

Learn, Share, Build

Each month, over 50 million developers come to Stack Overflow to learn, share their knowledge, and build their careers.

Join the world's largest developer community.

Google

Facebook


OR

Which sort algorithm works best on mostly sorted data? [closed]

Which sorting algorithm works best on mostly sorted data?


algorithm sorting

edited Oct 10 '15 at 14:37

 **gsamaras**

37.6k 21 64 131

asked Oct 20 '08 at 21:38

 **graphics**

792 2 6 4

closed as too broad by [Henk Holterman](#), [hexacyanide](#), [Ilya](#), [toniedzwiedz](#), [Toji](#) Oct 23 '13 at 19:58

Please edit the question to limit it to a specific problem with enough detail to identify an adequate answer. Avoid asking multiple distinct questions at once. See the [How to Ask](#) page for help clarifying this question.

If this question can be reworded to fit the rules in the [help center](#), please [edit the question](#).

Guessing from lack of context - you are asking about an in-memory sort with no requirement to spill intermediate results to disk? – [Jonathan Leffler](#) Oct 20 '08 at 22:03

1 [According to these animations](#) insertion sorting works best on mostly sorted data. – [dopple](#) Apr 3 '12 at 9:53

20 Answers

Based on the highly scientific method of watching [animated gifs](#) I would say Insertion and Bubble sorts are good candidates.

edited Nov 11 '10 at 16:12

 **Dominic Rodger**

68.4k 18 166 189

answered Oct 20 '08 at 21:41

 **Tom Ritter**

74.4k 27 121 158

14 that's an excellent link by the way, kudos and a +1 – [ninesided](#) Oct 20 '08 at 21:44

4 Bubble sort is terrible. It is always $O(n^2)$. At least take that out of your answer for it to be right please. – [jnguy](#) Oct 20 '08 at 22:03

66 jnguy, that is just plain wrong. I think you need to re-take your algorithms class. On nearly sorted data (it's adaptive case) it is $O(N)$. However, it takes 2 passes through the data and Insertion only takes 1 for nearly sorted data, which makes Insertion the winner. Bubble is still good though – [mmcdole](#) Oct 21 '08 at 1:59

3 Performance degrades really badly if your data is ever not nearly sorted though. I would still not use it, personally. – [Blorgbeard](#) Oct 21 '08 at 4:39

5 That link was broken when I tried it. Try this instead: [sorting-algorithms.com](#) – [Michael La Voie](#) Jul 31 '09 at 20:06

Only a few items => INSERTION SORT

Items are mostly sorted already => INSERTION SORT

Concerned about worst-case scenarios => HEAP SORT

Interested in a good average-case result => QUICKSORT

Items are drawn from a dense universe => BUCKET SORT

Desire to write as little code as possible => INSERTION SORT

answered Nov 25 '10 at 10:40



Jiaji Li

781 6 5

1 That is exactly the kind of answer I have been looking for, I read books but I don't seem to find any clear explanation for selection of algorithms at particular cases, could you please elaborate this or pass a link so that I can dig into it a little more? Thanks – [Simran kaur](#) Jun 24 '14 at 3:53

32 Concerned about being too quick => BOGO SORT – [Alexander](#) Apr 23 '15 at 23:18

4 You should add "Data is already sorted by another criterion => MERGE SORT" – [Jim Hunziker](#) Jan 22 '16 at 18:23

@AMomchilov Nah. Look up "Worstsort." – [ApproachingDarknessFish](#) Apr 27 '16 at 5:54

timsort

Timsort is "an adaptive, stable, natural mergesort" with "supernatural performance on many kinds of partially ordered arrays" (less than $\lg(N!)$ comparisons needed, and as few as $N-1$). Python's built-in `sort()` has used this algorithm for some time, apparently with good results. It's specifically designed to detect and take advantage of partially sorted subsequences in the input, which often occur in real datasets. It is often the case in the real world that comparisons are much more expensive than swapping items in a list, since one typically just swaps pointers, which very often makes timsort an excellent choice. However, if you know that your comparisons are always very cheap (writing a toy program to sort 32-bit integers, for instance), other algorithms exist that are likely to perform better. The easiest way to take advantage of timsort is of course to use Python, but since Python is open source you might also be able to borrow the code. Alternately, the description above contains more than enough detail to write your own implementation.

answered Oct 21 '08 at 4:17



zaphod

2,744 18 23

14 $\log(n!)$ is $O(n \log n)$ therefore It is not "supernatural". – [jfs](#) Oct 29 '08 at 9:32

Here's the Java implementation coming in JDK7:

cr.openjdk.java.net/~martin/webrevs/openjdk7/timsort/raw_files/... – [Tim](#) Aug 9 '09 at 15:06

$\log(n!)$ is not fast. [wolframalpha.com/input/?i=plot\[log\(N!\){N,0,1000}\]](http://wolframalpha.com/input/?i=plot[log(N!){N,0,1000}]) – [Behrooz](#) Dec 10 '09 at 10:36

9 @J.F. Sebastian: timsort is much faster than $\lg(n!)$ comparisons on an almost-sorted array, all the way down to $O(n)!$ | @behrooz: No comparison sort can have an average case of better than $O(n \log n)$, and $\lg(n!)$ is $O(n \log n)$. So timsort's worst case is asymptotically no worse than that of any other comparison sort. Furthermore its best case is better than or equal to any other comparison sort. – [Artelius](#) Dec 12 '09 at 0:14

3 Timsort is still $O(n \log n)$ in the worst case, but its good-cases are quite pleasing. Here's a comparison, with some graphs: stromberg.dnsalias.org/~strombrg/sort-comparison Note that timsort in Cython wasn't nearly as fast as Python's built in timsort in C. – [user1277476](#) Aug 16 '12 at 21:35

Insertion sort with the following behavior:

1. For each element k in slots $1..n$, first check whether $e_l[k] \geq e_l[k-1]$. If so, go to next element. (Obviously skip the first element.)
2. If not, use binary-search in elements $1..k-1$ to determine the insertion location, then scoot the elements over. (You might do this only if $k > \tau$ where τ is some threshold value; with small k this is overkill.)

This method makes the least number of comparisons.

answered Oct 20 '08 at 21:48



Jason Cohen

53.1k 22 98 105

I think bubble sort might beat this if the number of unsorted elements is very small (like, one or two), but in general this strikes me as probably the best solution. – [Sol](#) Oct 20 '08 at 21:55

Because of step 1, for any elements that are already sorted there is exactly one compare and zero data-moves, which is obviously the best you can do. Step 2 is the one you could improve on, but bubble will move the same number of elements and *might* have more compares, depending on your impl. – [Jason Cohen](#) Oct 20 '08 at 22:00

Actually, on further thought I think bubble sort is stronger than I was thinking. It's actually a fairly tricky question. For instance, if you take the case where the list is entirely sorted except the element which should be last is first, bubble sort will vastly outperform what you describe. – Sol Oct 20 '08 at 22:22

I tried to implement this but the binary search is not much of an improvement since you still have to move the entire block to insert the element. So instead of $2 \times \text{range}$ you get $\text{range} + \log b(\text{range})$. – this Jan 20 '14 at 9:41

Try introspective sort. <http://en.wikipedia.org/wiki/Introsort>

It's quicksort based, but it avoids the worst case behaviour that quicksort has for nearly sorted lists.

The trick is that this sort-algorithm detects the cases where quicksort goes into worst-case mode and switches to heap or merge sort. Nearly sorted partitions are detected by some non naive partition method and small partitions are handled using insertion sort.

You get the best of all major sorting algorithms for the cost of a more code and complexity. And you can be sure you'll never run into worst case behaviour no matter how your data looks like.

If you're a C++ programmer check your `std::sort` algorithm. It may already use introspective sort internally.

answered Oct 20 '08 at 22:29



Nils Pipenbrinck

59.3k 18 128 202

insertion or shell sort!

answered Oct 20 '08 at 21:43



ninesided

17.5k 13 68 107

[Splaysort](#) is an obscure sorting method based on [splay trees](#), a type of adaptive binary tree. Splaysort is good not only for partially sorted data, but also partially reverse-sorted data, or indeed any data that has any kind of pre-existing order. It is $O(n \lg n)$ in the general case, and $O(n)$ in the case where the data is sorted in some way (forward, reverse, organ-pipe, etc.).

Its great advantage over insertion sort is that it doesn't revert to $O(n^2)$ behaviour when the data isn't sorted at all, so you don't need to be absolutely sure that the data is partially sorted before using it.

Its disadvantage is the extra space overhead of the splay tree structure it needs, as well as the time required to build and destroy the splay tree. But depending on the size of data and amount of pre-sortedness that you expect, the overhead may be worth it for the increase in speed.

A [paper on splaysort](#) was published in Software--Practice & Experience.

answered Oct 21 '08 at 4:02



TimB

4,501 2 19 27

Dijkstra's smoothsort is a great sort on already-sorted data. It's a heapsort variant that runs in $O(n \lg n)$ worst-case and $O(n)$ best-case. I [wrote an analysis](#) of the algorithm, in case you're curious how it works.

Natural mergesort is another really good one for this - it's a bottom-up mergesort variant that works by treating the input as the concatenation of multiple different sorted ranges, then using the merge algorithm to join them together. You repeat this process until all of the input range is sorted. This runs in $O(n)$ time if the data is already sorted and $O(n \lg n)$ worst-case. It's very elegant, though in practice it isn't as good as some other adaptive sorts like Timsort or smoothsort.

answered Nov 9 '10 at 21:11



templatetypedef

232k 56 572 828

what's are the runtime constants of smoothsort compared to other sorting algorithms? (i.e.

runtime(smoothsort) / runtime(insertionsort) for the same data) – [Arne Babenhauserheide](#) May 2 at 23:03

Insertion sort takes time $O(n + \text{the number of inversions})$.

An inversion is a pair (i, j) such that $i < j$ && $a[i] > a[j]$. That is, an out-of-order pair.

One measure of being "almost sorted" is the number of inversions---one could take "almost sorted data" to mean data with few inversions. If one knows the number of inversions to be linear (for instance, you have just appended $O(1)$ elements to a sorted list), insertion sort takes $O(n)$ time.

answered Jun 2 '09 at 21:33



[Jonas Kölker](#)

5,559 1 30 47

If elements are already sorted or there are only few elements, it would be a perfect use case for Insertion Sort!

answered Oct 29 '12 at 11:42



[Roger](#)

626 1 9 22

As everyone else said, be careful of naive Quicksort - that can have $O(N^2)$ performance on sorted or nearly sorted data. Nevertheless, with an appropriate algorithm for choice of pivot (either random or median-of-three - see [Choosing a Pivot for Quicksort](#)), Quicksort will still work sanely.

In general, the difficulty with choosing algorithms such as insert sort is in deciding when the data is sufficiently out of order that Quicksort really would be quicker.

edited May 23 at 12:18



Community ♦

1 1

answered Oct 20 '08 at 22:00



[Jonathan Leffler](#)

498k 76 578 923

I'm not going to pretend to have all the answers here, because I think getting at the actual answers may require coding up the algorithms and profiling them against representative data samples. But I've been thinking about this question all evening, and here's what's occurred to me so far, and some guesses about what works best where.

Let N be the number of items total, M be the number out-of-order.

Bubble sort will have to make something like $2*M+1$ passes through all N items. If M is very small ($0, 1, 2?$), I think this will be very hard to beat.

If M is small (say less than $\log N$), insertion sort will have great average performance. However, unless there's a trick I'm not seeing, it will have very bad worst case performance. (Right? If the last item in the order comes first, then you have to insert every single item, as far as I can see, which will kill the performance.) I'm guessing there's a more reliable sorting algorithm out there for this case, but I don't know what it is.

If M is bigger (say equal or great than $\log N$), introspective sort is almost certainly best.

Exception to all of that: If you actually know ahead of time which elements are unsorted, then your best bet will be to pull those items out, sort them using introspective sort, and merge the two sorted lists together into one sorted list. If you could quickly figure out which items are out of order, this would be a good general solution as well -- but I haven't been able to figure out a simple way to do this.

Further thoughts (overnight): If $M+1 < N/M$, then you can scan the list looking for a run of N/M in a row which are sorted, and then expand that run in either direction to find the out-of-order items. That will take at most $2N$ comparisons. You can then sort the unsorted items, and do a sorted merge on the two lists. Total comparisons should be less than something like $4N+M \log_2(M)$, which is going to beat any non-specialized sorting routine, I think. (Even further thought: this is trickier than I was thinking, but I still think it's reasonably possible.)

Another interpretation of the question is that there may be many of out-of-order items, but they are very close to where they should be in the list. (Imagine starting with a sorted list and swapping every other item with the one that comes after it.) In that case I think bubble sort performs very well -- I think the number of passes will be proportional to the furthest out of

place an item is. Insertion sort will work poorly, because every out of order item will trigger an insertion. I suspect introspective sort or something like that will work well, too.

edited Oct 21 '08 at 12:34

answered Oct 21 '08 at 3:23



[Sol](#)

1,702 1 11 9

If you are in need of specific implementation for sorting algorithms, data structures or anything that have a link to the above, could I recommend you the excellent "[Data Structures and Algorithms](#)" project on CodePlex?

It will have everything you need without reinventing the wheel.

Just my little grain of salt.

answered Oct 21 '08 at 12:10



[Maxime Rouiller](#)

9,963 6 46 96

This nice collection of sorting algorithms for this purpose in the answers, seems to lack [Gnome Sort](#), which would also be suitable, and probably requires the least implementation effort.

answered Oct 15 '11 at 13:26



[haraldki](#)

3,031 16 42

Insertion sort is best case $O(n)$ on sorted input. And it is very close on mostly sorted input (better than quick sort).

answered Oct 20 '08 at 21:44



[jjnguy](#)

96.9k 38 242 299

ponder Try Heap. I believe it's the most consistent of the $O(n \lg n)$ sorts.

answered Oct 20 '08 at 21:49



[Paul Nathan](#)

27k 20 93 183

Consistency is not of concern here. Heapsort will give $O(n \lg n)$ even on sorted data, and is not really adaptive. Viable options can be: Insertion sort, Timsort and Bubblesort. – [Max](#) Feb 13 '13 at 7:39

Bubble-sort (or, safer yet, bi-directional bubble sort) is likely ideal for mostly sorted lists, though I bet a tweaked comb-sort (with a much lower initial gap size) would be a little faster when the list wasn't quite as perfectly sorted. Comb sort degrades to bubble-sort.

answered Oct 21 '08 at 12:59



[Brian](#)

17.5k 12 62 134

well it depends on use case. If you know which elements is changed, remove and insert will be the best case as far as I am concerned.

answered Aug 16 '12 at 19:52



[Helin Wang](#)

1,635 17 24

This "as far as I am concerned" test of algorithm efficiency brightened up my day :) Being serious, though, when writing "remove and insert" did you mean Insertion Sort (which was already mentioned in previous answers), or do you offer a new kind of algorithm? If so, please expand your answer. – [yonilavi](#) Apr 21 '15 at 13:05

Bubble sort is definitely the winner The next one on the radar would be insertion sort.

answered Sep 21 '12 at 11:48



vCillusion

617 8 15

4 post your answer with an explanation; – user1542476 Sep 21 '12 at 13:22

1 I would suggest you have a look at the available answers before posting to avoid duplicates. – [angainor](#) Sep 21 '12 at 21:49

Keep away from QuickSort - its very inefficient for pre-sorted data. Insertion sort handles almost sorted data well by moving as few values as possible.

answered Oct 20 '08 at 21:44



Werg38

702 1 5 11

-1 Every industrial implementation of Quicksort has a reasonable pivot selection – [Stephan Eggermont](#) Jan 21 '09 at 23:27

1 Yes, but no pivot selection is perfect unless it gets expensive. – [user1277476](#) Aug 16 '12 at 21:33
