Practical Documentation CS2001 W10

QuickSort Execution time Analysis.

Student ID: 180023970
Submission Date: 20/11/20

Overview

The aim of this practical was to explore the pathology of the quicksort sorting algorithm. The case is when the pivot happens to be the largest element in the sequence, thus rendering a recursive call useless, as partitioning does not progress sorting.

The specification required a not in-place implementation of the quicksort algorithm with a last element pivot (to set up the case) and an experimental analysis of the pathological case in terms of the execution time differences / trends. Two main questions were to be answered, which are:

1. How sortedness affects sort time for quicksort?
2. How sorted should a sequence be to trigger a significant slowdown?

I have implemented the algorithm as instructed, carried out the experiments and plotted data using Python. I believe I have successfully demonstrated negative relationship between sortedness and execution time (Question 1). As for the second question, I believe I did not collect enough data to support my guess on it. More on conclusions and evidence in Analysis part.

Design

As the aim of the practical was to produce a simple implementation to highlight a trend, not to sort in the fastest way, most of the decisions root from this, to promote better readability and reliability of the solution.

QuickSorter -> sorting algorithm implementation

In search of a pseudocode for a not in-place implementation of quicksort, I have been guided by a code snippet in Haskell, where I realised I could create and fill up two arrays (left & right) and return the result of such. Another influence on my implementation was finding this code for stable quicksort in python, after seen arrays filtering in which I closed it (as it was a complete solution) and wrote my solution to my derived pseudocode:

```
func qsort(List a) -> List

    if a.size() <= 1
        return a

    else:
        lastIndex = a.size() - 1
        pivot = a.get(lastIndex)

        List letfPartition
        List rightPartition

        for (i, i < lastIndex, i++)
            element = a[i]
            if element <= pivot
                leftPartition.add(element)
            else
            /*
             if equal, goes to the right as in the left would
            yield an empty left partition upon next qsort
            partitioning
            */
                rightPartition.add(element)

        List result

        result = qsort(leftPartition)
        result.add(pivot)
        result.addAll(qsort(rightPartition))

        return result
```

I have additionally added a method in the class that would instead of returning a sorted array, return the time of execution in milliseconds.

The choice of ArrayList as a data structure comes from the convenience for the partitioning part of the algorithm.

ListBuilder -> lists generation class & shuffler

I wrote this class for convenient generation of lists of different sizes & shuffling. Additionally, I had a hypothesis that repetition of elements could have an effect on sortedness and sort time relationship. Hence, there are methods for creating a list with no repetition starting from some element, and a method for generating list with repetition of some size.

In the method that generates a random list with repetitions, I had to utilise java.util.random functionality. I have considered that it is a pseudorandom solution (as it takes a seed number that generates a stream of integers/takes current time), therefore in all of method's calls, `Random` object is initialised from start, thus improving chances of true randomisation. Insight acquired from here.

The methods for shuffling do so by picking a pair at random and swapping values. I believe this approach is far less biased and has a lesser chance of producing a fixed pattern, that might undermine findings from the experiments that heavily rely on this functionality.

QuickSortAnalyser -> class for executing the experiment

Careful! Running the main method may take more than 40 minutes☺

This class was written to carry out experiments in an automated fashion, adhering to a fixed methodology.

Overall, there are two methods: one for performing the experiments and collecting data, and another one for transferring it to an output CSV file for further graphical analysis.

The method for experiments and data collection works as follows, it is instructed on the size of lists to be generated as well as if the list would have repetitions or not (for calling corresponding generator). Additionally, there is max shuffleness count passed, repetition of the results for the same shuffleness and repetition for the same list counts. The idea is to run things multiple times on all levels and collect the mean of the runs, as well as min max bounds and a standard deviation for further analysis. This is done to offset fluctuations of run times introduced by the not-ideal testing environment (JVM optimisations, other processes on the personal machine).

SortRunData -> data structure to hold experiment findings

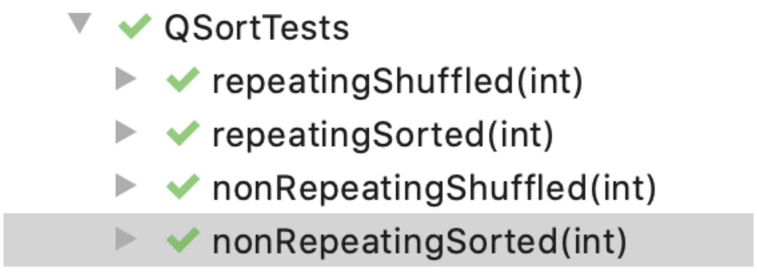
This is a class that would conveniently hold collected data for further transfer to a CSV File. It additionally finds a standard deviation from the array of values, as well as the highest value and the lowest value in the sample, for further graphical analysis.

Testing

To ensure the experiment's data is reliable, there are a few unit tests testing the core functionality of the quicksort implementation and the list generator class.

QSortTests

- Non repeating sorted
- Repeating sorted
- Non-repeating shuffled
- Repeating Shuffled

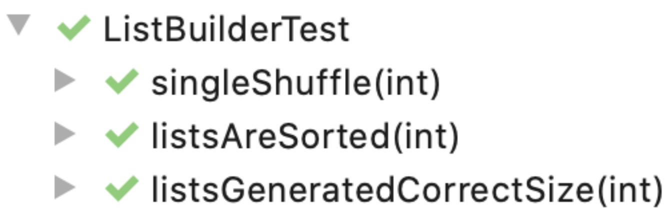


```
▼ ✓ QSortTests
    ▶ ✓ repeatingShuffled(int)
    ▶ ✓ repeatingSorted(int)
    ▶ ✓ nonRepeatingShuffled(int)
    ▶ ✓ nonRepeatingSorted(int)
```

ListBuilderTests

Proper shuffling unit testing was impossible due to the random nature of shuffling, therefore unit tests don't cover it. It was however done manually by printing out and tracing shuffles of elements.

- Lists are of correct size
- Lists are sorted
- Single shuffle shuffles by one



```
▼ ✓ ListBuilderTest
    ▶ ✓ singleShuffle(int)
    ▶ ✓ listsAreSorted(int)
    ▶ ✓ listsGeneratedCorrectSize(int)
```

Methodology

To uncover the relationship between sortedness and the execution time of the quicksort algorithm, I used a metric of the number of random shuffles. The hypothesis is that the more the elements are shuffled, the less ordered the list becomes. A less ordered list would need less time to be sorted, as the partitioning is effective when the pivot is not the largest element (which it is for all sorted lists / sequences of sorted elements). Therefore with more shuffles in theory more sorted sequences are broken down, aiding in partitioning effectiveness.

To produce data for analysing this scenario, the sorting method is run for 0 to 10000 shuffles. For each number of shuffles, the results are reproduced 50 times, and for each list sorted, it is sorted 10 times. Therefore, collecting a mean of individual list sort time and then collecting a mean of mean of lists sorts of this sortedness, there should be a reliable sample to observe the desired behaviour.

The repetition numbers were picked as a knowledgeable guess, aiming to reach an equilibrium between the least time taken for collecting a sample and the sample's data accuracy. For the purpose of former, not all number of shuffles is measured, rather all from 0 to 50 (largest differences), then with an increasing step skipping.

The samples are collected from running on a list of sizes of 10000, 5000 and 3000.

Analysis

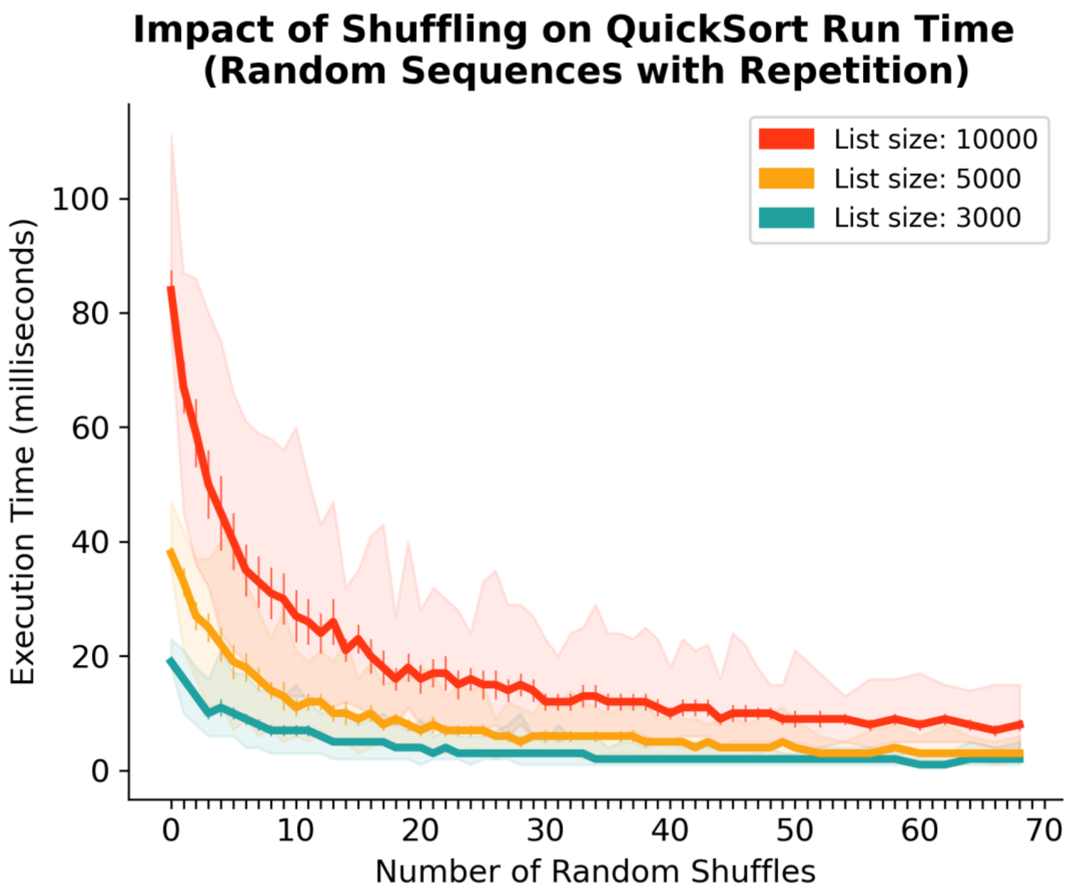


Figure 1

(The graph displays execution times for first 70 entries of the data, to highlight the point of rapid time growth)

The graph (see Figure 1) demonstrates a negative relationship between sortedness and execution time of quicksort in an exponential manner. It holds not only for the mean, but the same trend can be observed considering other data in the samples for each number of sorts. The standard deviation values in form of error bars display identical behaviour, as do the bounding values of the each sample of sortedness, shown in the filled polygonal figure with a decreased transparency.

While the relationship between sortedness and sort time in the pathological case is evident, it is more difficult to make an estimate of how sorted should a list be to trigger a significant slowdown. I believe to address this question, more data should be collected analysing the number and length of sorted sequences in the array.