Multi-Pivot Quicksort: an Experiment with Single, Dual, Triple, Quad, and Penta-Pivot Quicksort Algorithms in Python

M A Budiman*, E M Zamzami, and D Rachmawati
Departemen Ilmu Komputer, Fakultas Ilmu Komputer dan Teknologi Informasi,
Universitas Sumatera Utara, Jl. Universitas No. 9-A, Kampus USU, Medan 20155,
Indonesia

*mandrib@usu.ac.id

Abstract. Dual-pivot quicksort, which was proposed by Yaroslavsky, has been experimentally proven to be more efficient than the classical single-pivot quicksort under the Java Virtual Machine [6]. Moreover, Kushagara, López-Ortiz, and Munro [4] has shown that triple-pivot quicksort runs 7-8% faster than dual-pivot quicksort in C, mutatis mutandis. In this research, we implement and experiment with single, dual, triple, quad, and penta-pivot quicksort algorithms in Python. Our experimental results are as follows. Firstly, the quicksort with single pivot is the slowest among the five variants. Secondly, at least until five (penta) pivots are being used, it is proven that the more pivots are used in a quicksort algorithm, the faster its performance becomes. Thirdly, the increase of speed resulted by adding more pivots tends to decrease gradually.

1. Introduction

Sorting is a major algorithm in computer science. Many other algorithms depends on sorting operations; for example, if one would like to implement a searching algorithm whose purpose is to return the index number of a value in a list, then he or she must ensure that all the values in the list are sorted beforehand. Kruskal’s algorithm, a noteworthy algorithm to solve minimum spanning tree problem, also employs sorting algorithm in order to ascertain that all the edges of a graph are processed from lowest to highest weight.

Quicksort is a fast sorting algorithm whose procedure was published by Hoare [2] in 1961. Using the National-Elliott 405 computer on some random data (N = 500 to 2,000), quicksort was proven faster than the other state-of-the-art sorting algorithm, mergesort [3]. The average total running time of quicksort is O(N log N) [5]. This performance is significantly better than the theoretical performance of bubble sort, another popular sorting algorithm which has the average running time of O(N^2) [1]. Moreover, it has been experimentally shown that when the input size increases, the quicksort performs better than four popular sorting algorithms (bubble, selection, insertion, and merge) [7].

A new quicksort variant using dual-pivot partitioning was implemented in Java by Vladimir Yaroslavsky in 2009. Its performance was faster than the Hoare’s single-pivot quicksort algorithm. In 2013, Kushagara, López-Ortiz, and Munro rewrote the single and dual-pivot variants in C, and compared them with a new variant: the triple-pivot quicksort [4]. They showed that the triple-pivot
quicksort performed even faster than the dual-pivot quicksort. These results lead to a hypothesis: the more pivots are used in quicksort partitioning, the faster its performance becomes.

In order to confirm this hypothesis, we experiment with the single, dual, and triple-pivot quicksort algorithms on some data with various input size. Two more variants of quicksort, quad-pivot and penta-pivot, are also added to the experiments. All the algorithms are implemented in Python.

2. Method
The experiments are conducted on the iPad Air 2 which has A8X second generation processor with 64-bit architecture and M8 coprocessor. The development environment being used for coding the Python scripts is Pythonista v. 2.1.

In each step, the single-pivot quicksort algorithm employs a pivot, i.e. a single value taken from a list that will partition the list into two lists called ‘left’ and ‘right’. In this research, the pivot is chosen at the index 0 of the list.

At first, the ‘left’ and ‘right’ lists are set as empty. Then, the values of the list are scanned from the lowest index to the highest index. During the scan, the values which are smaller or equal than the pivot are put inside the ‘left’ list and the values which are bigger than the pivot are put inside the ‘right’ list. The ‘left’ and the ‘right’ lists are then ‘quicksorted’ recursively using the same mechanism until it is not possible to further partition the list anymore (in other words, until size the of the corresponding list has become one or zero). As intuitively expected, the original list is sorted.

The dual-pivot quicksort algorithm also employs the same recursive mechanism. The difference is that in each step, two pivots are chosen from index 0 and 1. To ensure that the first pivot are smaller or equal than the second pivot, a built-in Python sorting function (Timsort) is used. The two pivots then partitions the list into three lists called ‘a’, ‘b’, and ‘c’ where all values of list ‘a’ are smaller or equal than all values of list ‘b’ and all values of list ‘b’ are smaller or equal than all values of list ‘c’. (For the triple, quad, and penta-pivot versions, we use the letters from ‘a’ to ‘f’ to name the partitioned lists.)

In order to simplify the explanation, the Python source codes for the dual, triple, quad, and penta-pivot quicksort algorithms are shown as follows. (In Python, a colon followed by an indentation mark where a block or a function begins and ends)

```python
def QuickSort2Pivot(list):
    n = len(list)
    if n <= 1:
        return list
    elif n == 2:
        return sorted(list)
    pivot1, pivot2 = sorted([list.pop(0), list.pop(0)])
    a = []
    b = []
    c = []
    for element in list:
        if element < pivot1:
            a.append(element)
        elif pivot1 <= element < pivot2:
            b.append(element)
        else:
            c.append(element)
    return QuickSort2Pivot(a) + [pivot1] + QuickSort2Pivot(b) + [pivot2] + QuickSort2Pivot(c)

def QuickSort3Pivot(list):
    n = len(list)
    if n <= 1:
        return list
    elif n == 2:
        return sorted(list)
    pivot1, pivot2, pivot3 = sorted([list.pop(0), list.pop(0), list.pop(0)])
    a = []
    b = []
    c = []
    ```
def QuickSort3Pivot(list):
i = len(list)
if i <= 1:
    return list
elif i == 2 or i == 3:
    return sorted(list)
pivot1, pivot2, pivot3 = sorted([list.pop(0), list.pop(0), list.pop(0)])
a = []
b = []
c = []
d = []
e = []
for element in list:
    if element < pivot1:
        a.append(element)
    elif pivot1 <= element < pivot2:
        b.append(element)
    elif pivot2 <= element < pivot3:
        c.append(element)
    else:
        d.append(element)
e = d
return QuickSort3Pivot(a) + [pivot1] + QuickSort3Pivot(b) + [pivot2] + QuickSort3Pivot(c) + [pivot3] + QuickSort3Pivot(d)

def QuickSort4Pivot(list):
i = len(list)
if i <= 1:
    return list
elif i == 2 or i == 3:
    return sorted(list)
pivot1, pivot2, pivot3, pivot4 = sorted([list.pop(0), list.pop(0), list.pop(0), list.pop(0)])
a = []
b = []
c = []
d = []
e = []
f = []
for element in list:
    if element < pivot1:
        a.append(element)
    elif pivot1 <= element < pivot2:
        b.append(element)
    elif pivot2 <= element < pivot3:
        c.append(element)
    elif pivot3 <= element < pivot4:
        d.append(element)
    else:
        e.append(element)
f = e
return QuickSort4Pivot(a) + [pivot1] + QuickSort4Pivot(b) + [pivot2] + QuickSort4Pivot(c) + [pivot3] + QuickSort4Pivot(d) + [pivot4] + QuickSort4Pivot(e)

def QuickSort5Pivot(list):
i = len(list)
if i <= 1:
    return list
elif i == 2 or i == 3:
    return sorted(list)
pivot1, pivot2, pivot3, pivot4, pivot5 = sorted([list.pop(0), list.pop(0), list.pop(0), list.pop(0), list.pop(0)])
a = []
b = []
c = []
d = []
e = []
f = []
g = []
for element in list:
    if element < pivot1:
        a.append(element)
    elif pivot1 <= element < pivot2:
        b.append(element)
    elif pivot2 <= element < pivot3:
        c.append(element)
    elif pivot3 <= element < pivot4:
        d.append(element)
    elif pivot4 <= element < pivot5:
        e.append(element)
    else:
        f.append(element)
g = f
return QuickSort5Pivot(a) + [pivot1] + QuickSort5Pivot(b) + [pivot2] + QuickSort5Pivot(c) + [pivot3] + QuickSort5Pivot(d) + [pivot4] + QuickSort5Pivot(e) + [pivot5] + QuickSort5Pivot(f)
There are three sets of lists being experimented: ‘small’, ‘medium’, and ‘big’. The ‘small’ set consists of lists with \( N = 10,000 \) to \( N = 100,000 \) (with an increment of 10,000), while \( N \) denotes the size of each list. The ‘medium’ consists of lists with \( N = 100,000 \) to \( N = 1,000,000 \) with an increment of 100,000. The ‘big’ set consists of lists with \( N = 1,000,000 \) to \( N = 6,000,000 \) with an increment of 1,000,000. The data within each list are integers which are generated pseudorandomly using the built-in random.randrange(start, stop) function, with start = 1 and stop = 10,000. Each list is then processed in the five variants of quicksort algorithms and the real running time is recorded.

3. Results and Discussion
The results of the experiments of each set are presented in Table 1, Table 2, and Table 3 as follows:

| Table 1. The running time for the ‘small’ set. |
| N | QuickSort | QuickSort2Pivot | QuickSort3Pivot | QuickSort4Pivot | QuickSort5Pivot |
|---|---|---|---|---|---|
| 10,000 | 0.087 | 0.059 | 0.057 | 0.053 | 0.050 |
| 20,000 | 0.179 | 0.129 | 0.128 | 0.122 | 0.112 |
| 30,000 | 0.266 | 0.190 | 0.185 | 0.175 | 0.166 |
| 40,000 | 0.386 | 0.272 | 0.253 | 0.236 | 0.234 |
| 50,000 | 0.474 | 0.344 | 0.321 | 0.297 | 0.278 |
| 60,000 | 0.617 | 0.444 | 0.404 | 0.369 | 0.356 |
| 70,000 | 0.733 | 0.520 | 0.479 | 0.444 | 0.422 |
| 80,000 | 0.865 | 0.615 | 0.562 | 0.522 | 0.501 |
| 90,000 | 1.002 | 0.710 | 0.642 | 0.597 | 0.558 |
| 100,000 | 1.107 | 0.793 | 0.721 | 0.677 | 0.641 |

In Table 1, it can be seen that the penta-pivot quicksort are the fastest to sort the lists of 10,000 to 100,000 integers, followed consecutively by the quad, the triple, the dual, and the single-pivot quicksorts. When penta-pivot is used instead of single-pivot to sort the list of 100,000 integers, the performance is increased by approximately 42%.

| Table 2. The running time for the ‘medium’ set. |
| N | QuickSort | QuickSort2Pivot | QuickSort3Pivot | QuickSort4Pivot | QuickSort5Pivot |
|---|---|---|---|---|---|
| 100,000 | 1.107 | 0.793 | 0.721 | 0.677 | 0.641 |
| 200,000 | 2.687 | 1.923 | 1.694 | 1.611 | 1.541 |
| 300,000 | 4.533 | 3.137 | 2.770 | 2.659 | 2.513 |
| 400,000 | 6.936 | 4.801 | 4.165 | 3.949 | 3.807 |
| 500,000 | 10.034 | 6.576 | 5.769 | 5.442 | 5.246 |
| 600,000 | 12.824 | 8.290 | 7.351 | 6.882 | 6.651 |
| 700,000 | 16.211 | 10.545 | 9.039 | 8.521 | 8.158 |
| 800,000 | 19.514 | 12.609 | 11.177 | 10.411 | 10.032 |
| 900,000 | 23.244 | 14.969 | 13.148 | 12.555 | 11.965 |
| 1,000,000 | 27.802 | 17.708 | 15.584 | 14.815 | 14.180 |

Table 2 also shows that the penta-pivot quicksort are the fastest to sort the lists of 100,000 to 1,000,000 integers, followed consecutively by the quad, the triple, the dual, and the single-pivot quicksorts. When penta-pivot is used instead of single-pivot to sort a list of 1,000,000 integers, the performance is increased by approximately 48%.
Table 3. The running time for the ‘big’ set

| N          | QuickSort | QuickSort2Pivot | QuickSort3Pivot | QuickSort4Pivot | QuickSort5Pivot |
|------------|-----------|-----------------|-----------------|-----------------|-----------------|
| 1,000,000  | 27.802    | 17.708          | 15.584          | 14.815          | 14.180          |
| 2,000,000  | 88.206    | 55.339          | 46.818          | 44.493          | 43.137          |
| 3,000,000  | 205.545   | 106.584         | 93.031          | 91.273          | 85.176          |
| 4,000,000  | 582.986   | 353.163         | 158.157         | 146.858         | 138.652         |
| 5,000,000  | 839.947   | 590.783         | 507.473         | 288.593         | 220.870         |
| 6,000,000  | 1494.739  | 765.883         | 686.269         | 642.772         | 580.823         |

In Table 3, it can be shown that the penta-pivot quicksort are also the quickest to sort the lists of 1,000,000 to 6,000,000 integers, followed by the quad, the triple, the dual, and the single-pivot quicksorts. When penta-pivot is used instead of single-pivot to sort a list of size 6,000,000, the performance is increased by around 61%. Therefore, by inspecting Table 1, Table 2, and Table 3, we may conclude that the more pivots are used, the faster the performance becomes.

In addition, Figure 1 is constructed to summarize what happens if the results in Table 1, Table 2, and Table 3 are plotted in a lin-log scale. A lin-log scale is a graph with linear scale on its vertical axis and logarithmic scale on its horizontal axis.

![Figure 1. Real running time for all the three sets combined](image)

Figure 1. Real running time for all the three sets combined

From Figure 1, it can be noted that while it is true that more pivots can mean more speed, the increase of speed caused by adding more pivots may slowly diminish. In other words, it is expected that there will be a maximum point that if one is to add one more pivot, there will be no increase of speed at all.

4. Conclusion

The conclusion of this research are as follows. First, the quicksort with single pivot is the slowest among the other quicksort algorithms with many pivots. Second, it is confirmed (in the case of single, dual, triple, quad, and penta pivots) that the more pivots are used in a quicksort algorithm, the faster its performance becomes. Third, the increase of speed by implementing more pivots in a quicksort algorithm tends to diminish slowly.
5. Acknowledgments
We gratefully acknowledge that this research is funded by the Ministry of Research and Technology and Higher Education Republic of Indonesia. The support is under the research grant BP-PTN USU of Year 2016 Contract Number 192/UN5.2.3.1/PPM/SP/2016.

References
[1] Astrachan O 2003 Bubble sort: an archaeological algorithmic analysis ACM SIGCSE Bulletin 35 1.
[2] Hoare C A R 1961 Algorithm 64: Quicksort Communications of the ACM 4 321.
[3] Hoare C A R 1962 Quicksort The Computer Journal 5 10–6.
[4] Kushagra S, López-Ortiz A, Qiao A and Munro J I 2013 Multi-Pivot Quicksort: Theory and Experiments 2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX) 47–60.
[5] Sedgewick R 1977 The analysis of Quicksort programs Acta Informatica 7 327–55.
[6] Wild S and Nebel M E 2012 Average Case Analysis of Java 7’s Dual Pivot Quicksort Algorithms – ESA 2012 Lecture Notes in Computer Science 825–36.
[7] Yang Y, Yu P and Gan Y 2011 Experimental study on the five sort algorithms 2011 Second International Conference on Mechanic Automation and Control Engineering.