Programmatically obtaining Big-O efficiency of code

Asked 15 years, 11 months ago Modified 3 years, 1 month ago

Viewed 21k times

45

I wonder whether there is any automatic way of determining (at least roughly) the Big-O time complexity of a given function?

If I graphed an O(n) function vs. an O(n lg n) function I think I would be able to visually ascertain which is which; I'm thinking there must be some heuristic solution which enables this to be done automatically.

Any ideas?

Edit: I am happy to find a semi-automated solution, just wondering whether there is some way of avoiding doing a fully manual analysis.

algorithm complexity-theory

Share

Improve this question

Follow

edited Aug 29, 2009 at 10:17

Helen

96.9k ● 17 ● 272 ● 338

asked Jan 26, 2009 at 18:09

ljs
37.8k ● 36 ● 109 ● 124

2 You could try and get someone to do it on rentacoder, just like they did it with the halting problem :) – Tamas Czinege Jan 26, 2009 at 18:27

huh? Lol - wow those rentacoders are just tops, aren't they? =P – Erik Forbes Jan 26, 2009 at 18:52

If all you need is just a polynomial fitting of some sample data then here's an example stackoverflow.com/questions/464960/… (see link to the source code at the end of the post). – jfs Jan 26, 2009 at 21:42

18 Answers

Sorted by: Highest score (default)

62

✓

It sounds like what you are asking for is an extention of the Halting Problem. I do not believe that such a thing is possible, even in theory.

Just answering the question "Will this line of code ever run?" would be very difficult if not impossible to do in the general case.

Edited to add: Although the general case is intractable, see here for a partial solution: http://research.microsoft.com/apps/pubs/default.aspx?id=104919

Also, some have stated that doing the analysis by hand is the only option, but I don't believe that is really the correct way of looking at it. An intractable problem is still intractable even when a human being is added to the system/machine. Upon further reflection, I suppose that a 99% solution may be doable, and might even work as well as or better than a human.

Share Improve this answer Follow

edited Oct 23, 2009 at 5:58

answered Jan 26, 2009 at 18:15

Jeffrey L Whitledge
59.4k ● 9 ● 74 ● 100

1 Yes, it is an important thing to point out. Nonetheless there are still semi-automated solutions, as the OP has stated he is content with. – Noldorin Jan 26, 2009 at 18:28

Where do I find semi-automated solutions? I'd like to see one, as I have not been able to visualize one. – David Thornley Jan 26, 2009 at 18:41

The Halting Problem is only a problem if: 1) Your function can enter a state from which it will never finish. (Hopefully your production code doesn't fall into this category). - AND - 2) Your analyzer is unable to determine, by looking at the code, that #1 applies to your function. – mbeckish Jan 26, 2009 at 19:13

1 I believe that is the very definition of the problem. =P – Erik Forbes Jan 26, 2009 at 19:22

5 It's true that the halting problem applies in that it's not always possible to do determine the big-O (because you might not be able to prove that the function ever finishes). Nevertheless, we do manual big-O analyses all the time. The halting problem doesn't say we can never do it. It says that

there are programs for which it cannot be done. That's an important distinction. If the questioner is after a way to compare several algorithms that do halt, then it is possible to estimate big-O using empirical data rather than manual analysis. – Adrian McCarthy Dec 9, 2009 at 0:21

39

You can run the algorithm over various size data sets, and you could then use curve fitting to come up with an approximation. (Just looking at the curve you create probably will be enough in most cases, but any statistical package has curve fitting).

Note that some algorithms exhibit one shape with small data sets, but another with large... and the definition of large remains a bit nebulous. This means that an algorithm with a good performance curve could have so much real world overhead that (for small data sets) it doesn't work as well as the theoretically better algorithm.

As far as *code inspection* techniques, none exist. But instrumenting your code to run at various lengths and outputting a simple file (RunSize RunLength would be enough) should be easy. Generating proper test data could be more complex (some algorithms work better/worse with partially ordered data, so you would want to generate data that *represented your normal use-case*).

Because of the problems with the definition of "what is large" and the fact that performance is data dependent, I find that static analysis often is misleading. When optimizing performance and selecting between two algorithms, the real world "rubber hits the road" test is the only final arbitrator I trust.

Share Improve this answer Follow

edited Sep 28, 2014 at 17:30

Matt Ball
360k ● 102 ● 652 ● 719

answered Jan 26, 2009 at 18:15

Godeke
16.3k ● 4 ● 64 ● 87

I didn't down-vote, but the curve-fitting approach gives (at best) a guess at the big-O for an algorithm. There is a good chance that it will be wrong ... especially when you scale up to a problem size that is orders of magnitude bigger than your sample. – Stephen C Dec 9, 2009 at 0:15

"Note that some algorithms exhibit one shape with small data sets, but another with large... and the definition of large remains a bit nebulous." A properly derived big-O formula tells you how the algorithm behaves as N gets really large. You can (if you are prepared to do the math) also characterize an algorithm in other ways; e.g. for small N, on average/best case/worst case, for multiple parameters. – Stephen C Dec 9, 2009 at 0:20

The poster asked for an "automatic way" to make such determinations. Unless you have solved the halting problem (please share!) there is no automated method. I suggested curve fitting as the closest to automated, especially as you can graph your various algorithmic choices over the problem sizes you expect to handle and expand to handle. I made this suggestion because the original poster was not interested in the analytical approach but an automated one. – Godeke Dec 9, 2009 at 15:08

---

2 As an aside, I have often found that the algorithmically superior solution is actually slower in practice because of that pesky constant that reflects implementation efficiency. The best sorting algorithms, for example, are hybrids that use a simple approach until they reach a specific depth of recursion at which point they use the more "expensive" but "algorithmically better" solution. – Godeke Dec 9, 2009 at 15:13

15

A short answer is that **it's impossible because constants matter**.

For instance, I might write a function that runs in `O((n^3/k) + n^2)`. This simplifies to O(n^3) because as n approaches infinity, the `n^3` term will dominate the function, irrespective of the constant `k`.

However, if `k` is very large in the above example function, the function will appear to run in almost exactly `n^2` until some crossover point, at which the `n^3` term will begin to dominate. Because the constant `k` will be unknown to any profiling tool, it will be impossible to know just how large a dataset to test the target function with. If `k` can be arbitrarily large, you cannot craft test data to determine the big-oh running time.

Share Improve this answer Follow

answered Jan 26, 2009 at 18:56

Kenan Banks
212k ● 36 ● 159 ● 175

Excellent point, though if the number of steps is known it's possible to use curve fitting to get an approximate answer.

– Joey Robert Jul 14, 2009 at 16:10

“It’s impossible”. This answer needs rephrasing. The issue it points to does *not* make solving the question impossible. It only makes *some approach* for solving it (sampling with a stopwatch) non-applicable. – Maëlan Mar 29, 2021 at 19:54

13

I am surprised to see so many attempts to claim that one can "measure" complexity by a stopwatch. Several people have given the right answer, but I think that there is still room to drive the essential point home.

1. Algorithm complexity is not a "programming" question; it is a "computer science" question. Answering the question requires analyzing the code from the perspective of a mathematician, such that computing the Big-O complexity is practically a form of mathematical proof. It requires a very strong understanding of the fundamental computer operations, algebra, perhaps calculus (limits), and logic. No amount of "testing" can be substituted for that process.

2. The Halting Problem applies, so the complexity of an algorithm is fundamentally undecidable by a machine.

3. The limits of automated tools applies, so it might be possible to write a program to help, but it would only be able to help about as much as a calculator helps with one's physics homework, or as much as a

refactoring browser helps with reorganizing a code base.

4. For anyone seriously considering writing such a tool, I suggest the following exercise. Pick a reasonably simple algorithm, such as your favorite sort, as your subject algorithm. Get a solid reference (book, web-based tutorial) to lead you through the process of calculating the algorithm complexity and ultimately the "Big-O". Document your steps and results as you go through the process with your subject algorithm. Perform the steps and document your progress for several scenarios, such as best-case, worst-case, and average-case. Once you are done, review your documentation and ask yourself what it would take to write a program (tool) to do it for you. Can it be done? How much would actually be automated, and how much would still be manual?

Best wishes.

Share Improve this answer Follow

answered Jan 26, 2009 at 20:59

Rob Williams
7,921 ● 1 ● 37 ● 42

2 Good answer. But even if a problem is not decidable, there is no stopping us trying to solve it for special cases, especially of practical interest. – user51568 Jan 29, 2009 at 12:14

Yup, programming and programmers thrive on special cases, where even some of the hardest problems become tractable. – Rob Williams Jan 29, 2009 at 19:00

9

I am curious as to why it is that you want to be able to do this. In my experience when someone says: "I want to ascertain the runtime complexity of this algorithm" they are not asking what they think they are asking. What you are most likely asking is what is the realistic performance of such an algorithm for likely data. Calculating the Big-O of a function is of reasonable utility, but there are so many aspects that can change the "real runtime performance" of an algorithm in real use that nothing beats instrumentation and testing.

For example, the following algorithms have the same exact Big-O (wacky pseudocode):

example a:

```
huge_two_dimensional_array foo
for i = 0, i < foo[i].length, i++
  for j = 0; j < foo[j].length, j++
    do_something_with foo[i][j]
```

example b:

```
huge_two_dimensional_array foo
for j = 0, j < foo[j].length, j++
  for i = 0; i < foo[i].length, i++
    do_something_with foo[i][j]
```

Again, exactly the same big-O... but one of them uses row ordinality and one of them uses column ordinality. It turns out that due to locality of reference and cache coherency you might have two *completely* different actual runtimes, especially depending on the actual size of the array foo. This doesn't even begin to touch the actual performance characteristics of how the algorithm behaves if it's part of a piece of software that has some concurrency built in.

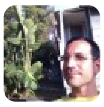
Not to be a negative nelly but big-O is a tool with a narrow scope. It is of great use if you are deep inside algorithmic analysis or if you are trying to *prove* something about an algorithm, but if you are doing commercial software development the proof is in the pudding, and you are going to want to have actual performance numbers to make intelligent decisions.

Cheers!

Share Improve this answer Follow

edited Jan 26, 2009 at 18:49

answered Jan 26, 2009 at 18:43



earino
2,925 ● 21 ● 20

But as your n doubles, the time that you spend thrashing the cache (basically) doubles. Your complexity doesn't change, but the constant does. This doesn't say that Big-O is useless. – Calyth Jan 26, 2009 at 19:10

1 @Calyth. No, it doesn't say big-o is useless. That's why earino **didn't say** big-o is useless, instead that it is of reasonable utility. – MarkJ Sep 22, 2009 at 16:49

4

This could work for simple algorithms, but what about O(n^2 lg n), or O(n lg^2 n)?

You could get fooled visually very easily.

And if its a really bad algorithm, maybe it wouldn't return even on n=10.

Share Improve this answer Follow

answered Jan 26, 2009 at 18:14

Pyrolistical
28k ● 21 ● 84 ● 109

An algorithm can be classified as bad by being a combination of incorrect, incomprehensible, or inefficient. If the latter, n=10 is a linear-timed algorithm (i.e, O(k), very desirable), meaning the code queried would have to contain no loops ... – rlb.usa Jan 26, 2009 at 19:53

4

Proof that this is undecidable:

Suppose that we had some algorithm HALTS_IN_FN(Program, function) which determined whether a program halted in O(f(n)) for all n, for some function f.

Let P be the following program:

```
if(HALTS_IN_FN(P,f(n)))
{
    while(1);
}
halt;
```

Since the function and the program are fixed, HALTS_IN_FN on this input is constant time. If HALTS_IN_FN returns true, the program runs forever and of course does not halt in O(f(n)) for any f(n). If HALTS_IN_FN returns false, the program halts in O(1) time.

Thus, we have a paradox, a contradiction, and so the program is undecidable.

Share Improve this answer Follow

edited Apr 29, 2015 at 20:33
Alex Riley
176k ● 46 ● 272 ● 245

answered Apr 15, 2009 at 21:09
mindvirus
5,236 ● 4 ● 31 ● 47

4

I'm using a `big_O` library (link here) that fits the change in execution time against independent variable `n` to infer the order of growth class `O()`.

The package automatically suggests the best fitting class by measuring the residual from collected data against each class growth behavior.

Check the code in this answer.

Example of output,

```
Measuring .columns[::-1] complexity against rapid
increase in # rows
--------------------------------------------------
------------------------------
Big O() fits: Cubic: time = -0.017 + 0.00067*n^3
--------------------------------------------------
------------------------------
Constant: time = 0.032
(res: 0.021)
Linear: time = -0.051 + 0.024*n
(res: 0.011)
Quadratic: time = -0.026 + 0.0038*n^2
(res: 0.0077)
Cubic: time = -0.017 + 0.00067*n^3
(res: 0.0052)
Polynomial: time = -6.3 * x^1.5
(res: 6)
Logarithmic: time = -0.026 + 0.053*log(n)
(res: 0.015)
Linearithmic: time = -0.024 + 0.012*n*log(n)
(res: 0.0094)
Exponential: time = -7 * 0.66^n
(res: 3.6)
--------------------------------------------------
------------------------------
```

Share Improve this answer

Follow

answered Aug 4, 2018 at 19:06

helcode

2,048 ● 1 ● 16 ● 32

3

A lot of people have commented that this is an inherently unsolvable problem in theory. Fair enough, but beyond that, even solving it for any but the most trivial cases would seem to be incredibly difficult.

Say you have a program that has a set of nested loops, each based on the number of items in an array. O(n^2).

But what if the inner loop is only run in a very specific set of circumstances? Say, on average, it's run in aprox log(n) cases. Suddenly our "obviously" O(n^2) algorithm is really O(n log n). Writing a program that could determine if the inner loop would be run, and how often, is potentially more difficult than the original problem.

Remember O(N) isn't god; high constants can and will change the playing field. Quicksort algorithms are O(n log n) of course, but when the recursion gets small enough, say down to 20 items or so, many implementations of quicksort will change tactics to a separate algorithm as it's actually quicker to do a different type of sort, say insertion sort with worse O(N), but much smaller constant.

So, understand your data, make educated guesses, and test.

Share Improve this answer Follow

answered Jan 26, 2009 at 18:45

Beska
12.7k ● 14 ● 79 ● 113

2

I think it's pretty much impossible to do this automatically. Remember that O(g(n)) is the worst-case upper bound and many functions perform better than that for a lot of data sets. You'd have to find the worst-case data set for each one in order to compare them. That's a difficult task on its own for many algorithms.

Share Improve this answer Follow

answered Jan 26, 2009 at 18:21

Kamil Kisiel
20.4k ● 12 ● 49 ● 57

2

You must also take care when running such benchmarks. Some algorithms will have a behavior heavily dependent on the input type.

Take Quicksort for example. It is a worst-case O(n²), but usually O(nlogn). For two inputs of the same size.

The traveling salesman is (I think, not sure) O(n²) (*EDIT: the correct value is 0(n!) for the brute force algotithm*) , but most algorithms get rather good approximated solutions much faster.

This means that the the benchmarking structure has to most of the time be adapted on an ad hoc basis. Imagine writing something generic for the two examples mentioned. It would be very complex, probably unusable, and likely will be giving incorrect results anyway.

Share Improve this answer Follow

edited Jan 26, 2009 at 19:21

answered Jan 26, 2009 at 18:19

user54579
4,688 ● 4 ● 25 ● 19

The TSP is NP-Complete, just FYI. – Andrew Coleson Jan 26, 2009 at 18:25

A nieve traveling salesman algorithm is O(n!), which is hyper-exponential, but I think algorithms are possible that are only exponential O(2^n). There is no known polynomial O(n^2) solution. – Jeffrey L Whitledge Jan 26, 2009 at 18:28

Actually, for what it's worth, O(n!) is actually exponential O(n!) ~=~ O((n/e)^n) < n^n = 2^(nlogn) < 2^(n^2), witch is clearly exponential, not hyper-exponential. As a matter of fact, any NP problem has an EXPTIME algorithm. – user51568 Jan 29, 2009 at 12:07

There is an O(n^3*2^n) dynamic programming algorithm, where you fill in a matrix M[1 <= i <= n][1 <= j <= n][c_1 = f/t, ..., c_n = f/t] = true iff there is a road from i to j using exactly the nodes k for which c_k = true. – user51568 Jan 29, 2009 at 12:12

1

Jeffrey L Whitledge is correct. A simple reduction from the halting problem proves that this is undecidable...

ALSO, if I could write this program, I'd use it to solve P vs NP, and have $1million... B-)

Share Improve this answer Follow

answered Jan 26, 2009 at 18:33

Brian Postow
12.1k ● 21 ● 85 ● 133

0

I guess this isn't possible in a fully automatic way since the type and structure of the input differs a lot between functions.

Share Improve this answer

answered Jan 26, 2009 at 18:14

Follow

mooware
1,762 ● 2 ● 18 ● 25

0

Well, since you can't prove whether or not a function even halts, I think you're asking a little much.

Otherwise @Godeke has it.

Share Improve this answer

Follow

answered Jan 26, 2009 at 18:22

geowa4
41.7k ● 17 ● 89 ● 107

0

I don't know what's your objective in doing this, but we had a similar problem in a course I was teaching. The students were required to implement something that works at a certain complexity.

In order not to go over their solution manually, and read their code, we used the method @Godeke suggested. The objective was to find students who used linked list instead of a balansed search tree, or students who implemented bubble sort instead of heap sort (i.e. implementations that do not work in the required complexity - but without actually reading their code).

Surprisingly, the results did not reveal students who cheated. That might be because our students are honest and want to learn (or just knew that we'll check this ;-)). It is possible to miss cheating students if the inputs are small, or if the input itself is ordered or such. It is also possible to be wrong about students who did not cheat, but have large constant values.

But in spite of the possible errors, it is well worth it, since it saves a lot of checking time.

Share Improve this answer Follow

answered Aug 29, 2009 at 10:39

Anna

4,139 ● 4 ● 26 ● 26

0

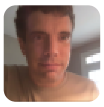
As others have said, this is theoretically impossible. But in practice, you *can* make an educated guess as to whether a function is O(*n*) or O(*n*^2), as long as you don't mind being wrong sometimes.

First time the algorithm, running it on input of various *n*. Plot the points on a [log-log graph](). Draw the best-fit line through the points. If the line fits all the points well, then the data suggests that the algorithm is O(*n*^*k*), where *k* is the slope of the line.

I am not a statistician. You should take all this with a grain of salt. But I have actually done this in the context of automated testing for performance regressions. [The patch here]() contains some JS code for it.

Share Improve this answer Follow

answered Dec 8, 2009 at 23:56



Jason Orendorff

44.4k ● 6 ● 65 ● 100

-1

If you have lots of homogenious computational resources, I'd time them against several samples and do linear regression, then simply take the highest term.

Share Improve this answer Follow

answered Jan 26, 2009 at 18:25

Dmitri Nesteruk
23.8k ● 23 ● 106 ● 169

-1

It's easy to get an indication (e.g. "is the function linear? sub-linear? polynomial? exponential")

It's hard to find the exact complexity.

For example, here's a Python solution: you supply the function, and a function that creates parameters of size N for it. You get back a list of (n,time) values to plot, or to perform regression analysis. It times it once for speed, to get a really good indication it would have to time it many times to minimize interference from environmental factors (e.g. with the timeit module).

```
import time
def measure_run_time(func, args):
  start = time.time()
  func(*args)
  return time.time() - start

def plot_times(func, generate_args,
plot_sequence):
  return [
    (n, measure_run_time(func,
generate_args(n+1)))
```

```
    for n in plot_sequence
  ]
```

And to use it to time bubble sort:

```
def bubble_sort(l):
  for i in xrange(len(l)-1):
    for j in xrange(len(l)-1-i):
      if l[i+1] < l[i]:
        l[i],l[i+1] = l[i+1],l[i]

import random
def gen_args_for_sort(list_length):
  result = range(list_length) # list of 0..N-1
  random.shuffle(result) # randomize order
  # should return a tuple of arguments
  return (result,)

# timing for N = 1000, 2000, ..., 5000
times = plot_times(bubble_sort, gen_args_for_sort,
xrange(1000,6000,1000))

import pprint
pprint.pprint(times)
```

This printed on my machine:

```
[(1000, 0.078000068664550781),
 (2000, 0.34400010108947754),
 (3000, 0.76499986648559570),
 (4000, 1.3440001010894775),
 (5000, 2.1410000324249268)]
```

Share Improve this answer Follow

answered Jan 26, 2009 at 18:54

orip
75.3k ● 21 ● 119 ● 149

this doesn't work in the general case. See my answer for why. (I didn't downvote) – Kenan Banks Jan 26, 2009 at 18:58

What doesn't work, plotting the actual run time and trying to see a rough pattern with your eyes? :) Notice that I didn't claim regression analysis to be easy or even any good, but looking at a plot is very good at getting a gut feeling. – orip Jan 26, 2009 at 19:00

1 looking at a plot doesn't work, because you don't know, and can't know, how much of the plot to look at. – Kenan Banks Jan 26, 2009 at 19:40