Applied Algorithms 2020: Final Project

Martin Aumüller and Riko Jacob

Published: November 16, 2020
Strict hand-in date: December 18, 2020, 13:59

This assignment has 3 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.) You yourself are in charge of creating the group by filling out the front page in learnIT. When submitting in learnIT, you need to add the other members of the group. Only one member of the group can upload the exam assignment in learnIT. Please submit a .zip file on learnIT, containing:

1. your code in subdirectories `part1/` and `part2/`, 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.

2. a single `report.pdf` file containing a report of at most 4 pages written text (**11pt** font, 1in margins, a4paper, normal line spacing). The sections **must** be called "Part 1" and "Part 2" with subsections for each exercise. Plots, tables, and segments of code **do count** towards the page limit.

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

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

Oral exam. There will be an oral group exam on January 13–15, 2020. 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.

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.

1 Part 1: Speeding up Binary Search using Lookup Tables

In this task, you implement a binary search data structure in two different versions, and experimentally compare them. Upon its construction, the data structure is given a set S containing n distinct, positive 32-bit integers. It should then support queries of the form

$$\texttt{Pred}(x) = \max\{y \in S \mid y \leq x\}\,.$$

That is, the data structure needs to have a method `Pred` that, on input x, returns the largest integer y with $y \in S$ and $y \leq x$.

Task 1. Implement two different versions of binary search.

1. `SortedArray`: The elements of S are stored in single, sorted array `A`, and queries are solved using a standard binary search.
2. `SortedArrayWithTabulation`: For a parameter $k \in \mathbb{N}$, create an array `table` with 2^k entries. Each entry is a pair of integers `(left, right)`. For a k-bit integer x with $0 \leq x < 2^k$, `table[x] = (left, right)` means the following. Let $\mathcal{I}_x$ be the set of all integers where the top-k bits represent the integer x. Then all answers to queries Pred(y) for $y \in \mathcal{I}_x$ can be found in `A[left..right]`. This means that we can use `table` to potentially speed up the binary search by using a good precomputed starting interval for the search.

   Given $x \in \{0,1\}^k$, good bounds on `left` and `right` can be found as follows. Let $x' = (x_1, \ldots, x_k, 0, \ldots, 0)$, i.e., the smallest 32-bit number that has the top-k bits set as x. Analogously, we let $x^* = (x_1, \ldots, x_k, 1, \ldots, 1)$ be the largest such number. We may set `left` as the index in the sorted array `A` that is found by the search $\texttt{Pred}(x')$, and may set `right` as the index found by the search for $\texttt{Pred}(x^*)$. (Alternatively, one left-to-right scan through `A` is sufficient to fill out the table.)

   For a query integer y, we can use its top-k bits to find an initial boundary that is used for the binary search, which potentially could speed up the search.

We provide an automatic test for your implementations on CodeJudge. Each of the two programs should read the following format on standard input:

```
4
93 1094 11 13
93 12 9999 1 18
```

The first line contains the size n of the set S and the second line contains the set S in some unknown order. The third line contains queries x to `Pred` – let us call the number of queries q. To be clear, each query must be answered using an individual call to the function `Pred` in the respective data structure (i.e. a solution using a different algorithm to give the correct answer will not be acceptable, even though it might pass all tests on CodeJudge). In the end, the programs should print the following on standard output:

```
93 11 1094 None 13
```

Task 2. Design, run, and evaluate an experiment comparing your two implementations with respect to the average running time per query.

In particular, remember to describe your experimental design using the proper terminology, describe and explain your a-priori hypothesis for what the experiment will show, describe and evaluate the results of your experiments, use plots with axis labels and units to visualize the obtained data, and explain in how far your hypothesis was supported by the findings. Remember to describe how your input is generated, and to use a large n and even larger q. Include a plot that shows for fixed large n and q the relationship between the parameter k and the average query times for `SortedArrayWithTabulation`.

To be considered for a 12, you have to:

1. describe input and query generation such that `SortedArrayWithTabulation` benefits most/least from the precomputation, and

2. compute the entries of `table` by inspecting each entry of `A` only once. To accomplish this task, you can carry out one left-to-right scan through `A`. If two neighboring entries `A`$[i]$ and `A`$[i+1]$ with $0 \leq i < n-1$ differ in their top-k bits, fill out the corresponding table entries that have their `right` boundary at i.

2 Part 2: Implementing Quicksort

Sorting an array of n elements of primitive types (`int[]`, `float[]`, ...) in Java uses a Quicksort algorithm, since around 2009, a *two pivot* variant. In this task, you are to implement three different quicksort variants and experimentally compare them. The basis for these implementations are the papers "Average Case Analysis of Java 7's Dual Pivot Quicksort" by Wild and Nebel [WN12], and "Multi-Pivot Quicksort: Theory and Experiments" from Kushagra et al. [K+13] (see learnIT).

Task 3. Implement the following three variants of Quicksort to sort an array of 32-bit integers:

- `ClassicQuicksort`: Implement Algorithm 1 in [WN12].
- `TwoPivotQuicksort`: Implement Algorithm 3 in [WN12].
- `ThreePivotQuicksort`: Implement Algorithm A.1.1 in [K+13].

Task 4. Test the correctness of your implementations. In your report, describe how many cases you used, what input generator you used, how you determined that the output was correct, and whether you ran your tests manually or in an automatic fashion.

To be considered for a 12 the following discussion is necessary: Algorithm 3 in [WN12] and Algorithm A.1.1 in [K+13] provide certain boundary checks. Remove the check $k < g$ in Line 11 of Algorithm 3 and $b \leq c$ in Line 6 of Algorithm A.1.1. In each case, provide an input that shows that the implementation is not correct or argue why the check is not necessary.

Task 5. Design, run, and evaluate an experiment comparing the average running time of sorting inputs across the three different implementations and for at least three different types of inputs. The paper [K+13] contains inspiration for different input types, but think about other interesting cases. Note that n has to be rather large for the experiments to be meaningful (e.g., much larger than the size of processor caches). Report the average running times in a plot for each combination of implementation and input type. Additionally, compare the fastest of the three implementations against the standard sorting algorithm in the programming lanuage of your choice (for example, `Arrays.sort()` in Java or `.sort()` in Python).

To be considered for a 12, at least one of the two following approaches has to be implemented and evaluated:

1. The paper `https://www.wild-inter.net/publications/wild-nebel-reitzig-laube-2013.pdf` says that it is beneficial to sort small arrays with InsertionSort instead of carrying out Quicksort. Design, run, and evaluate an experiment to find optimal array sizes for InsertionSort for each of your three implementations. Compare your running time results to your original implementation.

2. The 3-pivot algorithm described in Algorithm A.1.1 in [K+13] uses double swap operations to make room to accomodate a single new element. This can be made more efficient by using a cyclic rotation/shift operation.[1] Implement the 3-pivot algorithm by avoiding double swaps and using cyclic shifts. Compare the running time of these two variants with each other.

[1] The animation on Slide 18 of `http://pkqs.net/~tre/talks/thesis-defense.pdf` might help you in understanding the general idea behind a cyclic rotation.