask questions about your project, and also about topics from the entire course syllabus (insofar as they are related to the intended learning outcomes). In the remaining time, the examiners discuss the grade, tell you the grade and, if you want, explain it. By default, we will discuss the grade only after having done all the discussions with all the group members. Only after that, we will announce the individual grades to all participants of the group, and be ready to explain the grades. In this way, all the good things the different members of the group can say about the project can be added up, potentially leading to a better grade. If you prefer something else, like getting to hear your grade individually, let us know upfront.
Grading. The grade will be based on the project as well as the oral exam. In the project, we care about the quality of 1) your implementations, their correctness and efficiency, 2) your experiments, their design, execution and reproducibility, and 3) your evaluations, analysis, and reflections in the report.

Part 1: Basic MergeSort

In this assignment, we work with several variants of MergeSort, a stable, comparison based sorting algorithm. MergeSort is comparison based, and your implementations should work for any class that implements the java interface "Comparable". If you are working in a different programming language, similarly make sure that you can run your implementations with different underlying classes that allow comparisons. In this setting, we can't test all possible underlying data types. For each task (or many of them together), choose a meaningful set of data types to test and experiment with, and explain your choice. For all implementations it is implicitly part of the task to check its correctness. Is the output sorted in non-decreasing order? Is the sort stable? Are all input elements in the output and vice versa? For each implementation, describe one bug that can be caught with your testing framework (the more subtle the bug, the better). In general, if you are running out of time, focus on solving particular tasks well instead of providing incomplete solutions to many tasks. We will discuss this assignment in the lecture on November 27. The slides of that lecture expand on the explanations here. If you have questions, send them to Riko (rikj@itu.dk) or Martin (maau@itu.dk), and expect answers in the forum on learnit.

Task 1 Implement classical MergeSort recursively. Any variant is fine. One obvious place to start is the detailed description in Chapter 2.2 of Sedgewick and Wayne "Algorithms" [3], "Top-down MergeSort". We suggest that you stick to the paradigm of abstract in-place merging as in Chapter 2.2 of [3]. Instead of the functions not returning anything, make them return the total number of comparisons performed.

Task 2 Design and perform an experiment that investigates if the running time of an implementation is proportional to the number of comparisons. How does this relate to the theoretical running time of $O(n \log n)$? Does the content of the input array influence the running time or the number of comparisons? What influence does the type of objects have? Try with the objects being strings of the same length. Vary that length and include the case where all strings have a common prefix.

Task 3 Implement MergeSort with a base case of insertion sort: For a parameter c, instead of calling MergeSort recursively on a sub-array of length at most c, call InsertionSort on the sub-array.

Task 4 Design and perform an experiment to find a good value of the parameter c.

There are different ways to implement an iterative variant of MergeSort. Here, we use a variant that is based on the framework of TimSort (or PowerSort) [1]. One important building block for these algorithms is a run: a sub-array that is already sorted. For now, we use this framework without determining already sorted subsequences in the input.

Task 5 Implement MergeSort iteratively with a stack of runs: Keep a stack of already computed runs (leftmost is bottom of the stack). If the topmost two runs on the stack have the same length, merge them and place the result back on the top of the stack. Repeat this. Otherwise, turn the next element into a new run of length 1. Finally, if you reach the right end of the array (and you can't create new runs), repeatedly merge the two topmost runs on the stack until only one run is left that contains the final result.

Observe that all runs on the stack have as length a power of two.

Task 6 In the algorithm of Task 5, create new runs of length c by taking the c next elements and sort them using InsertionSort. If there are less than c elements remaining in the array, turn them into one new run using InsertionSort.

Task 7 Rerun your experiment of Task 4 for your iterative MergeSort. Compare your results for the recursive and iterative implementations.

Part 2: Different merging rules in TimSort

TimSort is an adaptive sorting algorithm that makes use of the input being already partially sorted. See [1][1] for a detailed description, in particular the pseudocode Algorithm 2 on page 63:10. The main idea is to make use of already sorted subsequences in the input - also called runs, just as much as parts of the input array that are sorted as a result of merging. When we explore a run from a given position i, we find the longest non-decreasing subsequence starting at i. If we instead find a long strictly decreasing subsequence starting at position i, we turn it into a run by reversing it in-place. (Observe that the strictness avoids trouble with the resulting algorithm being stable.) Now, the number of comparisons required to sort can be much smaller if there are many long runs in the input. We say that the algorithm is adaptive to the pre-sortedness of the input.
The basic framework of these algorithms is that of Task 5, with different rules on how to choose which runs to merge. All the variants merge runs (somehow) at the top of the stack of runs. In contrast to the original TimSort (and PowerSort), here, we do standard merges (no galloping). With the next few tasks, we explore different merging rules:

1. A variant of the PowerSort merging rule called **LevelSort**.
2. An alternative rule based on run length, called **BinomialSort**.

These rules give rise to a *merging tree*, where the original runs correspond to leaves of the tree, and an internal node stands for the merge of two runs. In particular, the root of the tree stands for the final merge that creates the output. Observe that any internal node of the merging tree corresponds to a boundary of runs in the input — even though the actual merge can range from two neighboring input runs alone (a node that has two leaves as children) to the final merge that produces the output (the root node) or anything in between.
In **LevelSort**, every boundary between runs is assigned a level, leaves have small level, nodes/boundaries closer to the root of the merging tree have high level. (Watch out: This is opposite to the Power described in [1], where the root node has power 1). The level of a boundary depends on the two runs in the input that are joined by this boundary. Let i_a, i_b, i_c be the boundaries of the two runs with boundary i_b, i.e., the runs are stored in the positions $i_a, \ldots, i_b - 1$ and $i_b, \ldots, i_c - 1$. We define the midpoints of these two runs as $m_l/2$ for $m_l = i_a + i_b$ and $m_r/2$ for $m_r = i_b + i_c$. The **level** of this boundary is the most significant bit in which m_l and m_r differ. Assuming we have stored m_l and m_r in longs (to avoid overflow when adding two integers), in java the level is `64-Long.numberOfLeadingZeroes(ml^mr)`. Observe that two consecutive boundaries never have the same level.
The level of a run is the level of its right boundary. We keep the invariant that the levels of runs on the stack are monotonic, the smallest level at the top, the largest level at the bottom. Besides the stack, the algorithm works with two runs: the left run L and the new run N. This allows us to compute the level l of the boundary between these two runs. While the level of the topmost run is $< l$, we pop the run of the stack and merge it with L and call this the new L. Now we place this merged L onto the stack with level l. This establishes the invariant. Finally, the run N becomes the left run L. See also the pseudocode [1], Algorithm 2.
The variant **BinomialSort** is based solely on the length of the runs. The invariant is that the length of runs on the stack increases by at least a factor of 2 from run to run as we go from top to bottom in the stack. When a new run is about to be placed on the stack, we repeatedly merge the two topmost runs until the topmost run has length at least the length of the new run. Then, while the topmost run on the stack has length less than twice the length of the new (merged) run, we merge the topmost run on the stack into the new run. Finally, we place the new (merged) run on the top of the stack.

[1]https://www.wild-inter.net/publications/munro-wild-2018.pdf

The name is inspired by binomial heaps, where we maintain the invariant that for each rank, there is at most one binomial tree of that rank. Observe that the algorithm of Task 5 has a similar property: When we are done merging, the length of the runs on the stack get smaller as we get closer to the top of the stack and there are no two runs with the same length. All run-lengths are powers of two, and the binary representation of the number of already touched elements is reflected in the current run sizes. Adding another run of length one is like incrementing that binary counter. These invariants are not quite true for BinomialSort, but it is similar enough in spirit to justify the name.
Both LevelSort and BinomialSort have an adaptive and a non-adaptive variant: In the non-adaptive variant, we always create a new run of length c (but at the end of the array) by blindly using InsertionSort on the next c elements. In the adaptive variant, we explore how long the next non-decreasing or decreasing run actually is. If it is longer than c, we use it, otherwise we take the next—c elements and turn them into a run by using InsertionSort.

> **Task 8** Implement the four variants (LevelSort vs BinomialSort and with or without run exploration) of the algorithm, including the parameter c for the initial run length (cut-off).

> **Task 9** Design and conduct an experiment to investigate the influence of the parameter c and the pre-sortedness of the input on the running time and number of comparisons. Perform this experiment for the four variants and discuss the following questions:
>
> 1. How should you choose the parameter c? Are there differences between the variants?
>
> 2. What differences in adaptiveness do you see between the algorithms?

> **Task 10** Include InsertionSort, plain MergeSort and `Arrays.sort()` (even though it is not stable) into the comparison. How do the algorithms compare in a horserace?

To pick a diverse set of input distributions to study the adaptiveness of the different implementations, we suggest that you take a look at Pages 10 and 12 of [2], available at `https://sci.utah.edu/~beiwang/publications/PersiSort_BeiWang_2024.pdf`.

Part 3: Parallelization

Turning away from adaptive MergeSort, we are now focusing our attention on parallel versions of MergeSort.

> **Task 11** Implement a parallel version of the recursive MergeSort (from Task 1), where the two recursive calls are performed in parallel. Use the recipe of the `ForkJoinPool` from Lecture 11. Include a parallel-cut-off parameter: if the recursive call happens on an input smaller than that parameter, recurse serially. Make the implementation return the overall number of comparisons by this being the return value of your recursive function.

One correctness / consistency check for this task is to compare the number of total comparisons to the sequential version.

> **Task 12** Design and perform an experiment to judge potential speed-ups and to find a good choice of the parallel-cut-off parameter.

To be considered for a 10, solve the following tasks:
As a building block for parallel merging, the function twoSequenceSelect() plays an important role.

Input: Two sorted sequences a and b (to be merged), and an integer k, a position in the output array ($0 \leq k < L(a) + L(b)$ where $L()$ is the length).

Output: The positions i_a and i_b that sequential merge would have before placing the k-th element $\min(a[i_a], b[i_b])$ into the merged output at position k.
Observe that $i_a + i_b = k$. The correct answer has the following properties (we assume $a[-1] = b[-1] = -\infty$):

$$a[i_a - 1] \leq b[i_b], \quad b[i_b - 1] < a[i_a],$$

and these two conditions identify i_a and i_b.
Equipped with this insight, we can perform a binary search to find i_a and $i_b = k - i_a$. For a candidate j_a ($j_b = k - j_a$), if the first inequality is violated, j_a is too large, if the second is violated, it is too small.

Task 13 Implement twoSequenceSelect() with binary search. Don't forget to show test-cases that your implementation is correctly identifying the positions in a and b that are used in the sequential merge.

Task 14 Implement a parallel version of merge by splitting the output array into equal parts. Use twoSequenceSelect() to jumpstart the serial merge that produces the intended part of the output. Use the recipe of the `ForkJoinPool` from Lecture 11, with Futures. Base it on a parameter p for the number of parallel tasks to create. Make the task return the number of comparisons executed.

Task 15 Implement a parallel recursive MergeSort, where the recursive calls happen in parallel, and the merge uses the parallel implementation.

Task 16 Design and conduct an experiment that investigates the scaling behavior of your implementations when the number of available threads changes. Include a comparison to `Arrays.parallelSort()`.

To be considered for a 12, solve the following task:
In parallel implementations of MergeSort, we can consider the "span of comparisons", for short "span": What is the longest chain of comparisons, that were executed serially by an implementation? In effect, this means that when there are several parallel calls to functions, the span is the maximum of the spans. The span of a function is then the sum of whatever happened serially in that function, where some of the terms might be maxima of parallel calls.

Task 17 Implement variants of the algorithms that compute the span (by returning the span of comparisons instead of the total number of comparisons). Design and perform an experiment that discusses if the span is a good predictor of the parallel running time of the different implementations.

References

[1] J. Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In *ESA*, volume 112 of *LIPIcs*, pages 63:1–63:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[2] Jens Kristian Refsgaard Schou and Bei Wang. Persisort: A new perspective on adaptive sorting based on persistence. In *Proceedings of the 36th Canadian Conference on Computational Geometry (CCCG)*, 2024.

[3] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition.* Addison-Wesley, 2011.