An analysis of the worst-case performance of Quicksort

Student Name

BU MET College

Department Computer Science

Fall, 2017

Abstract

C.A.R. Hoare’s quicksort algorithm has become a very popular sorting algorithm due to the average performance of Θ(n log n), limited use of extra storage (typically Θ(log2 n) recursive calls) and better performance on average compared to heapsort (another Θ(n log n) sorting algorithm). It may be found in several standard libraries supporting C, C++, and

2

Java. The major drawback in the quicksort algorithm is the Θ(n ) worse case performance. Unfortunately, this performance is exhibited for some rather common initial permutations.

I intend to look into this performance of the quicksort algorithm, and in particular potential modifications to minimize the probability that the worst-case performance will be exhibited.

Contents

1. [Historical 3](#_TOC_250014)
   1. [Algorithm 63 - partition 3](#_TOC_250013)
   2. [Algorithm 64 - quicksort 3](#_TOC_250012)
2. [Java implementation 3](#_TOC_250011)
3. [Average case performance 3](#_TOC_250010)
4. [Worse case performance 3](#_TOC_250009)
   1. [Realized when array is already in ascending sequence 3](#_TOC_250008)
   2. [Realized when array is in descending sequence 4](#_TOC_250007)
5. [Near worse case performance 4](#_TOC_250006)
6. [Avoidance of worse case and near worse case performance 4](#_TOC_250005)
   1. [Random selection 4](#_TOC_250004)
   2. [Median 4](#_TOC_250003)
7. [Alternative method for small subarrays 5](#_TOC_250002)
8. [Implementations with various improvements 5](#_TOC_250001)
9. [Results 5](#_TOC_250000)
10. Original version using first element as pivot 10
11. Decision tree for less than three elements 10
12. Decision tree for less than four elements 10
13. Less than 4 by decision tree, median pivot 10
14. Less than 4 by decision tree, random pivot 10

# Historical

The original quicksort was described by C.A.R. Hoare in Algorithms 63 and 64 of the Collected Algorithms from the Association for Computing Machinery. (Presented in the original Algol).[ACM98]

## Algorithm 63 - partition

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Code here . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

## Algorithm 64 - quicksort

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . Code here . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

# Java implementation

A Java implementation of Quicksort was created and instrumented to count the number of key comparisons and the number of key exchanges. The source code is included in the appendixes. The original Java implementation uses the first element of the subarray as the pivot value, as described in [BG99] pages 159-171.

# Average case performance

In the average case, quicksort has a recurrence relation of *T* (*n*) = 2*T* ( *N* ). That

2 *n*

is, on average, the pivot procedure produces two subarrays of approximately 2

elements. The depth of the recursion tree is log2 *n*. Summing over all the levels, we have Θ(*n* log *n*). [BG99] pages 165-168 [CLR90] pages 159-160 [Sta94] pages

544-546 [Wei96] pages 244-245

# Worse case performance

Quicksort has a Θ(*n*2) worse case performance. [BG99] pages 162-165 [CLR90] page 156 [Sta94] page 547 [Wei96] page 243

## Realized when array is already in ascending sequence

In the case where the array is in ascending sequence, the partition procedure will partition the array such that the left subarray has only one element. Here the recursion relationship degrades to *T* (*n*) = *T* (*n* − 1) + Θ(1). In this case, *T* (*n*) ∈ Θ(*n*2).

## Realized when array is in descending sequence

In the case where the array is in descending sequence, the partition procedure will partition the array such that the right subarray has only one element. Performance is also Θ(*n*2), as in the previous case.

# Near worse case performance

Near worse case performance is realized when array is already in nearly ascend- ing or descending sequence. A typical example would be a small set of elements (all with keys greater than the existing array) appended to the previously sorted array. The various version of quicksort were run with an array that had the first 90% of the elements in either ascending or descending sequence, followed by a set of either ordered or unordered elements with larger keys.

An occurrence of this is not uncommon, when an implementation na¨ıvely appends the new elements with assigned identification numbers to an already existing array. A much better approach would be to sort the new elements separately and then merge the results with the existing array. 1

Another situation that may occur is when the array is already sorted by one key and then sorted by another key that is not independent. Consider the case where the array is initially sorted by zip code and then quicksort is used to sort it by state. Since zip codes are grouped by state, the array will contain several long runs of keys. There is a similar dependence between social security numbers and state of residence when the number is assigned.

# Avoidance of worse case and near worse case performance

## Random selection

A randomly selected element in the subarray is exchanged with the first element and becomes the pivot element. This method was used by C.A.R. Hoare in his original implementation. [ACM98] Algorithm 63

## Median

The median of a small number of elements chosen from the subarray is exchanged with the first element and becomes the pivot element. [Knu73] page 123

1Since the new elements would have assigned identification numbers, these may be in a near ascending sequence, so the use of the first element as a pivot for a quicksort of the new elements would exhibit Θ(*n*2) behavior which should be avoided.

# Alternative method for small subarrays

Another quicksort optimization involves the use of an alternative sorting algo- rithm when the subarray size is below a certain limit. Typically, this limit is chosen as 2 or 3, in which case the elements may be ordered using a decision tree. Although this will not effect the asymptotic behavior, it will eliminate a few levels from the recursion tree and reduce stack usage. The reduction will

be Θ(*n*), reducing the number of recursive calls by *n*, 3*n* , 7*n* in the cases where

2 4

one, two, or three levels are eliminated.

# Implementations with various improvements

The following implementations of the quicksort algorithm where written in Java (included in the appendixes) and run to gather data.

q0 Original version using the first element as the pivot q1 Decision tree for *n <* 3

q2 Decision tree for *n <* 4

q3 Decision tree for *n <* 4, median of left, middle and right as pivot q4 Decision tree for *n <* 4, random pivot selection

# Results

Each implementation was executed 100 times for permutations of size 10 to 100 in steps of 10. Nine different types of permutations were used (all values were unique):

aa A strictly ascending permutation

ad The first 90% were ascending, 10% descending

ar The first 90% were ascending, 10% random

da The first 90% were descending, 10% ascending

dd A strictly descending permutation

dr The first 90% were descending, 10% random ra The first 90% were random, 10% ascending rd The first 90% were random, 10% descending rr A random permutation

Number of comparisons for q0

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Permutation | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| aa | 63 | 228 | 493 | 858 | 1323 | 1888 | 2553 | 3318 | 4183 | 5148 |
| ad | 63 | 229 | 494 | 860 | 1325 | 1891 | 2556 | 3322 | 4187 | 5153 |
| ar | 63 | 228 | 494 | 859 | 1324 | 1889 | 2553 | 3316 | 4178 | 5140 |
| da | 60 | 205 | 431 | 740 | 1130 | 1603 | 2157 | 2794 | 3512 | 4313 |
| dd | 68 | 238 | 508 | 878 | 1348 | 1918 | 2588 | 3358 | 4228 | 5198 |
| dr | 60 | 205 | 432 | 741 | 1131 | 1604 | 2157 | 2792 | 3507 | 4305 |
| ra | 53 | 137 | 233 | 330 | 436 | 551 | 668 | 796 | 918 | 1042 |
| rd | 53 | 138 | 232 | 338 | 437 | 560 | 679 | 799 | 921 | 1053 |
| rr | 54 | 138 | 235 | 340 | 448 | 570 | 680 | 794 | 919 | 1052 |

Number of comparisons for q1

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Permutation | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| aa | 61 | 226 | 491 | 856 | 1321 | 1886 | 2551 | 3316 | 4181 | 5146 |
| ad | 61 | 226 | 492 | 857 | 1323 | 1888 | 2554 | 3319 | 4185 | 5150 |
| ar | 61 | 226 | 491 | 856 | 1320 | 1884 | 2547 | 3310 | 4171 | 5132 |
| da | 56 | 200 | 427 | 735 | 1126 | 1598 | 2153 | 2789 | 3508 | 4308 |
| dd | 65 | 235 | 505 | 875 | 1345 | 1915 | 2585 | 3355 | 4225 | 5195 |
| dr | 56 | 200 | 427 | 735 | 1125 | 1596 | 2150 | 2782 | 3498 | 4294 |
| ra | 45 | 121 | 213 | 301 | 401 | 508 | 618 | 741 | 856 | 972 |
| rd | 45 | 122 | 211 | 309 | 402 | 517 | 630 | 742 | 858 | 983 |
| rr | 46 | 122 | 212 | 309 | 410 | 523 | 626 | 732 | 849 | 976 |

Number of comparisons for q2

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Permutation | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| aa | 58 | 223 | 488 | 853 | 1318 | 1883 | 2548 | 3313 | 4178 | 5143 |
| ad | 58 | 224 | 489 | 855 | 1320 | 1886 | 2551 | 3317 | 4182 | 5148 |
| ar | 58 | 223 | 488 | 854 | 1318 | 1881 | 2544 | 3307 | 4166 | 5127 |
| da | 53 | 195 | 421 | 730 | 1120 | 1593 | 2147 | 2784 | 3502 | 4303 |
| dd | 63 | 233 | 503 | 873 | 1343 | 1913 | 2583 | 3353 | 4223 | 5193 |
| dr | 53 | 195 | 421 | 731 | 1120 | 1591 | 2143 | 2776 | 3490 | 4287 |
| ra | 40 | 111 | 197 | 282 | 377 | 479 | 586 | 703 | 815 | 927 |
| rd | 40 | 112 | 196 | 290 | 377 | 490 | 597 | 707 | 816 | 938 |
| rr | 41 | 112 | 197 | 289 | 385 | 494 | 591 | 693 | 807 | 927 |

Number of comparisons for q3

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Permutation | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| aa | 39 | 101 | 168 | 247 | 322 | 396 | 484 | 577 | 662 | 744 |
| ad | 39 | 102 | 170 | 250 | 325 | 396 | 484 | 578 | 667 | 749 |
| ar | 39 | 101 | 169 | 247 | 325 | 399 | 487 | 581 | 669 | 752 |
| da | 50 | 124 | 208 | 297 | 395 | 484 | 578 | 682 | 792 | 895 |
| dd | 41 | 106 | 169 | 251 | 323 | 406 | 490 | 584 | 664 | 755 |
| dr | 50 | 124 | 208 | 300 | 397 | 491 | 585 | 690 | 800 | 906 |
| ra | 44 | 115 | 201 | 290 | 381 | 480 | 581 | 690 | 789 | 893 |
| rd | 42 | 117 | 203 | 294 | 387 | 481 | 582 | 684 | 802 | 899 |
| rr | 46 | 119 | 205 | 299 | 393 | 495 | 602 | 700 | 811 | 933 |

Number of comparisons for q4

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Permutation | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| aa | 40 | 112 | 195 | 277 | 369 | 471 | 569 | 673 | 782 | 883 |
| ad | 40 | 113 | 192 | 276 | 375 | 476 | 567 | 680 | 778 | 892 |
| ar | 41 | 112 | 189 | 279 | 370 | 470 | 568 | 675 | 775 | 895 |
| da | 41 | 113 | 192 | 283 | 385 | 471 | 571 | 684 | 786 | 887 |
| dd | 42 | 111 | 195 | 280 | 368 | 476 | 588 | 676 | 781 | 888 |
| dr | 40 | 111 | 197 | 282 | 375 | 473 | 571 | 678 | 789 | 886 |
| ra | 42 | 115 | 192 | 289 | 378 | 476 | 578 | 672 | 778 | 888 |
| rd | 41 | 113 | 194 | 288 | 383 | 479 | 578 | 685 | 791 | 890 |
| rr | 43 | 113 | 195 | 286 | 378 | 470 | 579 | 681 | 801 | 902 |

References

[ACM98] ACM. Conference Proceedings of the Twelfth Annual ACM Symposium

on Theory of Computing: Papers Presented at the Symposium,

Los Angeles, California, April 28-30, 1980. Assn for Computing Machinery,

1998.

[BG99] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design and Analysis (3rd Edition)*. Addison Wesley, 1999.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *In- troduction to Algorithms (MIT Electrical Engineering and Computer Science)*. The MIT Press, 1990.

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching. Volume 3 (Addison Wesley Series in Computer Science and Information Processing)*. Addison-Wesley, 1973.

[Sta94] Thomas A. Standish. *Data Structures, Algorithms, and Software Principles in C*. Addison Wesley, 1994.

[Wei96] Mark A. Weiss. *Data Structures and Algorithm Analysis in C (2nd Edition)*. Addison Wesley, 1996.