Assignment 2: Classic vs. Dual Pivot QuickSort

Gustav Gyrst

2021-10-04

1 Introduction

quicksort is a heavily used in-place sorting algorithm. It has an average performance of $O(nlogn)$ and is based on a partitioning structure where a pivot is selected and the subsections are sorted recursively. Until recently, the classic implementation of quicksort with a single pivot has shown to be the most efficient. Sedgewick found that the extra overhead created from dual pivot methods outweighed the benefits. However, the recent dual pivot implementation by Yaroslavskiy has proven to show otherwise when implemented and run in Java JVM. Kushagra et al. (2014) later investigated the reasons for the superior performance of the dual pivot implementation and found that it was not the difference in comparison and swap operations that determined the better performance of Yaroslavskiy's dual pivot implementation, but the significantly fewer cache-misses that the dual pivot approach results in. This papers aims to implement the classic quicksort and Yaroslavskiy's dual pivot quicksort implementation and test the two against one another in Python 3.8.

2 Implementation

2.1 From Pseudocode to Python

The implementation of the classic and dual pivot algorithms follows the pseudo-code implementations presented in the Wild and Nebel's paper (2012). However, some adjustment have been made to successfully convert the pseudo-code into working python code. Both were written and tested in Python 3.8.10. In the **classic quicksort**, the *do while* loops have been handled by creating the variables (counters i, j) that increment before the inner

while loops start, so that i and j increment once before the inner while loops start. Moreover, the outer-while loop is running on the condition while True, and breaks if $j > i$. For the **dual pivot quicksort** an initial check if $A[left] > A[right]$ was added to ensure that a swap of the right and left happens if the statement evaluates to true.

2.2 Testing for Correctness

Both quicksort algorithms were tested for correctness with tests on different list of numbers. Both small (i.e., lists of size 10) and larger lists (i.e., size 100) where used to test the algorithms. The algorithms where tested with simple if-statement tests and the use of a simple is_sorted(list) method, to check that the lists returned by the algorithms indeed were ordered. The algorithms were tested on ordered and randomly created lists (both with positive, negative, mixed integers, and different ranges of random numbers e.g., 0-5 and 0-1000). The sorting algorithms were also tested on a list with one repeated number (e.g., list = [42, 42, 42, 42]) and an empty list. Both were also tested with sentinel values. Despite this being mentioned as an assumption for the algorithms in the paper of Wild and Nebel (2012), the tests shows that the sentinel value does not have any effect on the algorithm when implemented in Python. In Python, the algorithm never reaches and IndexOutOfBounce exception, because the A[-1] will just look at the last element in the list, and not create a call for a none-existing index as would be the case in Java. The sentinel value was set to the lowest 32-bit integer (i.e., -2147483648). Both algorithms were also tested on cases where they were expected to fail (i.e., list mixed of different types: Strings, chars, integers, floats etc.)

3 Experiments

3.1 Setup and preparation

The experiments were run with 10 repetition (i.e., N = 10) of each sample test n to decrease the chances of random fluctuations in the measurements. We conducted the experiment on 30 sample sizes n with the growth in each sample size n determined by: $30 * 1.41^i$. The experiments were made on a Surface book 3 with 1.30GHz 8 cores Intel i7 processor, Memory of 32GiB RAMS. Python 3.8.10 was used. The tests was based on simple time measurements using Python's *Time* module. Moreover, the experiments were run with inputs of randomly generated lists. These inputs were created using python's

randint() function from the *Random* module, with 32-bit integers (i.e., integers ranging from -2147483648 to 2147483648). The same input was used for all tests, and a local auxiliary list was made to store the unsorted list, to avoid that the returned sorted list was used for the N-1 other repeated tests of each sample test i.

3.2 Results and Discussion

The Tables 1, 2, and 3 show the respective results for each algorithm. The results show the average runtime and the standard deviation both represented in seconds). As illustrated in 1 we find that with our implementations of the two algorithms in Python, we do not see any significant change in performance with the Dual Pivot implementation. On the contrary, we find that the classic quicksort still outperforms Yaroslavskiy's dual pivot version despite the results being very close. The dual pivot quicksort has about double as many swaps, but fewer comparisons (Kushagra et al. 2014). Thus, One explanation could be that the swap is more expensive in Python than in Java. Kushagra et al. (2014) concludes that fewer cache-misses is the determining factor for the dual pivot superior performance. So another likely explanation could be that the implementation in Python does not have the same decrease in cache misses as when being run in Java JVM. When we then add python's standard sorting algorithm *.sort*(), we see that it outperforms both of the quicksort implementations. In the Python documentation, the *.sort*() is written to be both stable and in-place, but the specifics on the implementation is not mentioned. According to Heumann (2018) the Python standard *.sort*() algorithm is based on a combination of Merge -and Insertion sort called Timsort. One must assume that the standard sorting algorithms for Python have been implemented with great care and thought. Thus this is not a surprising result. The result shows that the Python standard library *.sort*() growth in time fluctuates with the quicksort implementations (i.e., has the same performance $O(nlogn)$, but is faster by a constant factor.

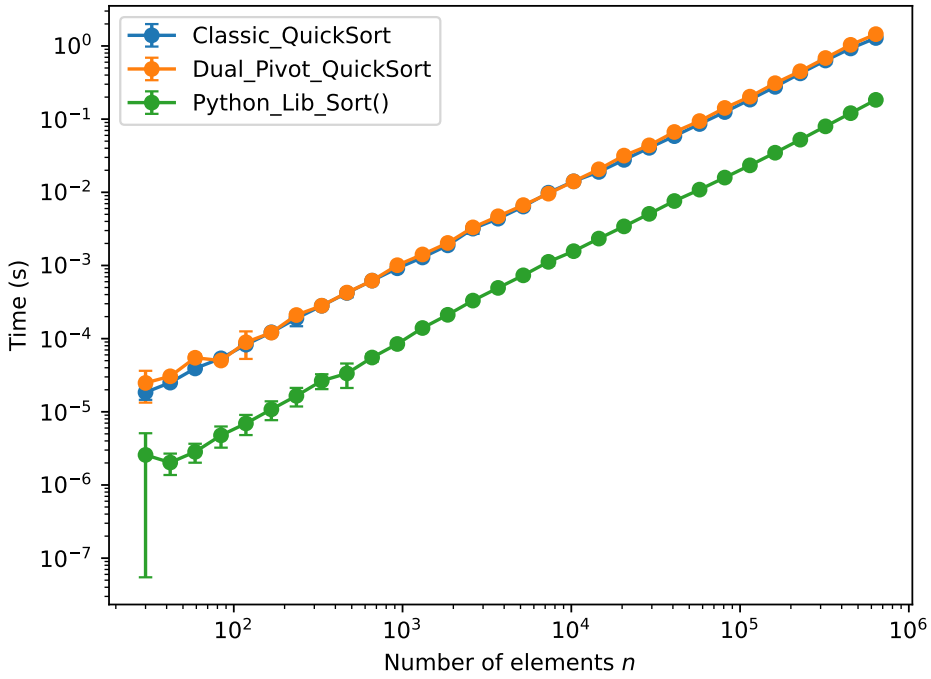

Figure 1: Comparison of classic quicksort, dual pivot, and the python library *.sort*() presented with log(time(s)) on the y-axis and the log(input size n) on the x-axis.

4 References

Heumann, A. (2018), "The magic behind the sort algorithm in Python", URL: https://medium.com/ub-women-data-scholars/the-magic-behind-the-sort-algorithm-in-python-1cb9515294b5

Wild, S., Nebel, M. E. (2012) *Average Case Analysis of Java 7's Dual Pivot Quicksort*, Fachbereich Informatik, Technische Universität Kaiserslautern

Kushagra, S., Lopez-Ortiz, A., Munro, J. I., Qiao, Aurick., (2014), *"Multi-Pivot Quicksort: Theory and Experiments"*, Society for Industrial and Applied Mathematics.

Table 1: The test results from the classic quicksort is presented below. n represents the data-set size, *Average* runtime and *Standard deviation* are represented in seconds.

n	Average	Standard deviation
30	0.000018	0.000004
42	0.000025	0.000001
59	0.000039	0.000002
84	0.000054	0.000001
118	0.000083	0.000002
167	0.000122	0.000003
235	0.000189	0.000042
332	0.000281	0.000025
468	0.000421	0.000054
660	0.000620	0.000034
931	0.000916	0.000052
1313	0.001287	0.000054
1852	0.001884	0.000047
2611	0.003187	0.000483
3682	0.004362	0.000302
5192	0.006385	0.000502
7321	0.009876	0.000455
10323	0.014130	0.000403
14556	0.018986	0.000836
20525	0.027760	0.000394
28940	0.040538	0.001498
40805	0.058533	0.001476
57536	0.085785	0.002091
81126	0.124957	0.002230
114387	0.184744	0.004036
161286	0.276189	0.010879
227414	0.422141	0.016481
320654	0.632830	0.064102
452122	0.924855	0.022969
637493	1.284919	0.004274

Table 2: The test results from the dual pivot quicksort is presented below. n represents the data-set size, *Average* runtime and *Standard deviation* are represented in seconds.

n	Average	Standard deviation
30	0.000025	0.000011
42	0.000031	0.000001
59	0.000055	0.000006
84	0.000050	0.000001
118	0.000089	0.000036
167	0.000121	0.000002
235	0.000210	0.000002
332	0.000282	0.000019
468	0.000426	0.000005
660	0.000620	0.000004
931	0.001008	0.000012
1313	0.001414	0.000019
1852	0.002032	0.000021
2611	0.003315	0.000141
3682	0.004701	0.000075
5192	0.006685	0.000109
7321	0.009581	0.000081
10323	0.014102	0.000127
14556	0.020623	0.000209
20525	0.031736	0.000642
28940	0.043825	0.000322
40805	0.066743	0.000228
57536	0.094101	0.000535
81126	0.142197	0.004308
114387	0.202742	0.001283
161286	0.309880	0.002634
227414	0.451188	0.003002
320654	0.684237	0.046517
452122	1.035635	0.074609
637493	1.450283	0.049553

Table 3: The test results from the Python library *.sort*() method is presented below. n represents the data-set size, *Average* runtime and *Standard deviation* are represented in seconds.

n	Average	Standard deviation
30	0.000003	0.000003
42	0.000002	0.000001
59	0.000003	0.000001
84	0.000005	0.000002
118	0.000007	0.000002
167	0.000011	0.000003
235	0.000017	0.000005
332	0.000027	0.000006
468	0.000033	0.000012
660	0.000055	0.000006
931	0.000085	0.000008
1313	0.000140	0.000008
1852	0.000212	0.000009
2611	0.000331	0.000019
3682	0.000494	0.000021
5192	0.000733	0.000032
7321	0.001121	0.000123
10323	0.001568	0.000041
14556	0.002331	0.000041
20525	0.003421	0.000021
28940	0.005075	0.000103
40805	0.007602	0.000284
57536	0.010873	0.000150
81126	0.015891	0.000136
114387	0.023352	0.000170
161286	0.034737	0.000632
227414	0.052397	0.000732
320654	0.079677	0.001041
452122	0.120636	0.001004
637493	0.183365	0.001720