Why is quicksort better than mergesort?

Asked 16 years, 3 months ago Modified 2 years, 4 months ago

Viewed 242k times

416

I was asked this question during an interview. They're both O(nlogn) and yet most people use Quicksort instead of Mergesort. Why is that?

algorithm sorting language-agnostic quicksort mergesort

Share

Improve this question

Follow

edited Jul 9, 2013 at 20:14

templatetypedef

372k ● 111 ● 941 ● 1.1k

asked Sep 16, 2008 at 8:37

Malik Daud Ahmad Khokhar

13.7k ● 24 ● 80 ● 81

111 This is not a very good interview question. Real-world data isn't shuffled: it often contains a lot of order which a smart sort can make use of, and while neither algorithm does this automatically, it's easier to hack a merge sort to do it than a quicksort. GNU libc's `qsort`, Python's `list.sort`, and the `Array.prototype.sort` in Firefox's JavaScript are all souped-up merge sorts. (GNU STL `sort` uses Introsort

instead, but that might be because in C++, swapping potentially wins big over copying.) – Jason Orendorff Dec 10, 2009 at 1:01

5 @Jason Orendorff: Why is it `"easier to hack a mergesort to do it than a quicksort"`? Any specific example that you can quote? – Lazer Apr 3, 2010 at 10:55

17 @eSKay A merge sort starts by grouping the initial data into sorted subarrays. If the array initially contains some already-sorted regions, you can save a lot of time just by detecting that they're there before you begin. And you can do that in O(n) time. For specific examples, see the source code of the three projects I mentioned! The best example might be Python's Timsort, described in detail here: svn.python.org/view/python/trunk/Objects/… and implemented in svn.python.org/view/python/trunk/Objects/… . – Jason Orendorff Apr 5, 2010 at 19:28

5 @JasonOrendorff: Not sure I buy your argument that mergesort can be more easily modified to take advantage of already-sorted sections. The partitioning step of quicksort can be trivially modified to afterwards check whether both resulting partitions are sorted, and halt recursion if they are. This potentially doubles the number of comparisons, but doesn't alter the O(n) time complexity of that step. – j_random_hacker Jul 15, 2012 at 4:06

4 @j_random_hacker: right, that's what I was implying. But consider: {10, 2, 3, 4, 5, 6, 7, 8, 1, 9} Despite being almost completely sorted already, checking before the partition won't find it, nor after. And the partition will screw it up before subsequent calls would check for it. Meanwhile, merge sorts check for sorted sequences in the division steps before any are moved, and smart ones will look for runs like this specifically during the division step (see: Tim Sort) – Mooing Duck Oct 28, 2014 at 0:13

29 Answers

Sorted by: Highest score (default)

334

Quicksort has O(n^2) worst-case runtime and O($n$log$n$) average case runtime. However, it's superior to merge sort in many scenarios because many factors influence an algorithm's runtime, and, when taking them all together, quicksort wins out.

In particular, the often-quoted runtime of sorting algorithms refers to the number of comparisons or the number of swaps necessary to perform to sort the data. This is indeed a good measure of performance, especially since it's independent of the underlying hardware design. However, other things – such as locality of reference (i.e. do we read lots of elements which are probably in cache?) – also play an important role on current hardware. Quicksort in particular requires little additional space and exhibits good cache locality, and this makes it faster than merge sort in many cases.

In addition, it's very easy to avoid quicksort's worst-case run time of O(n^2) almost entirely by using an appropriate choice of the pivot – such as picking it at random (this is an excellent strategy).

In practice, many modern implementations of quicksort (in particular libstdc++'s `std::sort`) are actually introsort, whose theoretical worst-case is O($n$log$n$), same as merge sort. It achieves this by limiting the recursion depth, and switching to a different algorithm (heapsort) once it exceeds log$n$.

Share Improve this answer Follow

edited May 19, 2014 at 6:59

answered Sep 16, 2008 at 9:14

Konrad Rudolph
545k ● 139 ● 956 ● 1.2k

7 The Wikipedia article states it switches to heapsort, not mergesort...just FYI. – Sev Sep 9, 2010 at 6:46

3 @Sev: … as does the orignal paper. Thanks for pointing out the mistake. – Not that it really matters, since their asymptotic running time is the same. – Konrad Rudolph Sep 11, 2010 at 11:46

128 why is this selected as the correct answer ?. All it explains is how quick sorts problems be patched. It still doesnt tell why quick sort is used more than other ?. Is the answer "quick sort is used more than other because after one depth you can switch to heapsort"? .. why not use heapsort in the first place then ? .. just trying to understand ... – codeObserver Apr 4, 2011 at 7:13

21 @p1 Good question. The real answer is that on average, for average data, quicksort is faster than merge sort (and heap sort, for that matter), and even though the worst case of quicksort is slower than merge sort's, this worst case can be mitigated very easily (hence my answer). – Konrad Rudolph Apr 4, 2011 at 7:17

6 Quicksort is better in terms of memory as well. – Shashwat May 18, 2014 at 21:38

312

As many people have noted, the average case performance for quicksort is faster than mergesort. **But** this is only true if you are assuming constant time to access any piece of memory on demand.

In RAM this assumption is generally not too bad (it is not always true because of caches, but it is not too bad). However if your data structure is big enough to live on disk, then quicksort gets *killed* by the fact that your average disk does something like 200 random seeks per second. But that same disk has no trouble reading or writing megabytes per second of data sequentially. Which is exactly what mergesort does.

Therefore if data has to be sorted on disk, you really, really want to use some variation on mergesort. (Generally you quicksort sublists, then start merging them together above some size threshold.)

Furthermore if you have to do *anything* with datasets of that size, think hard about how to avoid seeks to disk. For instance this is why it is standard advice that you drop indexes before doing large data loads in databases, and then rebuild the index later. Maintaining the index during the load means constantly seeking to disk. By contrast if you drop the indexes, then the database can rebuild the index by first sorting the information to be dealt with (using a mergesort of course!) and then loading it into a BTREE datastructure for the index. (BTREEs are naturally kept in order, so you can load one from a sorted dataset with few seeks to disk.)

There have been a number of occasions where understanding how to avoid disk seeks has let me make data processing jobs take hours rather than days or weeks.

Share Improve this answer Follow

answered Sep 18, 2008 at 6:19

user11318
9,343 ● 2 ● 27 ● 25

---

1 Very nice, didn't think about the assumptions made for accessing the data structure. Good insight :) – chutsu Feb 20, 2014 at 12:12

2 Can you explain what you mean by "seek to disk" does it mean searching for some single value when the data is stored on disk? – James Wierzba Jun 19, 2015 at 17:11

10 @JamesWierzba I take it from context that he means "seeking to a location on disk". "Seeking" on a rotating disk device means, picking up the read head and moving it to a new absolute address, which is a notoriously slow operation. When you access data in the order it was stored, the disk hardware doesn't have to seek, it just plows along at high speed, reading items sequentially. – nclark May 27, 2016 at 16:06

1 Can some explain this a bit more? This is how I am seeing it: Quicksort: If we are going with random pivot, the call stack has fragments of the array partitioned in a random way. This requires random access. However, for each call in the stack, both left and right pointers move sequentially. I am assuming these would be kept in the cache. The swaps are operations again on information that's in cache (and eventually written to Disk). (continued in my next comment) – sam May 6, 2017 at 2:13

1 Just a contribution *avoiding the **costly** disk read / write overhead*: When sorting very large data that needs disk access, it's advantageous to switch the direction of sort for each pass. That is, at the very top level of the loop, once you go from `0` towards `n` and the next time you go from `n` towards `0`. This brings the advantage of retreating (sorting) the data blocks that are already available in the memory (cache) and attacking twice for only one disk access. I think most DBMS's use this optimization technique. – ssd Mar 2, 2018 at 15:20

100

Actually, QuickSort is O(n^2). Its *average case* running time is O(nlog(n)), but its *worst-case* is O(n^2), which occurs when you run it on a list that contains few unique items. Randomization takes O(n). Of course, this doesn't change its worst case, it just prevents a malicious user from making your sort take a long time.

QuickSort is more popular because it:

1. Is in-place (MergeSort requires extra memory linear to number of elements to be sorted).
2. Has a small hidden constant.

Share Improve this answer Follow

edited Oct 22, 2015 at 23:35
david_adler
10.9k ● 10 ● 67 ● 118

answered Sep 16, 2008 at 8:41
Dark Shikari
8,009 ● 4 ● 28 ● 38

6 Actually, there are implementation of QuickSort which are O(n*log(n)), not O(n^2) in the worst case. – jfs Sep 16, 2008 at 22:17

15 It also depends on the computer architecture. Quicksort benefits from the cache, while MergeSort doesn't. – Cristian Ciupitu Sep 28, 2008 at 1:53

5 @J.F. Sebastian: These are most probably introsort implementations, not quicksort (introsort starts as quicksort and switches to heapsort if it is about to stop being n*log(n)). – CesarB Oct 19, 2008 at 21:50

51 You can implement a mergesort in place. – Marcin Oct 20, 2008 at 9:25

8 Merge sort may be implemented in a way that only requires O(1) extra storage, but most of those implementations suffer greatly in terms of performance. – Clearer Dec 21, 2014 at 19:07

37

"and yet most people use Quicksort instead of Mergesort. Why is that?"

One psychological reason that has not been given is simply that Quicksort is more cleverly named. ie good marketing.

Yes, Quicksort with triple partioning is probably one of the best general purpose sort algorithms, but theres no getting over the fact that "Quick" sort sounds much more powerful than "Merge" sort.

Share Improve this answer answered Nov 13, 2009 at 4:53

Follow

Ash
62.1k ● 31 ● 155 ● 172

15 Does not answer question about which is better. The name of the algorithm is irrelevant in determining which is better. – Nick Gallimore Feb 24, 2019 at 19:12

1 NOEGFFUQS = no one ever got fired for using quicksort – Code Whisperer Sep 23, 2022 at 20:51

23

As others have noted, worst case of Quicksort is O(n^2), while mergesort and heapsort stay at O(nlogn). On the average case, however, all three are O(nlogn); so they're for the vast majority of cases comparable.

What makes Quicksort better on average is that the inner loop implies comparing several values with a single one, while on the other two both terms are different for each comparison. In other words, Quicksort does half as many reads as the other two algorithms. On modern CPUs performance is heavily dominated by access times, so in the end Quicksort ends up being a great first choice.

Share Improve this answer

Follow

answered Sep 17, 2008 at 2:09

Javier
62.4k ● 9 ● 81 ● 126

12

This is a common question asked in the interviews that despite of better worst case performance of merge sort, quicksort is considered better than merge sort, especially for a large input. There are certain reasons due to which quicksort is better:

1- Auxiliary Space: Quick sort is an in-place sorting algorithm. In-place sorting means no additional storage space is needed to perform sorting. Merge sort on the other hand requires a temporary array to merge the sorted arrays and hence it is not in-place.

2- Worst case: The worst case of quicksort `O(n^2)` can be avoided by using randomized quicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort.

3- Locality of reference: Quicksort in particular exhibits good cache locality and this makes it faster than merge sort in many cases like in virtual memory environment.

4- Tail recursion: QuickSort is tail recursive while Merge sort is not. A tail recursive function is a function where recursive call is the last thing executed by the function. The tail recursive functions are considered better than non tail recursive functions as tail-recursion can be optimized by compiler.

edited Mar 19, 2020 at 15:56

answered Mar 19, 2020 at 15:49

Himanshu Kansal
550 ● 7 ● 18

Love this answer - thanks! Nice and concise. If anyone has corrections or amendments to this, please comment. – rinogo Sep 19, 2022 at 21:36

(Might be worth mentioning in point 2 that the "median of three random numbers" approach seems to prevent O(n^2) time complexity in most situations) – rinogo Sep 19, 2022 at 21:37

Some versions of merge sort that use auxiliary space--which isn't a problem if one wants to leave the original list intact but also have a sorted list of pointers to elements, aren't recursive at all. Further, when sorting files, the amount of RAM required can be very small. – supercat Mar 15, 2023 at 23:07

9

I'd like to add that of the three algoritms mentioned so far (mergesort, quicksort and heap sort) only mergesort is stable. That is, the order does not change for those values which have the same key. In some cases this is desirable.

But, truth be told, in practical situations most people need only good average performance and quicksort is... quick =)

All sort algorithms have their ups and downs. See Wikipedia article for sorting algorithms for a good overview.

Share Improve this answer Follow

answered Sep 16, 2008 at 8:47
Antti Rasinen
10.2k ● 2 ● 24 ● 18

8

Mu! Quicksort is not better, it is well suited for a different kind of application, than mergesort.

> Mergesort is worth considering if speed is of the essence, bad worst-case performance cannot be tolerated, and extra space is available.[1]

You stated that they «They're both O(nlogn) […]». This is wrong. «Quicksort uses about n^2/2 comparisons in the worst case.»[1].

However the most important property according to my experience is the easy implementation of sequential access you can use while sorting when using programming languages with the imperative paradigm.

1 Sedgewick, Algorithms

Share Improve this answer Follow

answered Sep 16, 2008 at 9:13
Roman Glass
784 ● 1 ● 13 ● 20

Mergesort can be implemented in-place, such that it does not need extra space. For example with a double linked list: stackoverflow.com/questions/2938495/… – lanoxx May 8, 2013 at 7:51

8

I would like to add to the existing great answers some math about how QuickSort performs when diverging from best case and how likely that is, which I hope will help people understand a little better why the O(n^2) case is not of real concern in the more sophisticated implementations of QuickSort.

Outside of random access issues, there are two main factors that can impact the performance of QuickSort and they are both related to how the pivot compares to the data being sorted.

1) A small number of keys in the data. A dataset of all the same value will sort in n^2 time on a vanilla 2-partition QuickSort because all of the values except the pivot location are placed on one side each time. Modern implementations address this by methods such as using a 3-partition sort. These methods execute on a dataset of all the same value in O(n) time. So using such an implementation means that an input with a small number of keys actually improves performance time and is no longer a concern.

2) Extremely bad pivot selection can cause worst case performance. In an ideal case, the pivot will always be such that 50% the data is smaller and 50% the data is larger, so that the input will be broken in half during each iteration. This gives us n comparisons and swaps times log-2(n) recursions for O(n*logn) time.

How much does non-ideal pivot selection affect execution time?

Let's consider a case where the pivot is consistently chosen such that 75% of the data is on one side of the pivot. It's still O(n*logn) but now the base of the log has changed to 1/0.75 or 1.33. The relationship in performance when changing base is always a constant represented by log(2)/log(newBase). In this case, that constant is 2.4. So this quality of pivot choice takes 2.4 times longer than the ideal.

How fast does this get worse?

Not very fast until the pivot choice gets (consistently) very bad:

- 50% on one side: (ideal case)
- 75% on one side: 2.4 times as long
- 90% on one side: 6.6 times as long
- 95% on one side: 13.5 times as long
- 99% on one side: 69 times as long

As we approach 100% on one side the log portion of the execution approaches n and the whole execution asymptotically approaches O(n^2).

In a naive implementation of QuickSort, cases such as a sorted array (for 1st element pivot) or a reverse-sorted array (for last element pivot) will reliably produce a worst-case O(n^2) execution time. Additionally, implementations with a predictable pivot selection can be subjected to DoS attack by data that is designed to produce worst case execution. Modern implementations avoid this by a variety of methods, such as randomizing the data before sort, choosing the median of 3 randomly chosen indexes, etc. With this randomization in the mix, we have 2 cases:

- Small data set. Worst case is reasonably possible but O(n^2) is not catastrophic because n is small enough that n^2 is also small.
- Large data set. Worst case is possible in theory but not in practice.

How likely are we to see terrible performance?

The chances are *vanishingly small*. Let's consider a sort of 5,000 values:

Our hypothetical implementation will choose a pivot using a median of 3 randomly chosen indexes. We will consider pivots that are in the 25%-75% range to be "good" and pivots that are in the 0%-25% or 75%-100% range to be "bad". If you look at the probability distribution using the median of 3 random indexes, each recursion has an 11/16 chance of ending up with a good pivot. Let us make 2 conservative (and false) assumptions to simplify the math:

1. Good pivots are always exactly at a 25%/75% split and operate at 2.4*ideal case. We never get an ideal split or any split better than 25/75.

2. Bad pivots are always worst case and essentially contribute nothing to the solution.

Our QuickSort implementation will stop at n=10 and switch to an insertion sort, so we require 22 25%/75% pivot partitions to break the 5,000 value input down that far. (10*1.333333^22 > 5000) Or, we require 4990 worst case pivots. Keep in mind that if we accumulate 22 good pivots at *any point* then the sort will complete, so worst case or anything near it requires *extremely* bad luck. If it took us 88 recursions to actually achieve the 22 good pivots required to sort down to n=10, that would be 4*2.4*ideal case or about 10 times the execution time of the ideal case. How likely is it that we would *not* achieve the required 22 good pivots after 88 recursions?

Binomial probability distributions can answer that, and the answer is about 10^-18. (n is 88, k is 21, p is 0.6875) Your user is about a thousand times more likely to be struck by lightning in the 1 second it takes to click [SORT] than they are to see that 5,000 item sort run *any worse* than 10*ideal case. This chance gets smaller as the dataset gets larger. Here are some array sizes and their corresponding chances to run longer than 10*ideal:

- Array of 640 items: 10^-13 (requires 15 good pivot points out of 60 tries)
- Array of 5,000 items: 10^-18 (requires 22 good pivots out of 88 tries)
- Array of 40,000 items:10^-23 (requires 29 good pivots out of 116)

Remember that this is with 2 conservative assumptions that are worse than reality. So actual performance is better yet, and the balance of the remaining probability is closer to ideal than not.

Finally, as others have mentioned, even these absurdly unlikely cases can be eliminated by switching to a heap sort if the recursion stack goes too deep. So the TLDR is that, for good implementations of QuickSort, the worst case *does not really exist* because it has been engineered out and execution completes in O(n*logn) time.

Share Improve this answer Follow

answered Sep 25, 2015 at 3:50

Lance Wisely
85 ● 1 ● 7

2 "the existing great answers" -- which are those? I can't locate them. – Jim Balter Nov 3, 2018 at 12:47

Do any variations of Quick Sort notify the comparison function about partitions, in such a way that would allow it to exploit situations where a substantial portion of the key will be the same for all items in a partition? – supercat Feb 11, 2020 at 23:02

If Quicksort uses uses any deterministic pseudo-random method for choosing pivots, someone who can submit a list that will need to be sorted could deliberately arrange the data so as to yield worst-case performance. – supercat Mar 15, 2023 at 23:14

From the Wikipedia entry on Quicksort:

7

> Quicksort also competes with mergesort, another recursive sort algorithm but with the benefit of worst-case Θ(nlogn) running time. Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network attached storage. Although quicksort can be written to operate on linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that, when operating on arrays, it requires Θ(n) auxiliary space in the best case, whereas the variant of quicksort with in-place partitioning and tail recursion uses only Θ(logn) space. (Note that when operating on linked lists, mergesort only requires a small, constant amount of auxiliary storage.)

Share Improve this answer Follow

answered Sep 16, 2008 at 8:42

gnobal
18.5k ● 4 ● 33 ● 36

6

Quicksort is the fastest sorting algorithm in practice but has a number of pathological cases that can make it perform as badly as O(n2).

Heapsort is guaranteed to run in O(n*ln(n)) and requires only finite additional storage. But there are many citations of real world tests which show that heapsort is significantly slower than quicksort on average.

Share Improve this answer

Follow

answered Sep 16, 2008 at 8:41

Niyaz

54.8k ● 56 ● 152 ● 183

6

Quicksort is NOT better than mergesort. With O(n^2) (worst case that rarely happens), quicksort is potentially far slower than the O(nlogn) of the merge sort. Quicksort has less overhead, so with small n and slow computers, it is better. But computers are so fast today that the additional overhead of a mergesort is negligible, and the risk of a very slow quicksort far outweighs the insignificant overhead of a mergesort in most cases.

In addition, a mergesort leaves items with identical keys in their original order, a useful attribute.

Share Improve this answer

Follow

edited Nov 21, 2014 at 4:08

answered Sep 16, 2008 at 22:29

xpda

15.8k ● 8 ● 53 ● 83

2 Your second sentence says "...mergesort is potentially far slower than ... mergesort". The first reference should presumaably be to quicksort. – Jonathan Leffler Sep 28, 2008 at 2:26

Merge sort is only stable if the merge algorithm is stable; this is not guaranteed. – Clearer Dec 21, 2014 at 19:05

@Clearer It's guaranteed if `<=` is used for comparisons rather than `<`, and there's no reason not to. – Jim Balter Nov 3, 2018 at 12:49

@JimBalter I could easily come up with an unstable merge algorithm (quicksort for example, would serve that role). The reason why quick sort is faster than merge sort in many cases is *not* because of reduced overhead but because of how quicksort accesses data, which is a lot more cache friendly than a standard mergesort. – Clearer Nov 4, 2018 at 16:46

@Clearer quicksort is not a merge sort ... your Dec 21 '14 statement that I responded to was strictly about merge sort and whether it is stable. quicksort and which is faster is not at all relevant to your comment or my response. End of discussion for me ... over and out. – Jim Balter Nov 4, 2018 at 17:01

5

Wikipedia's explanation is:

> Typically, quicksort is significantly faster in practice than other Θ(nlogn) algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the probability of requiring quadratic time.

Quicksort

Mergesort

I think there are also issues with the amount of storage needed for Mergesort (which is Ω(n)) that quicksort implementations don't have. In the worst case, they are the same amount of algorithmic time, but mergesort requires more storage.

Share Improve this answer Follow

answered Sep 16, 2008 at 8:43
Mat Mannion
3,365 ● 2 ● 32 ● 31

Worst case of quicksort is O(n), mergesort O(n log n) - so ther'es a big difference there. – paul23 Sep 4, 2016 at 12:43

1 worst case quicksort is O(n^2) - can't edit my previous comment and made a typo – paul23 Sep 4, 2016 at 13:54

@paul23 comments can be deleted. Also, the answer already addressed your point: "in most real-world data it is possible to make design choices which minimize the probability of requiring quadratic time" – Jim Balter Nov 3, 2018 at 13:13

4

Why Quicksort is good?

- QuickSort takes N^2 in worst case and NlogN average case. The worst case occurs when data is sorted. This can be mitigated by random shuffle before sorting is started.
- QuickSort doesn't takes extra memory that is taken by merge sort.

- If the dataset is large and there are identical items, complexity of Quicksort reduces by using 3 way partition. More the no of identical items better the sort. If all items are identical, it sorts in linear time. [This is default implementation in most libraries]

Is Quicksort always better than Mergesort?

Not really.

- Mergesort is stable but Quicksort is not. So if you need stability in output, you would use Mergesort. Stability is required in many practical applications.
- Memory is cheap nowadays. So if extra memory used by Mergesort is not critical to your application, there is no harm in using Mergesort.

Note: In java, Arrays.sort() function uses Quicksort for primitive data types and Mergesort for object data types. Because objects consume memory overhead, so added a little overhead for Mergesort may not be any issue for performance point of view.

Reference: Watch the QuickSort videos of Week 3, Princeton Algorithms Course at Coursera

Share Improve this answer Follow

answered Nov 8, 2013 at 7:30

Sanjeev Kumar Dangi
7,235 ● 7 ● 29 ● 30

1 "This can be mitigated by random shuffle before sorting is started. " - er, no, that would be expensive. Instead, use

random pivots. – Jim Balter Nov 3, 2018 at 12:56

Very much agreed with @JimBalter - Why would we shuffle the data? If anything, I believe one could argue that nearly-sorted data actually sorts faster since it requires fewer write operations. As Jim mentioned, choosing a better pivot is a good fix. Median of three random numbers can be useful. – rinogo Sep 19, 2022 at 21:33

4

This is a pretty old question, but since I've dealt with both recently here are my 2c:

Merge sort needs on average ~ N log N comparisons. For already (almost) sorted sorted arrays this gets down to 1/2 N log N, since while merging we (almost) always select "left" part 1/2 N of times and then just copy right 1/2 N elements. Additionally I can speculate that already sorted input makes processor's branch predictor shine but guessing almost all branches correctly, thus preventing pipeline stalls.

Quick sort on average requires ~ 1.38 N log N comparisons. It does not benefit greatly from already sorted array in terms of comparisons (however it does in terms of swaps and probably in terms of branch predictions inside CPU).

My benchmarks on fairly modern processor shows the following:

When comparison function is a callback function (like in qsort() libc implementation) quicksort is slower than mergesort by 15% on random input and 30% for already sorted array for 64 bit integers.

On the other hand if comparison is not a callback, my experience is that quicksort outperforms mergesort by up to 25%.

However if your (large) array has a very few unique values, merge sort starts gaining over quicksort in any case.

So maybe the bottom line is: if comparison is expensive (e.g. callback function, comparing strings, comparing many parts of a structure mostly getting to a second-third-forth "if" to make difference) - the chances are that you will be better with merge sort. For simpler tasks quicksort will be faster.

That said all previously said is true: - Quicksort can be N^2, but Sedgewick claims that a good randomized implementation has more chances of a computer performing sort to be struck by a lightning than to go N^2 - Mergesort requires extra space

Share Improve this answer Follow

answered Aug 25, 2016 at 23:55

virco
277 ● 2 ● 4

Does qsort beats mergesort even for sorted inputs if comparison is cheap? – eonil May 17, 2019 at 20:07

4

Unlike Merge Sort Quick Sort doesn't uses an auxilary space. Whereas Merge Sort uses an auxilary space O(n). But Merge Sort has the worst case time complexity of O(nlogn) whereas the worst case complexity of Quick Sort is O(n^2) which happens when the array is already is sorted.

Share Improve this answer Follow

edited Mar 24, 2017 at 6:41

answered Aug 26, 2016 at 6:56

Shantam Mittal

383 ● 4 ● 8

No, QuickSort's worst case does not happen when the array is already sorted, unless you use the first or last item as the pivot, but no one does that. – Jim Balter Nov 3, 2018 at 12:30

3

The answer would slightly tilt towards quicksort w.r.t to changes brought with DualPivotQuickSort for primitive values . It is used in **JAVA 7** to sort in **java.util.Arrays**

```
It is proved that for the Dual-Pivot Quicksort the
average number of
comparisons is 2*n*ln(n), the average number of
swaps is 0.8*n*ln(n),
whereas classical Quicksort algorithm has
2*n*ln(n) and 1*n*ln(n)
respectively. Full mathematical proof see in
attached proof.txt
and proof_add.txt files. Theoretical results are
```

```
also confirmed
by experimental counting of the operations.
```

You can find the JAVA7 implmentation here - http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7-b147/java/util/Arrays.java

Further Awesome Reading on DualPivotQuickSort - http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628

Share Improve this answer Follow

edited Apr 3, 2013 at 14:44

answered Apr 3, 2013 at 14:31

appbootup
9,537 ● 3 ● 35 ● 65

3

In merge-sort, the general algorithm is:

1. Sort the left sub-array
2. Sort the right sub-array
3. Merge the 2 sorted sub-arrays

At the top level, merging the 2 sorted sub-arrays involves dealing with N elements.

One level below that, each iteration of step 3 involves dealing with N/2 elements, but you have to repeat this process twice. So you're still dealing with 2 * N/2 == N elements.

One level below that, you're merging 4 * N/4 == N elements, and so on. Every depth in the recursive stack involves merging the same number of elements, across all calls for that depth.

Consider the quick-sort algorithm instead:

1. Pick a pivot point
2. Place the pivot point at the correct place in the array, with all smaller elements to the left, and larger elements to the right
3. Sort the left-subarray
4. Sort the right-subarray

At the top level, you're dealing with an array of size N. You then pick one pivot point, put it in its correct position, and can then ignore it completely for the rest of the algorithm.

One level below that, you're dealing with 2 sub-arrays that have a combined size of N-1 (ie, subtract the earlier pivot point). You pick a pivot point for each sub-array, which comes up to 2 additional pivot points.

One level below that, you're dealing with 4 sub-arrays with combined size N-3, for the same reasons as above.

Then N-7... Then N-15... Then N-32...

The depth of your recursive stack remains approximately the same (logN). With merge-sort, you're always dealing with a N-element merge, across each level of the recursive stack. With quick-sort though, the number of elements that you're dealing with diminishes as you go down the stack. For example, if you look at the depth midway through the recursive stack, the number of elements you're dealing with is N - 2^((logN)/2)) == N - sqrt(N).

Disclaimer: On merge-sort, because you divide the array into 2 exactly equal chunks each time, the recursive depth is exactly logN. On quick-sort, because your pivot point is unlikely to be exactly in the middle of the array, the depth of your recursive stack may be slightly greater than logN. I haven't done the math to see how big a role this factor and the factor described above, actually play in the algorithm's complexity.

Share Improve this answer Follow

answered Mar 12, 2016 at 13:51

RvPr
1,144 ● 1 ● 10 ● 29

That the pivots aren't part of the sorts at the next level isn't why QS is more performant. See the other answers for additional insight. – Jim Balter Nov 3, 2018 at 12:37

@JimBalter Which "other answers" are you referring to? The top answer just says that QS "requires little additional space and exhibits good cache locality" but gives no explanation as to why that is, nor does it provide any citations. The 2nd answer simply says that merge-sort is better for larger data-sets – RvPr Jun 13, 2019 at 20:01

You're moving the goalposts, from why QS is more performant to explaining basic facts about how it operates. Answers to other questions do that: stackoverflow.com/questions/9444714/… ... I hope that's enough for you; I won't respond further. – Jim Balter Jun 14, 2019 at 1:30

2

Quicksort has a better average case complexity but in some applications it is the wrong choice. Quicksort is vulnerable to denial of service attacks. If an attacker can choose the input to be sorted, he can easily construct a set that takes the worst case time complexity of o(n^2).

Mergesort's average case complexity and worst case complexity are the same, and as such doesn't suffer the same problem. This property of merge-sort also makes it the superior choice for real-time systems - precisely because there aren't pathological cases that cause it to run much, much slower.

I'm a bigger fan of Mergesort than I am of Quicksort, for these reasons.

Share Improve this answer Follow

edited Sep 16, 2008 at 9:04

answered Sep 16, 2008 at 8:42

Simon Johnson
7,902 ● 5 ● 38 ● 50

2 How does Quicksort have a better average case complexity? They are both O(nlgn). I would argue that an attacker wont be providing input to any sorting algorithm...but in the interest of not assuming security by obscurity, lets assume he could. While n^2 running time is worse than nlgn, it is not sufficiently worse that a web server would crash based on a single attack. In fact, the DOS argument is pretty much null, because any web server is vulnerable to a DDOS attack, and it is more likely for an attacker to use a distributed network of hosts, all TCP SYN flooding. – CaTalyst.X Apr 15, 2013 at 20:44

"Quicksort has a better average case complexity" -- no it doesn't. – Jim Balter Nov 3, 2018 at 13:09

2

That's hard to say.The worst of MergeSort is n(log2n)-n+1,which is accurate if n equals 2^k(I have already proved this).And for any n,it's between (n lg n - n + 1) and (n lg n + n + O(lg n)).But for quickSort,its best is nlog2n(also n equals 2^k).If you divide Mergesort by quickSort,it equals one when n is infinite.So it's as if the worst case of MergeSort is better than the best case of QuickSort,why do we use quicksort?But remember,MergeSort is not in place,it require 2n memeroy space.And MergeSort also need to do many array copies,which we don't include in the analysis of algorithm.In a word,MergeSort is really faseter than quicksort in theroy,but in reality you need to consider memeory space,the cost of array copy,merger is slower than quick sort.I once made an experiment where I was given 1000000 digits in java by Random class,and it took 2610ms by mergesort,1370ms by quicksort.

Share Improve this answer Follow

answered Sep 10, 2011 at 15:33

Peter
29 ● 1

2

Quick sort is worst case O(n^2), however, the average case consistently out performs merge sort. Each algorithm is O(nlogn), but you need to remember that when talking about Big O we leave off the lower complexity factors. Quick sort has significant improvements over merge sort when it comes to constant factors.

Merge sort also requires O(2n) memory, while quick sort can be done in place (requiring only O(n)). This is another reason that quick sort is generally preferred over merge sort.

Extra info:

The worst case of quick sort occurs when the pivot is poorly chosen. Consider the following example:

[5, 4, 3, 2, 1]

If the pivot is chosen as the smallest or largest number in the group then quick sort will run in O(n^2). The probability of choosing the element that is in the largest or smallest 25% of the list is 0.5. That gives the algorithm a

0.5 chance of being a good pivot. If we employ a typical pivot choosing algorithm (say choosing a random element), we have 0.5 chance of choosing a good pivot for every choice of a pivot. For collections of a large size the probability of always choosing a poor pivot is 0.5 * n. Based on this probability quick sort is efficient for the average (and typical) case.

Share Improve this answer Follow

edited Feb 13, 2014 at 16:33

Richard
60.8k ● 39 ● 194 ● 273

answered Jul 9, 2013 at 20:12

Wade Anderson
2,519 ● 1 ● 21 ● 22

O(2n) == O(n). The correct statement is that Mergesort needs O(n) additional memory (more specifically, it needs n/2 auxilliary memory). And this isn't true for linked lists.
– Jim Balter Nov 3, 2018 at 12:54

@JimBalter Sir, would you mind sharing your brilliant and worthwhile ideas with us about their perfomances as an answer of the question? Thanks in advance.
– Soner from The Ottoman Empire Nov 28, 2018 at 10:51

2

When I experimented with both sorting algorithms, by counting the number of recursive calls, quicksort consistently has less recursive calls than mergesort. It is because quicksort has pivots, and pivots are not included in the next recursive calls. That way quicksort can reach recursive base case more quicker than mergesort.

Share Improve this answer

Follow

answered Feb 12, 2017 at 1:49

Aldian Fazrihady
149 ● 1 ● 1 ● 6

Pivots have nothing to do with why QS has fewer recursive calls ... it's because half of QS's recursion is tail recursion, which can be eliminated. – Jim Balter Nov 3, 2018 at 12:32

1

While they're both in the same complexity class, that doesn't mean they both have the same runtime. Quicksort is usually faster than mergesort, just because it's easier to code a tight implementation and the operations it does can go faster. It's because that quicksort is generally faster that people use it instead of mergesort.

However! I personally often will use mergesort or a quicksort variant that degrades to mergesort when quicksort does poorly. Remember. Quicksort is only O(n log n) on *average*. It's worst case is O(n^2)! Mergesort is always O(n log n). In cases where realtime performance or responsiveness is a must and your input data could be coming from a malicious source, **you should not use plain quicksort.**

Share Improve this answer

Follow

answered Sep 16, 2008 at 8:44

DJ Capelis
959 ● 5 ● 8

1

All things being equal, I'd expect most people to use whatever is most conveniently available, and that tends to be qsort(3). Other than that quicksort is known to be very fast on arrays, just like mergesort is the common choice for lists.

What I'm wondering is why it's so rare to see **radix** or bucket sort. They're O(n), at least on linked lists and all it takes is some method of converting the key to an ordinal number. (strings and floats work just fine.)

I'm thinking the reason has to do with how computer science is taught. I even had to demonstrate to my lecturer in Algorithm analysis that it was indeed possible to sort faster than O(n log(n)). (He had the proof that you can't *comparison* sort faster than O(n log(n)), which is true.)

In other news, floats can be sorted as integers, but you have to turn the negative numbers around afterwards.

Edit: Actually, here's an even more vicious way to sort floats-as-integers: http://www.stereopsis.com/radix.html. Note that the bit-flipping trick can be used regardless of what sorting algorithm you actually use...

Share Improve this answer Follow

edited Sep 28, 2008 at 1:14

answered Sep 28, 2008 at 0:45

Anders Eurenius
4,226 ● 2 ● 25 ● 20

1 I've seen my share of radix sorts. But it's pretty hard to use because if analyzed correctly, its runtime is *not* O(n) as it depends on more than the number of input elements. In general, it's very hard to make that kind of strong predictions that radix sort needs to be efficient about the input. – Konrad Rudolph Oct 20, 2008 at 9:14

It *is* O(n), where n is the *total* input size, that is, including the size of the elements. It's true that you can implement it so you have to pad with a lot of zeroes, but it's nonsense to use a poor implementation for comparison. (That said, implementation can be hard, ymmv.) – Anders Eurenius Oct 20, 2008 at 15:31

Note that if you're using GNU libc, `qsort` is a merge sort. – Jason Orendorff Dec 10, 2009 at 0:44

Er, to be precise, it's a merge sort unless the necessary temporary memory can't be allocated. cvs.savannah.gnu.org/viewvc/libc/stdlib/… – Jason Orendorff Dec 10, 2009 at 0:52

1

Small additions to quick vs merge sorts.

Also it can depend on kind of sorting items. If access to items, swap and comparisons is not simple operations, like comparing integers in plane memory, then merge sort can be preferable algorithm.

For example , we sort items using network protocol on remote server.

Also, in custom containers like "linked list", the are no benefit of quick sort.
1. Merge sort on linked list, don't need additional memory.
2. Access to elements in quick sort is not sequential (in memory)

Share Improve this answer Follow

answered Nov 5, 2014 at 9:32

minorlogic

1,900 ● 2 ● 19 ● 26

0

Quick sort is an in-place sorting algorithm, so its better suited for arrays. Merge sort on the other hand requires extra storage of O(N), and is more suitable for linked lists.

Unlike arrays, in liked list we can insert items in the middle with O(1) space and O(1) time, therefore the merge operation in merge sort can be implemented without any extra space. However, allocating and de-allocating extra space for arrays have an adverse effect on the run time of merge sort. Merge sort also favors linked list as data is accessed sequentially, without much random memory access.

Quick sort on the other hand requires a lot of random memory access and with an array we can directly access the memory without any traversing as required by linked lists. Also quick sort when used for arrays have a good locality of reference as arrays are stored contiguously in memory.

Even though both sorting algorithms average complexity is O(NlogN), usually people for ordinary tasks uses an array for storage, and for that reason quick sort should be the algorithm of choice.

EDIT: I just found out that merge sort worst/best/avg case is always nlogn, but quick sort can vary from n2(worst case when elements are already sorted) to nlogn(avg/best case when pivot always divides the array in two halves).

edited Jul 13, 2016 at 19:31

answered Jun 28, 2016 at 19:49

Saad

897 ● 1 ● 10 ● 21

0

Consider time and space complexity both. For Merge sort : Time complexity : O(nlogn) , Space complexity : O(nlogn)

For Quick sort : Time complexity : O(n^2) , Space complexity : O(n)

Now, they both win in one scenerio each. But, using a random pivot you can almost always reduce Time complexity of Quick sort to O(nlogn).

Thus, Quick sort is preferred in many applications instead of Merge sort.

Share Improve this answer Follow

answered Dec 23, 2018 at 14:06

pankaj
530 ● 6 ● 12

-1

In c/c++ land, when not using stl containers, I tend to use quicksort, because it is built into the run time, while mergesort is not.

So I believe that in many cases, it is simply the path of least resistance.

In addition performance can be much higher with quick sort, for cases where the entire dataset does not fit into the working set.

Share Improve this answer Follow

edited Sep 27, 2008 at 23:56

answered Sep 17, 2008 at 2:00

EvilTeach
28.8k ● 21 ● 88 ● 144

3 Actually, if it is the qsort() library function you are talking about, it may or may not be implemented as quicksort. – Thomas Padron-McCarthy Oct 12, 2008 at 7:03

3 Konrad, sorry to be a bit anal about this, but where do you find that guarantee? I can't find it in the ISO C standard, or in the C++ standard. – Thomas Padron-McCarthy Oct 21, 2008 at 11:12

2 GNU libc's `qsort` is a merge sort unless the number of elements is truly gigantic or the temporary memory can't be allocated. cvs.savannah.gnu.org/viewvc/libc/stdlib/…
– Jason Orendorff Dec 10, 2009 at 0:49

-5

One of the reason is more philosophical. Quicksort is Top->Down philosophy. With n elements to sort, there are n! possibilities. With 2 partitions of m & n-m which are mutually exclusive, the number of possibilities go down in several orders of magnitude. m! * (n-m)! is smaller by several orders than n! alone. imagine 5! vs 3! *2!. 5! has 10 times more possibilities than 2 partitions of 2 & 3 each . and extrapolate to 1 million factorial vs 900K!*100K! vs. So instead of worrying about establishing any order within a range or a partition,just establish order at a broader level in partitions and reduce the possibilities within a partition. Any order established earlier within a range will be disturbed later if the partitions themselves are not mutually exclusive.

Any bottom up order approach like merge sort or heap sort is like a workers or employee's approach where one starts comparing at a microscopic level early. But this order is bound to be lost as soon as an element in between them is found later on. These approaches are very stable & extremely predictable but do a certain amount of extra work.

Quick Sort is like Managerial approach where one is not initially concerned about any order , only about meeting a broad criterion with No regard for order. Then the partitions are narrowed until you get a sorted set. The real challenge in Quicksort is in finding a partition or criterion in the dark when you know nothing about the elements to sort. That is why we either need to spend some effort to find a median value or pick 1 at random or some arbitrary "Managerial" approach . To find a perfect median can take significant amount of effort and leads to a stupid bottom up approach again. So Quicksort says just a pick a random pivot and hope that it will be somewhere in the middle or do some work to find median of 3 , 5 or something more to find a better median but do not plan to be perfect & don't waste any time in initially ordering. That seems to do well if you are lucky or sometimes degrades to n^2 when you don't get a median but just take a chance. Any way data is random. right. So I agree more with the top ->down logical approach of quicksort & it turns out that the chance it takes about pivot selection & comparisons that it saves earlier seems to work better more times than any meticulous & thorough stable bottom ->up approach like merge sort. But

Share Improve this answer Follow

edited Dec 10, 2017 at 23:21

answered Dec 10, 2017 at 22:57

Winter Melon
95 ● 1 ● 6

---

quicksort benefits from randomness of pivot selection. The random pivot would naturally tend toward 50:50 partition and is unlikely to be consistently towards one of the extremes. The constant factor of nlogn is fairly low till average partitioning is 60-40 or even till 70-30. – Winter Melon Jan 1, 2018 at 22:05

1 This is complete nonsense. quicksort is used because of its performance, not "philosophy" ... and the claims about "order is bound to be lost" is simply false. – Jim Balter Nov 3, 2018 at 12:26