Lab 6: Optimization
12:00 PM, Feb 26, 2020

Contents

Objectives

By the end of this lab, you will be able to:

- implement a nice algorithm for quicksort
- optimize an algorithm for better time and space complexity
- find and implement potential optimizations

Before We Start: Complete A Survey

Before you start today’s lab, Kathi would appreciate it if you each could fill out this survey: https://algo-limesurvey.uni-muenster.de/limesurvey/index.php/284516?lang=en The survey should be completed individually.

Setup

Task: Before beginning this assignment, copy the source code to the ‘lab06’ directory in your javaproject:

```
cp /course/cs0180/sol/lab06/sol/* /course/cs0180/workspace/javaproject/sol/lab06/sol/
```

You will be referring to this source code throughout the lab. It contains:

- QuickSorter, a concrete class that implements the most basic sorting algorithm using quicksort. You will be optimizing the algorithm based on the sort method we provide.

Do not alter this file!

You will be implementing optimized versions of the sort method using quicksort in new classes QuickSorterFast and QuickSorterInplace, and you will be analyzing time and space comparisons between different versions.

Note: For the simplicity of this assignment, you will only be working on lists of integers, but it can be easily extended to lists of any comparable generic type T.
The Mystery Continues...

After all of your hard work helping Lena organize the Ticket system, the evening of the CIT formal is finally here! You’re excited to hang up your magnifying glass as the resident CIT detective for a night and have fun with your friends. You walk through the lobby doors into the first floor common area (looking very dapper, if you do say so yourself) and are instantly taken aback — you hardly recognize it with all the decorations and lights. Through the Sunlab doors, you think you can even see a sliver of the ACTUAL setting sun (did they add a window in there? Is the CIT even structurally capable of having windows?) Even though it looks amazing, something’s off. All you can hear is the hum of the Sunlab computers — where is the music? In the corner, you can see your CS 18 TAs Shenandoah and Isaac frantically clicking buttons on a fancy looking audio system. Curious, you head over to check it out.

The two are deep in conversation but look up when they see you approaching. You ask them what’s going on and Shenandoah tells you that there’s no music right now — not only does the sound system not work, but they don’t know what to play. Isaac sheepishly explains that a week ago, they sent everyone a form to ask what music they’d like to hear at the formal. They had intended on creating a music recommendation system, but because of their CS homework, they ran out of time. There’s no time to implement an entire recommendation system now (that would take at least two weeks, or something...), so you all brainstorm a quick and easy solution: you can sort the songs by how many people requested them and play the most popular ones.

Shenandoah and Isaac throw something together but they can tell from running their tests that it’s way too unwieldy to sort the thousands of responses. They still have to fix the broken sound system, so they task you, the resident CIT detective, with optimizing it. Being the helpful TAs that they are, they explain to you that they’ve written an implementation of quicksort to get the job done.

Quicksort

Quicksort is a fast sorting algorithm. Isaac explains that it works by first choosing an element in the given list, called a pivot. Then it moves all items in the list with value less than the pivot before the pivot, and all items in the list with value greater than the pivot after the pivot. At this point, the pivot is in the correct spot for the final sorted list.

Shenandoah continues, telling you that the algorithm then recursively repeats the process on the sub-list before the pivot and the sub-list after the pivot, sorting both. This type of algorithm is called a divide-and-conquer algorithm, because the two halves of the list are being sorted independently, and eventually are simply concatenated.

In short: one element is picked and moved to its correct (sorted) index, then the left half and the right half of the list are also sorted recursively.

Task: Analyze both the time and space complexity of the quicksort algorithm given in QuickSorter.
You’ve reached a checkpoint! Please call over a lab TA to review your work.

## Optimizations

Now that you have an idea of the runtime and the space complexity of the basic quicksort algorithm we gave you, we want you to be thinking about improving it from both time and space.

First, take a close look at the given algorithm and see if you have any ideas on improving its runtime.

**Task:** Optimize the given algorithm to improve the time complexity. Create a new class `QuickSorterFast` and write a new `sort` method in this new class.

After you have improved the runtime for the algorithm, now you should be thinking about improving the space efficiency of the algorithm.

**Task:** Optimize the modified version of the quicksort algorithm to improve the space complexity. Create a new class `QuickSorterInplace` and write a new `sort` method in this new class.

**Hint:** As implied in the name of the new class, consider in-place sorting! Also, use `ArrayList` for the in-place sorting. You might need to write helper methods for this.

**Task:** Analyze both the time and space complexity of this new improved algorithm.

**Task:** Answer the following questions:

- What if we did in-place sorting with a `LinkedList`? What would the new runtime be?
- Compare the time and space complexity between the sorting algorithms in `QuickSorterFast` and `QuickSorterInplace`.

## Analysis

Now that you have finished the coding part of this assignment, it’s time to look back and think critically about what you have done.

**Task:** Answer the following questions:

- Summarize the techniques we used to improve time and space complexity respectively. What did you look for to find those optimizations?
- Are there downsides to the optimized solutions? How do you decide how much to optimize your code?

_The Mystery Continues to Continue..._ As soon as you put the finishing touches on your analysis, Shenandoah and Isaac get the sound system back online. They look over your code and analysis, okay it, and sort all of the songs. It’s a success — the sweet tune of victory starts to blast from the speakers!

Everyone at the formal cheers and the party begins — you’re quickly dragged by your friends onto the dance floor as true bop after true bop fills the room.
Suddenly, a loud shock of electricity crackles through the building. The lights and sound cut off abruptly, plunging the dance floor in darkness and chaos. The room fills with the gasps and murmurs of frantic formal-goers — nobody knows what’s happening, except you. You would recognize this handiwork anywhere — they’re here, they have to be, and tonight is the night you finally bring them to justice.

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)