Are algorithms with high time complexity ever used in the real world for small inputs? [closed]

Asked 1 year, 11 months ago Modified 1 year, 10 months ago

Viewed 6k times

67

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed last year.

The community reviewed whether to reopen this question last year and left it closed:

> Original close reason(s) were not resolved

Improve this question

Let's say we have a problem where a certain algorithm, let's call it algorithm_1, solves it in time complexity of `O(n^2)` and another algorithm, let's call it algorithm_2, solves it in time complexity `O(n)`, but in reality we see that for `n < 1000` algorithm_1 is faster and otherwise algorithm_2 is faster.

Why can't we just write code like this:

```
if ( n < 1000)
  do algorithm_1
else
  do algorithm_2
```

Is this a real thing programmers do or are there downsides for this?

On a smaller program this seems to be a good idea.

algorithm time-complexity implementation

Share

Improve this question

Follow

edited Feb 25, 2023 at 9:31

Peter Mortensen

31.6k • 22 • 109 • 133

asked Jan 12, 2023 at 22:16

A

Ak2399

837 • 4 • 9

24 Off the top of my head, for example, on most implementation, `qsort` does this with insertion sort and quicksort. Timsort is a mix, too. – Neil Jan 12, 2023 at 22:20

8 Yes this is a valid option if you have enough data to support your decision. But of course there are also downsides: Twice the amount of code is also twice the amount of code to maintain and twice as much possibility to make mistakes ... – derpirscher Jan 12, 2023 at 22:25

16 Standard libraries in many programming languages are **full** of if-else conditions exactly like the one you describe. – Stef Jan 12, 2023 at 22:31

6 I once used a bubble sort in production code, because I knew it was quick enough for our typical case of n=2 or n=4. The next version of the program added a new feature that completely invalidated that typical case assumption, and I'm sure I was cursed many times by the person who inherited that code. – Mark Ransom Jan 13, 2023 at 1:38

6 "Why cant we just write code like this" – It's hard to answer this question without knowing why you think that we can't write code like this, especially since you prove *in the very next line* that we *can* write code like this. – Jörg W Mittag Jan 13, 2023 at 11:02

9 Answers

Sorted by: Highest score (default)

75

This does happen in the real world! For example, a famous sorting algorithm is Timsort:

Timsort

> Details of the below implementation:
>
> We consider the size of the run as 32 and the input array is divided into sub-array.
>
> We one-by-one sort pieces of size equal to run with a simple insertion sort. After sorting individual pieces, we merge them one by one with the merge sort.

> We double the size of merged subarrays after every iteration.

Insertion sort has complexity O(N^2) but is faster for tiny lists, Merge Sort has complexity O(N logN) so it is better for longer lists.

Introsort

Another example is introsort, the sort used in the C++ standard library:

> Introsort or introspective sort is a hybrid sorting algorithm that provides both fast average performance and (asymptotically) optimal worst-case performance. It begins with quicksort, it switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted and it switches to insertion sort when the number of elements is below some threshold. This combines the good parts of the three algorithms, with practical performance comparable to quicksort on typical data sets and worst-case O(n log n) runtime due to the heap sort. Since the three algorithms it uses are comparison sorts, it is also a comparison sort.

More complexity downside

The downside of using more algorithms for a single task is clearly increased complexity. It is worth it if you are writing standard library code for a programming language that will be re-used millions or even billions of times. For smaller projects focusing on saving developer time over machine time by implementing only one algorithm is often the better choice.

References:

- *TimSort* [sic]
- *Introsort*

Share Improve this answer

Follow

edited Jan 25, 2023 at 23:05

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 12, 2023 at 22:21

Caridorc

6,631 ● 3 ● 36 ● 47

The downside of using more algorithms for a single task is clearly increased complexity. – Caridorc Jan 12, 2023 at 22:28

3 Not algorithmic complexity though, if you did it right! – Mad Physicist Jan 14, 2023 at 3:16

2 And presumably, the more code involved, the more likely it is to suffer from bad cache performance? – gidds Jan 14, 2023 at 13:40

@gidds No. Where did you get that idea? – Passer By Jan 16, 2023 at 7:14

@PasserBy: Instruction cache, not data cache. Large code taking up too much instruction cache can cause performance problems, which is one of the reasons we don't do things like aggressively unroll every loop in a program. – user2357112 Feb 24, 2023 at 22:38

34

Yes, this is common. For example, bignums multiplication can be done in several ways:

- naive `O(N^2)`
- Karatsuba `O((N^1.585)`
- Schönhage–Strassen `O(N*log(N)*(log(log(N))))`
- and there are also more advanced slightly faster algorithms now

So based on input variables used, the bitwidth fastest version is used (I use this for performance-critical "low-level" math operations on bignums all the time, because they are used as a building block for higher operations and without it it would not work with optimal speed on "whole/practical" number ranges).

However, the thresholds depends on computing hardware architecture, used compiler and sometimes even code usage, so they might differ on a per computer basis, so to ensure best performance, the thresholds are sometimes measured at program startup or configuration phase instead of using hardcoded values.

This is usually used on functions that has a huge variety of input sizes and also not on trivial functions, because the initial `if` statements that selects between algorithms is also performance hit (branch). In some "rare" cases I think you can do this branchlessly. For example, if the input size is also the input parameter of a template or maybe even a macro, for example, like in here (C++):

Gaussian elimination without result for acceleration

```
double matrix<1>::det() { return a[0][0]; }
double matrix<2>::det() { return (a[0][0]*a[1][1])-(a[
template <DWORD N> double matrix<N>::det()
        {
        double d=0.0; int j;
        matrix<N-1> m;
        for (j=0;j<N;j++)
            {
            m=submatrix(0,j);
            if (int(j&1)==0) d+=a[0][j]*m.det();
             else            d-=a[0][j]*m.det();
            }
        return d;
        }
```

As you can see, there are three different methods for the determinant based on input size, but no branches for the selection. However, only hardcoded thresholds can be used.

You can also achieve this with function pointers, for example, (C++):

```
int algoritm1(int x){ return 10; }
int algoritm2(int x){ return 20; }
int algoritm3(int x){ return 30; }
int (*functions[4])(int x)={ algoritm1,algoritm2,algor
int function(int x)
   {
   return functions[(x>>10)&3](x);
   }
```

So for `x` up to 1024 using algoritm1, up to 2048 using algorithm2 and for the rest, algorithm3 without any branches too. Here you can have dynamic thresholds (not that flexible, but still usable), so you can sample your x range by some power of 2 (like I did 2^10=1024) and just use duplicates. So, for example, if rounded thresholds are `3*1024` and `5*1024`, you can do this:

```
int algoritm1(int x){ return 10; }
int algoritm2(int x){ return 20; }
int algoritm3(int x){ return 30; }
int (*functions[8])(int x)=
   {
   algoritm1,
   algoritm1,
   algoritm1,
   algoritm2,
   algoritm2,
   algoritm3,
   algoritm3,
   algoritm3
   };
int function(int x)
   {
   return functions[(x>>10)&7](x);
   }
```

So you can create the function pointers array based on measured threshold at runtime too with this approach...

Share Improve this answer

Follow

edited Feb 24, 2023 at 22:23

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 13, 2023 at 7:25

Spektre

51.8k ● 12 ● 118 ● 395

Thank you for your contribution, can you please add a source for your claim (that different algorithms are implemented depending on the size of input in well-known bignum manipulation libraries)? – Caridorc Jan 13, 2023 at 13:46

1 @Caridorc you can measure it easily as for example Schönhage-Strassen is very slow for inputs below ~100000 bits in comparison to Karatsuba. so its enough to measure something like `1e10 * 1e10` and `1e100000 * 1e100000` and check if the complexity is the same or not. You can use any big integer lib or computing SW. and measure the complexity as function of input size see: measuring time complexity once you see that the complexity change after some input size then its clearly the case. – Spektre Jan 13, 2023 at 15:27

6 You can also disect any decent BigInteger lib source code to confirm this like GMP for example mine own libs does it all the time for cruicial functions which are used as building block for higher operations – Spektre Jan 13, 2023 at 15:28

"*You can achieve this also with function pointers*" - I wouldn't call that approach branchless though – Bergi Jan 14, 2023 at 22:42

3 Yes. A computed jump is branching just like a conditional jump. – Bergi Jan 15, 2023 at 2:27

19

As the other answer have said: yes!

Another example is Java's HashMap class. This class resolves collisions using separate chaining. Initially, each bucket contains a linked list, but if the length of this list growth past some threshold (8 in Java 8), it is converted into a TreeNode (a TreeNode is implemented as a red-black tree). Finding an item in a bucket through a linked list has a O(n) time complexity, while the TreeNode have a O(log n) time complexity.

Interestingly, the use of the linked list instead of the TreeNode is not (mainly) to save time, but rather to save space. A comment in the source code says:

> Because TreeNodes are about twice the size of regular nodes, we use them only when bins contain enough nodes to warrant use

Share Improve this answer Follow

edited Jan 14, 2023 at 7:16

answered Jan 13, 2023 at 11:10

Dada

6,626 ● 7 ● 26 ● 46

17

Yes, using multiple algorithms can even increase speed even for large n

Other answers mention many advanced algorithms that combine many primitive algorithms to create a more efficient advanced algorithm, including:

- Combining insertion sort and merge sort (in the case of Timsort)
- Combining naive and Strassen's algorithms for matrix multiplication
- Combining naive and Schönhage-Strassen algorithms for multiplying big numbers

Note that the better complexity, the worse runtime algorithms used here are recursive. That means they will call themselves on smaller bits of the data. That means that even if the size of the data structure is enough to make the better complexity algorithm faster, it will *eventually* reduce to one or more problems of a small size, even if `n` is initially big.

This means that even for large `n` where the recursive algorithm is initially used, a large performance benefit can be gained by switching to a faster algorithm once the problem size has been reduced enough to make it viable.

Share Improve this answer

Follow

edited Jan 25, 2023 at 23:09

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 13, 2023 at 13:00

mousetail 'he-him'
7,991 ● 5 ● 29 ● 52

4 Matrix multiplication is such a good example of this that it would warrant its own answer. Not only is a mix of naive and Strassen's algorithms actually used in practice with a clause `if side < 100: naive; else: Strassen's`; but there exist several different algorithms with better asymptotic complexity than Strassen's, and none of them is used in practice. Naive is O(n^3), Strassen's is O(n^2.8), best-known is O(n^2.37) (where n is the side of the matrices, assuming the two matrices are square). – Stef Jan 15, 2023 at 20:41

2 @Stef The algorithms which are better than Strassen are generally galactic algorithms : they have a so huge (hidden) constant in the complexity that they are basically useless even for pretty huge matrices on current computers. As for Strassen, AFAIK, most BLAS libraries do not use it. They just use the naive `O(n**3)` algorithm, even for big matrices. This is because Strassen is only interesting for huge matrices (due to being less cache-friendly) and it is AFAIK *less numerically stable*. – Jérôme Richard Jan 22, 2023 at 15:17

8

The other answers here have given plenty of examples, but I wanted to expand on them with one major reason high time complexity algorithms are sometimes used.

When talking about the space/time complexity of an algorithm, it's generally implied that we mean the worst-case, or perhaps average-case of the algorithm. However, real world datasets and problems often have

exploitable patterns in them, where they might be closer to the best-case for an algorithm than its worst-case.

Consider a couple of specific examples:

- Timsort is a hybrid sort that primarily uses merge sort which is relatively performant with a time complexity of *O(n log n)*, but it also intentionally utilizes several much worse performance algorithms. The reason it is designed this way is that it acts like a state machine, observing whether any of the data being sorted is already somewhat ordered, and if so picks an appropriate algorithm that performs well in the presence of that kind of order. (See also adaptive sorting.)
- Compression algorithms generally produce *larger* compressed sizes in their worst cases, but the data people actually want to compress isn't random; video, audio, text all have patterns that can be exploited that mean compression yields large space savings.

Share Improve this answer

Follow

edited Feb 25, 2023 at 9:40

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 13, 2023 at 21:48

Blackhawk

6,100 ● 4 ● 29 ● 58

4 And sometimes high-complexity algorithms are used because they've got better constant factors. For example, there's an

O(n log n) algorithm for multiplying numbers, but the constant factors on that are so bad that it's impractical for any number small enough to be stored in a modern computer. – Mark Jan 13, 2023 at 22:19

1 @Blackhawk Your sentence *"combining these algorithms allows Timsort to perform much closer to its best-case than its worst-case for a lot of real world data."* still makes it sound like Timsort could reach a worst-case of n^2 on some "unlucky" data, but that's okay because it's O(n log(n)) on most "real-world" data. I don't think that"s true. I think Timsort is guaranteed to be O(n log(n)). – Stef Jan 16, 2023 at 8:51

1 @Stef The question is, whether Insertion Sort is only used in Timsort in a way (only in cases), where it is running in `O(n * log n)`. The check for exploitable structures can also run in `O(n * log n)`. – Sebastian Jan 16, 2023 at 16:49

1 @KellyBundy ...From a computer science perspective, algorithms are measured based on their performance across all possible inputs, whereas practically speaking humans only care about a very small subset of all inputs (audio, video, text, structured records, partially sorted lists, etc.), and algorithms that perform well in the average case on specific subsets are often more useful in practice than generalized algorithms that minimize the worst case. – Blackhawk Feb 9, 2023 at 16:16

1 @KellyBundy to quote my earlier comment, "Now for the more interesting cases. lg(n!) is the information-theoretic limit for the best any comparison-based sorting algorithm can do on average (across all permutations). When a method gets significantly below that, it's either astronomically lucky, or is finding exploitable structure in the data." - Tim Peters
– Blackhawk Feb 9, 2023 at 16:20

6

In formal verification, we solve NP-complete, exponential and undecidable problems all the time (all of these using algorithms much worse than $O(n^2)$ (provided a potentially neverending search can be considered an algorithm)).

The compiler for the Ethereum smart contract language, Solidity, interfaces with the SMT solver Z3 in order to try to formally prove the `asserts` in the code will always hold, and that problem is undecidable.

The computer algebra algorithm F5 is exponential and was used to break HFE cryptosystems.

There are many more examples. It is less about the size of the input, and more about luck and the input having the kind of structure where the algorithm will not hit its worst case complexity (E.g., bubble sort is $O(n^2)$ in the worst case, but if the array is almost sorted in a certain way, it can run in linear time).

Share Improve this answer

Follow

edited Feb 25, 2023 at 9:37

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 13, 2023 at 18:57

Ivella

13.4k ● 12 ● 60 ● 119

2 Good examples, except for Bubble Sort; one element far out of place can make it take nearly the full N^2, or just N depending on which direction it's moving. A better example is Insertion Sort, which is always pretty close to linear for

almost-sorted arrays. en.wikipedia.org/wiki/Bubble_sort#Rabbits_and_turtles / Why bubble sort is not efficient? / Bubble Sort: An Archaeological Algorithmic Analysis – Peter Cordes Jan 14, 2023 at 2:07

A problem is NP-complete. An algorithm is not NP-complete. If a particular algorithm's worst-case runtime is much worse than n^2, then it's going to remain much worse than n^2 regardless of whether P and NP are equal or not. – Stef Jan 15, 2023 at 20:34

@Stef I think Ivella meant algorithms for NP complete problems?!? – Sebastian Jan 15, 2023 at 22:07

@Stef What I meant is if you can find a $O(n^2)$ algorithm to a NP-complete problem, then P = NP, not the converse. – Ivella Jan 16, 2023 at 11:11

@Ivella May I suggest rewriting the first sentence of the answer, *"In formal verification we use NP-complete, exponential and undecidable algorithms all the time (all of these much worse than $O(n^2)$ (provided P != NP))."*, so that it's less wrong? – Stef Jan 16, 2023 at 11:16

5

In most of NP-type problems, you have to use approximation algorithms or brute-force/inefficient algorithms which have high time complexities.

There are lots of divide and conquer algorithms that depend on input and the complexity may vary based on type of the input that are given like in sorting, searching, arithmetics, etc.

Most algorithms take inputs and inputs do vary indeed, so basically every real-world algorithm that are being used

are made of smaller algorithms that do well on specific type of inputs. And there are going to be research and evolvements on those specific algorithms that improves the complexity those algorithms, but you also have to realize that time complexity is not always a good way of measuring the speed of your algorithm since it's just putting a limit and a relation on the growth of it when going to infinity (not counting smaller constants or the way those O(1) instruction are made).

Older implementations of quicksort use insertion sort, because it's efficient for small data sets which is better than most other simple quadratic algorithms, such as selection sort or bubble sort.

There is another use cases which are hard to compute by design choices such as cryptocurrency mining algorithms and CPU cycle measuring...

Share Improve this answer

Follow

edited Feb 24, 2023 at 22:23

Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jan 13, 2023 at 1:15

user20962667

4

This answer doesn't fully answer the question in that it is a ways away from software development. However, it describes cases beyond almost trivial sorting algorithm optimizations already described in other answers.

In competitive programming, there are problems involving combining two algorithms that go beyond constant factor optimizations, and result in a better time complexity, usually from O(N^2) to O(N^1.5).

This material (Chapter 27, section 2) describes one such simplified case very well in which one has a grid with a number of colors on it and the question being for each color how far is the minimum distance between two cells of the same color. A naive approach involves running a BFS for each color with a time complexity of O(number of colors * N) which reduces to O(N * N) (in the case that the grid contains colors only once).

Another involves for each pair of cells of a color finding their distance and taking the minimum with a time complexity of the sum of (k * k) across all colors which can be shown to reduce to O(N * N) in the worst case (the entire grid is the same color). A clever method of selecting only some colors to run the first algorithm on and using the second on others is described in the text to reach a complexity of O(N*sqrt(N)).

For those who want to challenge themselves, here is a problem that uses a similar idea to the one described in the material with a more complex data structure. The solution can be found here.

edited Feb 12, 2023 at 14:01

answered Jan 15, 2023 at 2:42

Username_taken12

41 • 4

What's "Chapter 21, section 2"? Chapter 21 is number theory and I don't see numbered sections. – Kelly Bundy Feb 9, 2023 at 8:52

Sorry, it should be chapter 27.2; edited to reflect that. – Username_taken12 Feb 12, 2023 at 14:02

2

One reason for ***not*** writing code like **if (n < 1000) alg_1 else alg_2** is that you need to develop and test two algorithms and also need to check that they perform exactly the same under all circumstances.

Since the question states that algorithm time is a critical factor, testing might be a very time-consuming matter. It might be a good choice to just take the algorithm that gives you the best overall performance. It is also a trade-off between developer efficiency and algorithm efficiency.

Share Improve this answer

Follow

edited Jan 25, 2023 at 23:16

Peter Mortensen

31.6k • 22 • 109 • 133

answered Jan 17, 2023 at 21:51

OldFrank

878 • 6 • 7

1 The need for testing to ensure *correctness* has nothing to do with algorithmic complexity at *runtime*, though. I, as a user, do not have to confirm that insertion sort and merge sort are

both correct every time I use Timsort in a program. – chepner Jan 17, 2023 at 22:03

I provided a general answer to the general problem in the question. Many other answers including yours are algorithm-specific. In some specific cases where one algorithm is well-proven you might be able to skip a few steps. With more complexity and more time needed for testing, the more challenging correctness becomes. – OldFrank Jan 18, 2023 at 11:51