Which is the best sort algorithm?

Rolfe Bozier

20-Mar-2014

Which is the best sort algorithm?

- ~~Answer: quicksort~~
- The real answer of course is, it depends:
  - How much data are you sorting?
  - Is your data random or partially ordered?
  - What format is your data in?
  - Is your sort key simple or complex?
  - Are your keys repeated or unique?
  - Is a recursive algorithm acceptable?
  - How much extra memory can you spare?
  - Is sort stability important?
- So, let's look at some possible solutions

Comparison sorts

- Most common sort algorithms require comparisons between pairs of keys
  - **insertion**, **selection**, **shell**, **merge**, **heap**, **quick**, **bubble**, etc.
- Although there are differences, they have a few things in common:
  - The average case *cannot* be better than O(n log n).
  - For some algorithms, the best case can be much better than the average case. If this is the case, the algorithm is described as *adaptive*.
  - For some algorithms, the worst case is much worse than average case.
- In most cases, the difference between best, average and worst cases is how the data is presented to the sort algorithm.
  - The main exception to this is **quicksort**, where it depends on how the random pivot points are chosen.

Choosing a comparison sort algorithm

- Each comparison sort algorithm has some different characteristics. These will determine which one is best for your situation:
  - Is it *stable*? That is, if two elements in your data compare as equal, will they retain their original relative ordering?
  - How much extra space does it require? Some sorts can be done in place, whereas other work better or require extra working memory.
  - How many key comparisons does it take (best, average, worst)?
  - How many swaps does it take? Some algorithms swap many elements around, whereas others just swap the minimal number of elements. If swapping is expensive, this could make a difference.
  - Is it adaptive?
  - What is the overhead? Some algorithms require little overhead apart from the comparison and swapping operations, some are heavier.

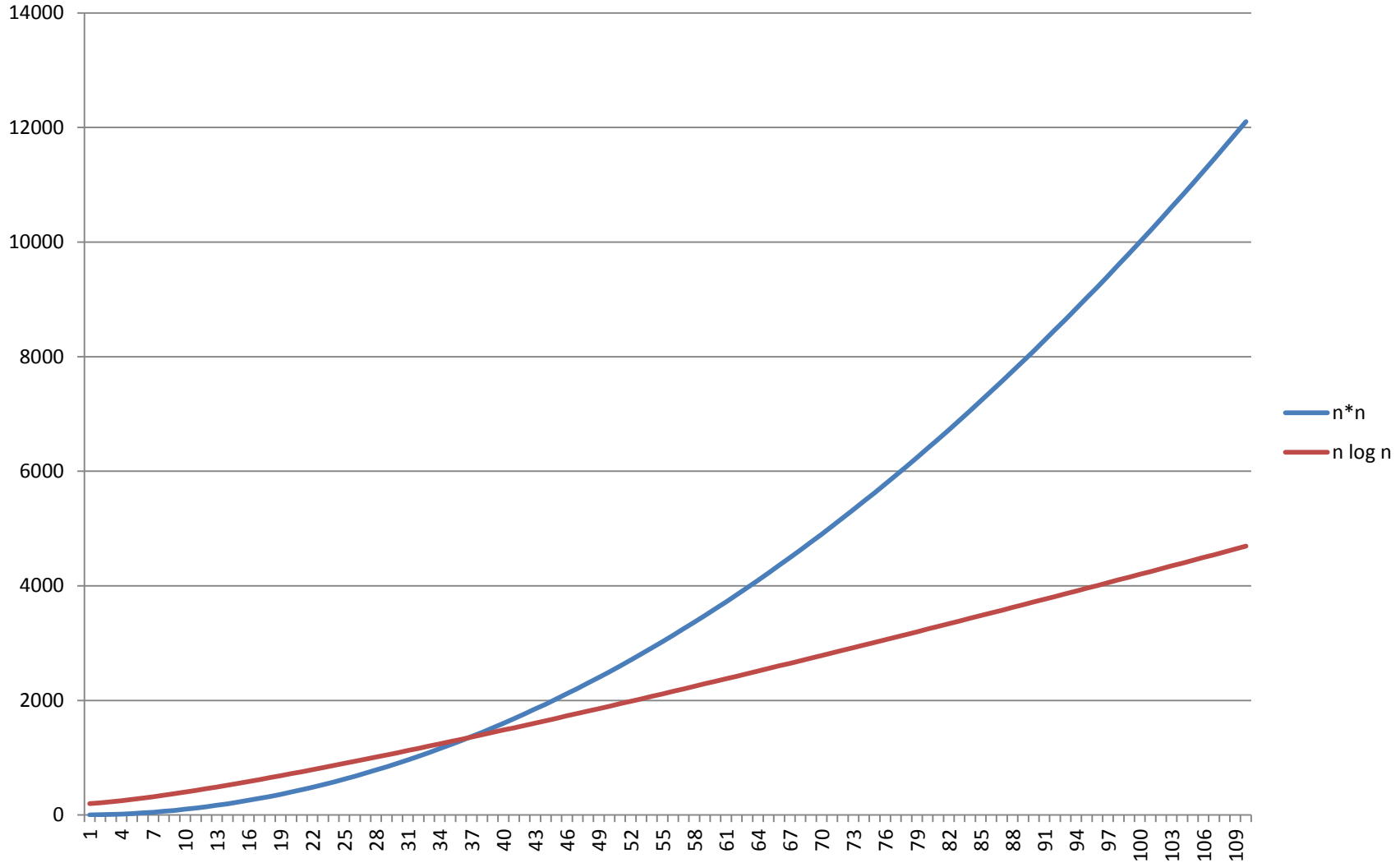
The main contenders

	Insertion	Selection	Bubble	Shell	Merge	Heap	Quick	Quick3	Introsort	Timsort
Stable?	**Yes**	No	**Yes**	No	**Yes**	No	No	No	No	**Yes**
Space	**1**	**1**	**1**	**1**	n or n log n	**1**	n log n	n log n	n log n	n
Best case	**n**	n^2	**n**	**n**	n log n	n log n	n log n	**n**	n log n	**n**
Avg case	n^2	n^2	n^2	$n^{1.5}$	**n log n**	**n log n**	**n log n**	**n log n**	**n log n**	**n log n**
Worst case	n^2	n^2	n^2	$n \log^2 n$	**n log n**	**n log n**	$(n^{2)}$	$(n^{2)}$	n log n	n log n
Swaps	n^2	**n**	n^2	n^2						
Adaptive?	If nearly sorted	If nearly sorted		If nearly sorted	If nearly sorted			If few unique keys		

Surely O(n log n) is what you want…?

- Time = K f(n) + C
- For large n, you definitely want to look at the O(n log n) average case algorithms.
- But... for small n, K and C can be much more important
- When you are sorting a relatively small set (many instances, otherwise why do you care?) a good alternate is **Insertion** sort. It requires no extra space, is adaptive, and has a low overhead.

Surely O(n log n) is what you want…?



Other types of sort algorithms

- There are other ways of sorting a set which don't require comparisons of sort keys.
- They are not so general purpose, and usually you need:
  - a good understanding of the range of sort key values
  - a good understanding of the typical key distribution in the
- Some examples include:
  - **Radix** sort: usually performed in numeric keys, and requires 1 pass for each digit in the key.
  - **Pigeonhole** and **Bucket** sort: 1 pass to "scatter" your data into an array, and 1 pass to gather the ordered results.
  - **Counting** sort: 1 pass to count the frequency of your keys, and 1 pass to copy them out.

Optimizing your sort (1)

- You've chosen an appropriate sort algorithm, but you need it to be faster. What can you do?
- *Don't* sort - especially if you don't need the ordered results. If you are doing some of the following, there are better algorithms:
  - Looking for, or counting duplicates in a set
  - Partioning a set
  - Looking for the smallest N elements
  - Looking for the Nth element
  - Only sorting the 1st N elements

Optimizing your sort (2)

- Optimize your data:
  - Can you reduce the amount of data you are sorting?
  - Can you skew your data towards the best case? Don't waste too much time, though, as it has to pay off in faster sorting.
- Optimize the comparison operation:
  - Can you compute an optimised sort key that is faster to compare?
  - Can you optimise the code in your comparison function? If your comparison function contains conditionals or loops, there is room for improvement.
  - Can you eliminate the overhead of calling the comparison function? The qsort() function requires an external comparison function, but if you implement your own you can in-line it...
  - Can you make the sort key integral? If you can fit it in a 32-bit or 64-bit value, you can reduce the comparison to a single instruction.

Optimizing your sort (3)

- Step back and re-appraise the problem:
  - Sometimes it is more efficient to replace many medium-sized sort operations with one big one.
  - If you are sorting and then de-duplicating, it may be more efficient to de-duplicate first (say, with a hash function), and then sort the results.
- Implement your own function:
  - You could whip up a simple version that uses Insertion sort for small sets, and falling back to quicksort for bigger ones.
  - Consider looking at **Timsort** (from python), or **Introsort** (from C++ STL).
  - Maybe look at a SIMD implementation. Some research suggest you can get a 3x speedup…
  - If sorting is a big issue, thinking about parallel sorting algorithms.
- Above everything else:
  - try
  - measure
  - repeat

Summary

- Just because you have qsort() doesn't mean you should use it
- Think twice if your data is "small"
- Consider a new adaptive sort (**Timsort**, **Introsort**)
- Consider a non-selection sort
- Consider not sorting
- Measure and optimise
- Don't be a slave to the O(n log n) paradigm!

- Some links:
  - http://en.wikipedia.org/wiki/Sorting_algorithm
  - http://www.sorting-algorithms.com/
  - Knuth v3 "Sorting and Searching"