What is a plain English explanation of "Big O" notation?

Asked 15 years, 11 months ago   Modified 2 months ago

Viewed 842k times

▲

5389

I'd prefer as little formal definition as possible and simple mathematics.

algorithm   complexity-theory   computer-science   big-o

time-complexity

▼

🔖

🕘   Share

Improve this question

Follow

edited Jul 22, 2016 at 15:40

Steve Chambers
39.3k 🟡 29 ⚪ 172 🟤 220

asked Jan 28, 2009 at 11:10

Arec Barrwin
62k 🟡 9 ⚪ 30 🟤 25

---

69   Summary: The upper bound of the complexity of an algorithm. See also the similar question [Big O, how do you calculate/approximate it?](#) for a good explaination.
– Kosi2801 Jan 28, 2009 at 11:18

---

7   The other answers are quite good, just one detail to understand it: O(log n) or similar means, that it depends on the "length" or "size" of the input, not on the value itself. This could be hard to understand, but is very important. For

example, this happens when your algorithm is splitting things in two in each iteration. – Harald Schilly Jan 28, 2009 at 11:23

18 There is a lecture dedicated to complexity of the algorithms in the Lecture 8 of the MIT "Introduction to Computer Science and Programming" course youtube.com/watch?v=ewd7Lf2dr5Q It is not completely plain English, but gives nice explanation with examples that are easily understandable. – ivanjovanovic Jul 17, 2010 at 20:57

19 Big O is an estimate of the worst case performance of a function assuming the algorithm will perform the maximum number of iterations. – Paul Sweatte Aug 28, 2012 at 0:16

39 Big-O notation explained by a self-taught programmer – Soner Gönül Sep 7, 2013 at 18:02

43 Answers

Sorted by: Highest score (default) ⇕

| 1 | 2 | Next |

▲

7122

▼

🔖

✓

+550

🕘

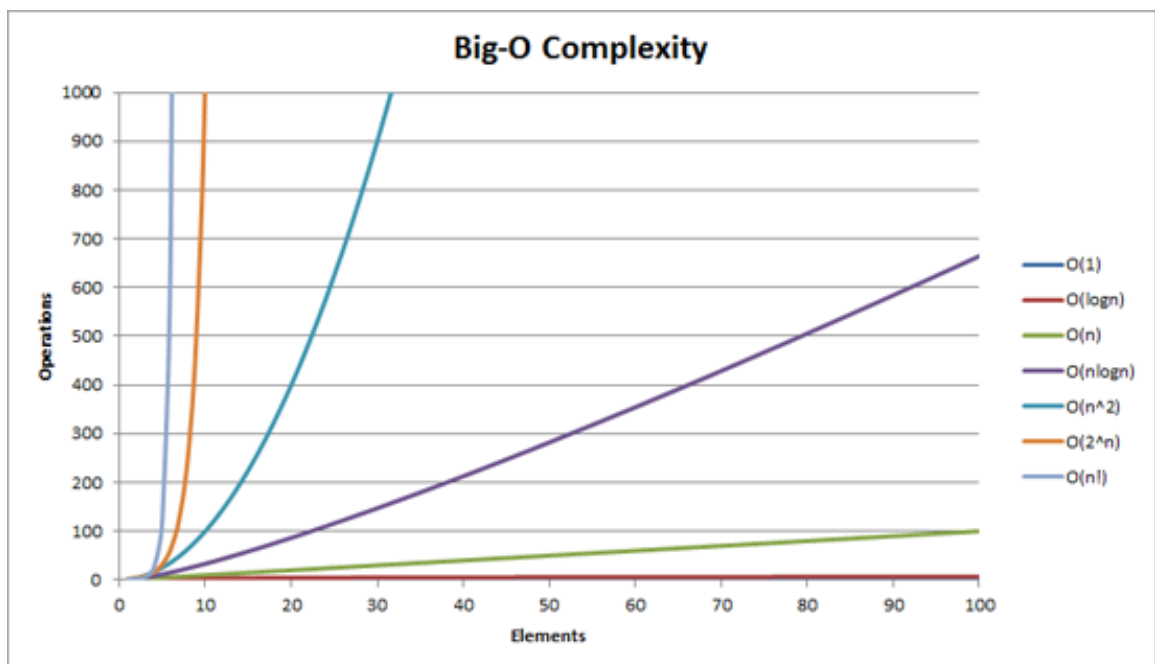Quick note, my answer is almost certainly confusing Big Oh notation (which is an upper bound) with Big Theta notation "Θ" (which is a two-side bound). But in my experience, this is actually typical of discussions in non-academic settings. Apologies for any confusion caused.

BigOh complexity can be visualized with this graph:

Big-O Complexity

The simplest definition I can give for Big Oh notation is this:

Big Oh notation is a relative representation of the complexity of an algorithm.

There are some important and deliberately chosen words in that sentence:

- **relative:** you can only compare apples to apples. You can't compare an algorithm that does arithmetic multiplication to an algorithm that sorts a list of integers. But a comparison of two algorithms to do arithmetic operations (one multiplication, one addition) will tell you something meaningful;

- **representation:** BigOh (in its simplest form) reduces the comparison between algorithms to a single variable. That variable is chosen

based on observations or assumptions. For example, sorting algorithms are typically compared based on comparison operations (comparing two nodes to determine their relative ordering). This assumes that comparison is expensive. But what if the comparison is cheap but swapping is expensive? It changes the comparison; and

- **complexity:** if it takes me one second to sort 10,000 elements, how long will it take me to sort one million? Complexity in this instance is a relative measure to something else.

Come back and reread the above when you've read the rest.

The best example of BigOh I can think of is doing arithmetic. Take two numbers (123456 and 789012). The basic arithmetic operations we learned in school were:

- addition;
- subtraction;
- multiplication; and
- division.

Each of these is an operation or a problem. A method of solving these is called an **algorithm**.

The addition is the simplest. You line the numbers up (to the right) and add the digits in a column writing the last number of that addition in the result. The 'tens' part of that number is carried over to the next column.

Let's assume that the addition of these numbers is the most expensive operation in this algorithm. It stands to reason that to add these two numbers together we have to add together 6 digits (and possibly carry a 7th). If we add two 100 digit numbers together we have to do 100 additions. If we add **two** 10,000 digit numbers we have to do 10,000 additions.

See the pattern? The **complexity** (being the number of operations) is directly proportional to the number of digits *n* in the larger number. We call this **O(n)** or **linear complexity**.

Subtraction is similar (except you may need to borrow instead of carry).

Multiplication is different. You line the numbers up, take the first digit in the bottom number and multiply it in turn against each digit in the top number and so on through each digit. So to multiply our two 6 digit numbers we must do 36 multiplications. We may need to do as many as 10 or 11 column adds to get the end result too.

If we have two 100-digit numbers we need to do 10,000 multiplications and 200 adds. For two one million digit numbers we need to do one trillion (10^{12}) multiplications and two million adds.

As the algorithm scales with n-*squared*, this is **O(n^2)** or **quadratic complexity**. This is a good time to introduce another important concept:

We only care about the most significant portion of complexity.

The astute may have realized that we could express the number of operations as: $n^2 + 2n$. But as you saw from our example with two numbers of a million digits apiece, the second term (2n) becomes insignificant (accounting for 0.0002% of the total operations by that stage).

One can notice that we've assumed the worst case scenario here. While multiplying 6 digit numbers, if one of them has 4 digits and the other one has 6 digits, then we only have 24 multiplications. Still, we calculate the worst case scenario for that 'n', i.e when both are 6 digit numbers. Hence Big Oh notation is about the Worst-case scenario of an algorithm.

The Telephone Book

The next best example I can think of is the telephone book, normally called the White Pages or similar but it varies from country to country. But I'm talking about the one that lists people by surname and then initials or first name, possibly address and then telephone numbers.

Now if you were instructing a computer to look up the phone number for "John Smith" in a telephone book that

contains 1,000,000 names, what would you do? Ignoring the fact that you could guess how far in the S's started (let's assume you can't), what would you do?

A typical implementation might be to open up to the middle, take the 500,000th and compare it to "Smith". If it happens to be "Smith, John", we just got really lucky. Far more likely is that "John Smith" will be before or after that name. If it's after we then divide the last half of the phone book in half and repeat. If it's before then we divide the first half of the phone book in half and repeat. And so on.

This is called a **binary search** and is used every day in programming whether you realize it or not.

So if you want to find a name in a phone book of a million names you can actually find any name by doing this at most 20 times. In comparing search algorithms we decide that this comparison is our 'n'.

- For a phone book of 3 names it takes 2 comparisons (at most).

- For 7 it takes at most 3.

- For 15 it takes 4.

- …

- For 1,000,000 it takes 20.

That is staggeringly good, isn't it?

In BigOh terms this is **O(log n)** or **logarithmic complexity**. Now the logarithm in question could be ln (base e), $\log_{10}$, $\log_2$ or some other base. It doesn't matter it's still O(log n) just like $O(2n^2)$ and $O(100n^2)$ are still both $O(n^2)$.

It's worthwhile at this point to explain that BigOh can be used to determine three cases with an algorithm:

> - **Best Case:** In the telephone book search, the best case is that we find the name in one comparison. This is **O(1)** or **constant complexity**;
>
> - **Expected Case:** As discussed above this is O(log n); and
>
> - **Worst Case:** This is also O(log n).

Normally we don't care about the best case. We're interested in the expected and worst case. Sometimes one or the other of these will be more important.

Back to the telephone book.

What if you have a phone number and want to find a name? The police have a reverse phone book but such look-ups are denied to the general public. Or are they? Technically you can reverse look-up a number in an ordinary phone book. How?

You start at the first name and compare the number. If it's a match, great, if not, you move on to the next. You have to do it this way because the phone book is **unordered** (by phone number anyway).

So to find a name given the phone number (reverse lookup):

> - **Best Case:** O(1);
>
> - **Expected Case:** O(n) (for 500,000); and
>
> - **Worst Case:** O(n) (for 1,000,000).

The Traveling Salesman

This is quite a famous problem in computer science and deserves a mention. In this problem, you have N towns. Each of those towns is linked to 1 or more other towns by a road of a certain distance. The Traveling Salesman problem is to find the shortest tour that visits every town.

Sounds simple? Think again.

If you have 3 towns A, B, and C with roads between all pairs then you could go:

> - A → B → C
>
> - A → C → B
>
> - B → C → A

- B → A → C
  - C → A → B
  - C → B → A

Well, actually there's less than that because some of these are equivalent (A → B → C and C → B → A are equivalent, for example, because they use the same roads, just in reverse).

In actuality, there are 3 possibilities.

  - Take this to 4 towns and you have (iirc) 12 possibilities.
  - With 5 it's 60.
  - 6 becomes 360.

This is a function of a mathematical operation called a **factorial**. Basically:

  - 5! = 5 × 4 × 3 × 2 × 1 = 120
  - 6! = 6 × 5 × 4 × 3 × 2 × 1 = 720
  - 7! = 7 × 6 × 5 × 4 × 3 × 2 × 1 = 5040
  - …
  - 25! = 25 × 24 × … × 2 × 1 = 15,511,210,043,330,985,984,000,000
  - …

- 50! = 50 × 49 × … × 2 × 1 = 3.04140932 × 10^{64}

So the BigOh of the Traveling Salesman problem is **O(n!)** or **factorial or combinatorial complexity**.

By the time you get to 200 towns there isn't enough time left in the universe to solve the problem with traditional computers.

Something to think about.

Polynomial Time

Another point I wanted to make a quick mention of is that any algorithm that has a complexity of **O(n^a)** is said to have **polynomial complexity** or is solvable in **polynomial time**.

O(n), O(n^2) etc. are all polynomial time. Some problems cannot be solved in polynomial time. Certain things are used in the world because of this. [Public Key Cryptography](#) is a prime example. It is computationally hard to find two prime factors of a very large number. If it wasn't, we couldn't use the public key systems we use.

Anyway, that's it for my (hopefully plain English) explanation of BigOh (revised).

Share   Improve this answer   edited May 29, 2022 at 3:32

Gabriel Staples
51.5k ● 30 ● 273 ● 354

answered Jan 28, 2009 at 11:18

cletus
625k ● 169 ● 917 ● 945

---

624 While the other answers focus on explaining the differences between O(1), O(n^2) et al.... yours is the one which details how algorithms can get classified into n^2, nlog(n) etc. +1 for a good answer that helped me understand Big O notation as well – Yew Long Jan 28, 2009 at 11:42

---

30 one might want to add that big-O represents an upper bound (given by an algorithm), big-Omega give a lower bound (usually given as a proof independent from a specific algorithm) and big-Theta means that an "optimal" algorithm reaching that lower bound is known. – mdm Feb 2, 2009 at 19:16

---

23 This is good if you're looking for the longest answer, but not for the answer that best explains Big-O in a simple manner. – kirk.burleson Jul 16, 2010 at 18:21

---

180 -1: This is blatantly wrong: _"BigOh is relative representation of complexity of algorithm". No. BigOh is an asymptotic upper bound and exists quite well independent of computer science. O(n) is linear. No, you are confusing BigOh with theta. log n is O(n). 1 is O(n). The number of upvotes to this answer (and the comments), which makes the basic mistake of confusing Theta with BigOh is quite embarassing... – Aryabhatta May 24, 2011 at 4:44 ✎

---

78 *"By the time you get to 200 towns there isn't enough time left in the universe to solve the problem with traditional computers."* When the universe is going to end? – Isaac Jun 18, 2012 at 10:43 ✎

It shows how an algorithm scales based on input size.

O(n^2): known as **Quadratic complexity**

- 1 item: 1 operations
- 10 items: 100 operations
- 100 items: 10,000 operations

Notice that the number of items increases by a factor of 10, but the time increases by a factor of 10^2. Basically, n=10 and so O(n^2) gives us the scaling factor n^2 which is 10^2.

O(n): known as **Linear complexity**

- 1 item: 1 operation
- 10 items: 10 operations
- 100 items: 100 operations

This time the number of items increases by a factor of 10, and so does the time. n=10 and so O(n)'s scaling factor is 10.

O(1): known as **Constant complexity**

- 1 item: 1 operations
- 10 items: 1 operations
- 100 items: 1 operations

The number of items is still increasing by a factor of 10, but the scaling factor of O(1) is always 1.

O(log n): known as **Logarithmic complexity**

- 1 item: 1 operations

- 10 items: 2 operations

- 100 items: 3 operations

- 1000 items: 4 operations

- 10,000 items: 5 operations

The number of computations is only increased by a log of the input value. So in this case, assuming each computation takes 1 second, the log of the input `n` is the time required, hence `log n`.

That's the gist of it. They reduce the maths down so it might not be exactly n^2 or whatever they say it is, but that'll be the dominating factor in the scaling.

Share Improve this answer

Follow

edited Jun 26, 2022 at 20:13

Naresh Goradara
1,896 ● 2 ● 30 ● 41

answered Jan 28, 2009 at 11:28

Ray Hidayat
16.2k ● 4 ● 39 ● 43

5   what does this definition mean exactly? (The number of items is still increasing by a factor of 10, but the scaling factor of O(1) is always 1.) – Zach Smith Mar 25, 2010 at 22:10

114 Not seconds, operations. Also, you missed out on factorial and logarithmic time. – Chris Charabaruk Jul 17, 2010 at 1:27

7   This doesn't explain very well that O(n^2) could be describing an algorithm that runs in precisely .01*n^2 + 999999*n + 999999. It's important to know that algorithms are compared using this scale, and that the comparison works when n is 'sufficiently large'. Python's timsort actually uses insertion sort (worst/average case O(n^2)) for small arrays due to the fact that it has a small overhead. – Casey Kuball May 21, 2012 at 23:14

7   This answer also confuses big O notation and Theta notation. The function of n that returns 1 for all its inputs (usually simply written as 1) is actually in O(n^2) (even though it is also in O(1)). Similarly, an algorithm that only has to do one step which takes a constant amount of time is also considered to be an O(1) algorithm, but also to be an O(n) and an O(n^2) algorithm. But maybe mathematicians and computer scientists don't agree on the definition :-/. – Jacob Akkerboom Aug 8, 2013 at 11:11

2   The O(log n) Logarithmic complexity considered in this answer is of the base 10. Generally the standard is to calculate with base 2. One should keep in mind this fact and should not get confused. Also as mentioned by @ChrisCharabaruk the complexity denotes number of operations and not seconds. – akshit m Feb 17, 2016 at 6:46

449

Big-O notation (also called "asymptotic growth" notation) is *what functions "look like" when you ignore constant factors and stuff near the origin*. We use it to talk about **how thing scale**.

Basics

for "sufficiently" large inputs...

- `f(x) ∈ O(upperbound)` means `f` "grows no faster than" `upperbound`

- `f(x) ∈ θ(justlikethis)` mean `f` "grows exactly like" `justlikethis`

- `f(x) ∈ Ω(lowerbound)` means `f` "grows no slower than" `lowerbound`

big-O notation doesn't care about constant factors: the function `9x²` is said to "grow exactly like" `10x²`. Neither does big-O *asymptotic* notation care about *non-asymptotic* stuff ("stuff near the origin" or "what happens when the problem size is small"): the function `10x²` is said to "grow exactly like" `10x² - x + 2`.

Why would you want to ignore the smaller parts of the equation? Because they become completely dwarfed by the big parts of the equation as you consider larger and larger scales; their contribution becomes dwarfed and irrelevant. (See example section.)

Put another way, it's all about the **ratio** as you go to infinity. *If you divide the actual time it takes by the* `O(...)` *, you will get a constant factor in the limit of large inputs.* Intuitively this makes sense: functions "scale like" one another if you can multiply one to get the other. That is when we say...

```
actualAlgorithmTime(N) ∈ O(bound(N))
                                        e.g. "time
to mergesort N elements
                                              is
O(N log(N))"
```

... this means that **for "large enough" problem sizes N** (if we ignore stuff near the origin), there exists some constant (e.g. 2.5, completely made up) such that:

```
actualAlgorithmTime(N)                    e.g.
"mergesort_duration(N)        "
─────────────────────── < constant
─────────────────────── < 2.5
       bound(N)
N log(N)
```

There are many choices of constant; often the "best" choice is known as the "constant factor" of the algorithm... but we often ignore it like we ignore non-largest terms (see Constant Factors section for why they don't usually matter). You can also think of the above equation as a bound, saying "*In the worst-case scenario, the time it takes will never be worse than roughly* `N*log(N)` *, within a factor of 2.5 (a constant factor we don't care much about)*".

In general, `O(...)` is the most useful one because we often care about worst-case behavior. If `f(x)` represents something "bad" like the processor or memory usage, then " `f(x) ∈ O(upperbound)` " means " `upperbound` is the worst-case scenario of processor/memory usage".

Applications

As a purely mathematical construct, big-O notation is not limited to talking about processing time and memory. You can use it to discuss the asymptotics of anything where scaling is meaningful, such as:
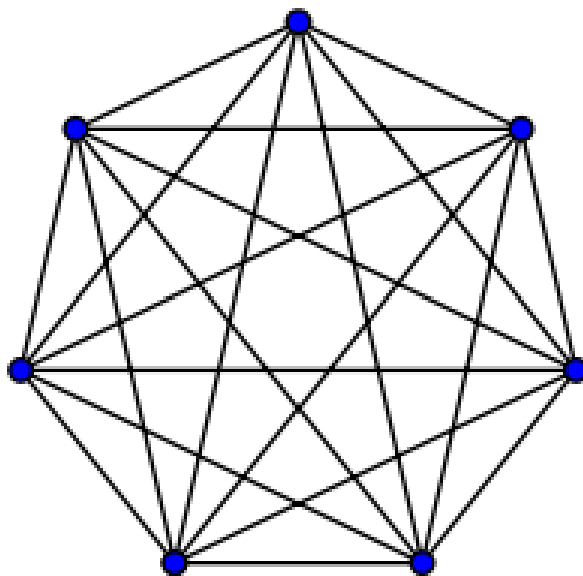
- the number of possible handshakes among `N` people at a party (`Θ(N²)` , specifically `N(N-1)/2` , but what matters is that it "scales like" `N²`)

- probabilistic expected number of people who have seen some viral marketing as a function of time

- how website latency scales with the number of processing units in a CPU or GPU or computer cluster

- how heat output scales on CPU dies as a function of transistor count, voltage, etc.

- how much time an algorithm needs to run, as a function of input size

- how much space an algorithm needs to run, as a function of input size

Example

For the handshake example above, everyone in a room shakes everyone else's hand. In that example, `#handshakes ∈ θ(N²)` . Why?

Back up a bit: the number of handshakes is exactly n-choose-2 or `N*(N-1)/2` (each of N people shakes the hands of N-1 other people, but this double-counts handshakes so divide by 2):

	1	2	3	4	5	6	7	8	9	10	11	...
1		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
3	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
4	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
5	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
6	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
7	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓
8	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
9	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
10	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
...	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

However, for very large numbers of people, the linear term `N` is dwarfed and effectively contributes 0 to the ratio (in the chart: the fraction of empty boxes on the diagonal over total boxes gets smaller as the number of participants becomes larger). Therefore the scaling behavior is `order N²`, or the number of handshakes "grows like N²".

```
#handshakes(N)
─────────────── ≈ 1/2
     N²
```

It's as if the empty boxes on the diagonal of the chart (N*(N-1)/2 checkmarks) weren't even there (N^2 checkmarks asymptotically).

(temporary digression from "plain English":) If you wanted to prove this to yourself, you could perform some simple algebra on the ratio to split it up into multiple terms (`lim` means "considered in the limit of", just ignore it if you haven't seen it, it's just notation for "and N is really really big"):

```
      N²/2 - N/2              (N²)/2   N/2              1/2
 lim ───────────── = lim (  ─────── - ─── ) = lim ─── =
 1/2
 N→∞       N²        N→∞      N²       N²       N→∞   1
                                      └──┘
               this is 0 in the limit of N→∞:
               graph it, or plug in a really large
 number for N
```

tl;dr: The number of handshakes 'looks like' x^2 so much for large values, that if we were to write down the ratio #handshakes/x^2, the fact that we don't need *exactly* x^2 handshakes wouldn't even show up in the decimal for an arbitrarily large while.

> e.g. for x=1million, ratio #handshakes/x^2: 0.499999...

Building Intuition

This lets us make statements like...

> "For large enough inputsize=N, no matter what the constant factor is, if I **double** *the input size*...

- ... I double the time an O(N) ("linear time") algorithm takes."

  Reveal spoiler ⚠

- ... I double-squared (quadruple) the time an O(N^2) ("quadratic time") algorithm takes." *(e.g. a problem 100x as big takes 100²=10000x as long... possibly unsustainable)*

  Reveal spoiler ⚠

- ... I double-cubed (octuple) the time an $O(N^3)$ ("cubic time") algorithm takes." *(e.g. a problem 100x as big takes 100^3=1000000x as long... very unsustainable)*

  Reveal spoiler ⚠

- ... I add a fixed amount to the time an $O(\log(N))$ ("logarithmic time") algorithm takes." *(cheap!)*

  Reveal spoiler ⚠

- ... I don't change the time an $O(1)$ ("constant time") algorithm takes." *(the cheapest!)*

  Reveal spoiler ⚠

- ... I "(basically) double" the time an $O(N \log(N))$ algorithm takes." *(fairly common)*

  Reveal spoiler ⚠

- ... I ridiculously increase the time a O(2^N) ("exponential time") algorithm takes." *(you'd double (or triple, etc.) the time just by increasing the problem by a single unit)*

Reveal spoiler ⚠

[for the mathematically inclined, you can mouse over the spoilers for minor sidenotes]

(with credit to https://stackoverflow.com/a/487292/711085)

(technically the constant factor could maybe matter in some more esoteric examples, but I've phrased things above (e.g. in log(N)) such that it doesn't)

These are the bread-and-butter orders of growth that programmers and applied computer scientists use as reference points. They see these all the time. (So while you could technically think "Doubling the input makes an O($\sqrt{N}$) algorithm 1.414 times slower," it's better to think of it as "this is worse than logarithmic but better than linear".)

Constant factors

Usually, we don't care what the specific constant factors are, because they don't affect the way the function grows. For example, two algorithms may both take `O(N)` time to complete, but one may be twice as slow as the other. We usually don't care too much unless the factor is very large since optimizing is tricky business ([When is optimisation premature?](#)); also the mere act of picking an algorithm with a better big-O will often improve performance by orders of magnitude.

Some asymptotically superior algorithms (e.g. a non-comparison `O(N log(log(N)))` sort) can have so large a constant factor (e.g. `100000*N log(log(N))`), or overhead that is relatively large like `O(N log(log(N)))` with a hidden `+ 100*N`, that they are rarely worth using even on "big data".

Why O(N) is sometimes the best you can do, i.e. why we need datastructures

`O(N)` algorithms are in some sense the "best" algorithms if you need to read all your data. The **very act of reading** a bunch of data is an `O(N)` operation. Loading it into memory is usually `O(N)` (or faster if you have hardware support, or no time at all if you've already read the data). However, if you touch or even *look* at every piece of data (or even every other piece of data), your algorithm will take `O(N)` time to perform this looking. No matter how long your actual algorithm takes, it will be at least `O(N)` because it spent that time looking at all the data.

The same can be said for the **very act of writing**. All algorithms which print out N things will take N time because the output is at least that long (e.g. printing out all permutations (ways to rearrange) a set of N playing cards is factorial: `O(N!)` (which is why in those cases, good programs will ensure an iteration uses O(1) memory and doesn't print or store every intermediate step)).

This motivates the use of **data structures**: a data structure requires reading the data only once (usually `O(N)` time), plus some arbitrary amount of preprocessing (e.g. `O(N)` or `O(N log(N))` or `O(N²)`) which we try to keep small. Thereafter, modifying the data structure (insertions/deletions/ etc.) and making queries on the data take very little time, such as `O(1)` or `O(log(N))`. You then proceed to make a large number of queries! In general, the more work you're willing to do ahead of time, the less work you'll have to do later on.

For example, say you had the latitude and longitude coordinates of millions of road segments and wanted to find all street intersections.

- Naive method: If you had the coordinates of a street intersection, and wanted to examine nearby streets, you would have to go through the millions of segments each time, and check each one for adjacency.

- If you only needed to do this once, it would not be a problem to have to do the naive method of `O(N)` work only once, but if you want to do it many times

(in this case, `N` times, once for each segment), we'd have to do `O(N²)` work, or 1000000²=1000000000000 operations. Not good (a modern computer can perform about a billion operations per second).

- If we use a simple structure called a hash table (an instant-speed lookup table, also known as a hashmap or dictionary), we pay a small cost by preprocessing everything in `O(N)` time. Thereafter, it only takes constant time on average to look up something by its key (in this case, our key is the latitude and longitude coordinates, rounded into a grid; we search the adjacent gridspaces of which there are only 9, which is a constant).

- Our task went from an infeasible `O(N²)` to a manageable `O(N)`, and all we had to do was pay a minor cost to make a hash table.

- **analogy**: The analogy in this particular case is a jigsaw puzzle: We created a data structure that exploits some property of the data. If our road segments are like puzzle pieces, we group them by matching color and pattern. We then exploit this to avoid doing extra work later (comparing puzzle pieces of like color to each other, not to every other single puzzle piece).

The moral of the story: a data structure lets us speed up operations. Even more, advanced data structures can let you combine, delay, or even ignore operations in incredibly clever ways. Different problems would have

different analogies, but they'd all involve organizing the data in a way that exploits some structure we care about, or which we've artificially imposed on it for bookkeeping. We do work ahead of time (basically planning and organizing), and now repeated tasks are much much easier!

Practical example: visualizing orders of growth while coding

Asymptotic notation is, at its core, quite separate from programming. Asymptotic notation is a mathematical framework for thinking about how things scale and can be used in many different fields. That said... this is how you *apply* asymptotic notation to coding.

The basics: Whenever we interact with every element in a collection of size A (such as an array, a set, all keys of a map, etc.), or perform A iterations of a loop, that is a multiplicative factor of size A. Why do I say "a multiplicative factor"?--because loops and functions (almost by definition) have multiplicative running time: the number of iterations, times work done in the loop (or for functions: the number of times you call the function, times work done in the function). (This holds if we don't do anything fancy, like skip loops or exit the loop early, or change control flow in the function based on arguments, which is very common.) Here are some examples of visualization techniques, with accompanying pseudocode.

(here, the `x`s represent constant-time units of work, processor instructions, interpreter opcodes, whatever)

```
for(i=0; i<A; i++)        // A * ...
    some O(1) operation    // 1

--> A*1 --> O(A) time

visualization:

|<------ A ------->|
1 2 3 4 5 x x ... x

other languages, multiplying orders of growth:
  javascript, O(A) time and space
    someListOfSizeA.map((x,i) => [x,i])
  python, O(rows*cols) time and space
    [[r*c for c in range(cols)] for r in
range(rows)]
```

Example 2:

```
for every x in listOfSizeA:   // A * (...
    some O(1) operation          // 1
    some O(B) operation          // B
    for every y in listOfSizeC: // C * (...
        some O(1) operation        // 1))

--> O(A*(1 + B + C))
    O(A*(B+C))           (1 is dwarfed)

visualization:

|<------ A ------->|
1 x x x x x x ... x

2 x x x x x x ... x ^
3 x x x x x x ... x |
4 x x x x x x ... x |
5 x x x x x x ... x B   <-- A*B
```

```
x x x x x x x ... x |
.................. |
x x x x x x x ... x v

x x x x x x x ... x ^
x x x x x x x ... x |
x x x x x x x ... x |
x x x x x x x ... x C   <-- A*C
x x x x x x x ... x |
.................. |
x x x x x x x ... x v
```

Example 3:

```
function nSquaredFunction(n) {
    total = 0
    for i in 1..n:          // N *
        for j in 1..n:      // N *
            total += i*k     // 1
    return total
}
// O(n^2)

function nCubedFunction(a) {
    for i in 1..n:                  // A *
        print(nSquaredFunction(a))  // A^2
}
// O(a^3)
```

If we do something slightly complicated, you might still be able to imagine visually what's going on:

```
for x in range(A):
    for y in range(1..x):
        simpleOperation(x*y)

x x x x x x x x x x x x |
x x x x x x x x x x x   |
x x x x x x x x        |
```

```
x x x x x x x          |
x x x x x x            |
x x x x x              |
x x x x                |
x x x                  |
x x                    |
x_____|
```

Here, the smallest recognizable outline you can draw is what matters; a triangle is a two dimensional shape (0.5 A^2), just like a square is a two-dimensional shape (A^2); the constant factor of two here remains in the asymptotic ratio between the two, however, we ignore it like all factors... (There are some unfortunate nuances to this technique I don't go into here; it can mislead you.)

Of course this does not mean that loops and functions are bad; on the contrary, they are the building blocks of modern programming languages, and we love them. However, we can see that the way we weave loops and functions and conditionals together with our data (control flow, etc.) mimics the time and space usage of our program! If time and space usage becomes an issue, that is when we resort to cleverness and find an easy algorithm or data structure we hadn't considered, to reduce the order of growth somehow. Nevertheless, these visualization techniques (though they don't always work) can give you a naive guess at a worst-case running time.

Here is another thing we can recognize visually:

```
<----------------------------- N -----------------
------------>
x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x
x x x x x x x x x x x x x x x x x x x x
x x x x x x x x
x x x x
x x
x
```

We can just rearrange this and see it's O(N):

```
<----------------------------- N -----------------
------------>
x x x x x x x x x x x x x x x x x x x x x x x x x x x x
x x x x x x x x
x x x x x x x x x x x x x x x x x|x x x x x x x x x|x
x x x|x x|x
```

Or maybe you do log(N) passes of the data, for O(N*log(N)) total time:

```
<----------------------------- N --------------
---------------->
 ^   x x x x x x x x x x x x x x x x x x x|x x x x x x x x
x x x x x x x x x x
 |   x x x x x x x x x|x x x x x x x x x|x x x x x x x x
x|x x x x x x x x
lgN  x x x x|x x x x|x x x x|x x x x|x x x x|x x x x
x|x x x x x|x x x x
 |   x x|x x|x x|x x|x x|x x|x x|x x|x x|x x|x x|x
x|x x|x x|x x|x x
 v
x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|x|
```

Unrelatedly but worth mentioning again: If we perform a hash (e.g. a dictionary/hashtable lookup), that is a factor

of O(1). That's pretty fast.

```
[myDictionary.has(x) for x in listOfSizeA]
 \----- O(1) ------/

--> A*1 --> O(A)
```

If we do something very complicated, such as with a recursive function or divide-and-conquer algorithm, ~~you can use the [Master Theorem](#) (usually works), or in ridiculous cases the Akra-Bazzi Theorem (almost always works)~~ you look up the running time of your algorithm on Wikipedia.

But, programmers don't think like this because eventually, algorithm intuition just becomes second nature. You will start to code something inefficient and immediately think "am I doing something **grossly inefficient?**". If the answer is "yes" AND you foresee it actually mattering, then you can take a step back and think of various tricks to make things run faster (the answer is almost always "use a hashtable", rarely "use a tree", and very rarely something a bit more complicated).

Amortized and average-case complexity

There is also the concept of "amortized" and/or "average case" (note that these are different).

Average Case: This is no more than using big-O notation for the expected value of a function, rather than the function itself. In the usual case where you consider all

inputs to be equally likely, the average case is just the average of the running time. For example with quicksort, even though the worst-case is `O(N^2)` for some really bad inputs, the average case is the usual `O(N log(N))` (the really bad inputs are very small in number, so few that we don't notice them in the average case).

Amortized Worst-Case: Some data structures may have a worst-case complexity that is large, but guarantee that if you do many of these operations, the average amount of work you do will be better than worst-case. For example, you may have a data structure that normally takes constant `O(1)` time. However, occasionally it will 'hiccup' and take `O(N)` time for one random operation, because maybe it needs to do some bookkeeping or garbage collection or something... but it promises you that if it does hiccup, it won't hiccup again for N more operations. The worst-case cost is still `O(N)` per operation, but the amortized cost *over many runs* is `O(N)/N` = `O(1)` per operation. Because the big operations are sufficiently rare, the massive amount of occasional work can be considered to blend in with the rest of the work as a constant factor. We say the work is "amortized" over a sufficiently large number of calls that it disappears asymptotically.

> The analogy for amortized analysis:
>
> You drive a car. Occasionally, you need to spend 10 minutes going to the gas station and then spend 1 minute refilling the tank with gas. If you

> did this every time you went anywhere with your car (spend 10 minutes driving to the gas station, spend a few seconds filling up a fraction of a gallon), it would be very inefficient. But if you fill up the tank once every few days, the 11 minutes spent driving to the gas station is "amortized" over a sufficiently large number of trips, that you can ignore it and pretend all your trips were maybe 5% longer.

Comparison between average-case and amortized worst-case:

- Average-case: We make some assumptions about our inputs; i.e. if our inputs have different probabilities, then our outputs/runtimes will have different probabilities (which we take the average of). Usually, we assume that our inputs are all equally likely (uniform probability), but if the real-world inputs don't fit our assumptions of "average input", the average output/runtime calculations may be meaningless. If you anticipate uniformly random inputs though, this is useful to think about!

- Amortized worst-case: If you use an amortized worst-case data structure, the performance is guaranteed to be within the amortized worst-case... eventually (even if the inputs are chosen by an evil demon who knows everything and is trying to screw you over). Usually, we use this to analyze algorithms that may be very 'choppy' in performance with unexpected

large hiccups, but over time perform just as well as other algorithms. (However unless your data structure has upper limits for much outstanding work it is willing to procrastinate on, an evil attacker could perhaps force you to catch up on the maximum amount of procrastinated work all-at-once.

Though, if you're [reasonably worried](#) about an attacker, there are many other algorithmic attack vectors to worry about besides amortization and average-case.)

Both average-case and amortization are incredibly useful tools for thinking about and designing with scaling in mind.

(See [Difference between average case and amortized analysis](#) if interested in this subtopic.)

Multidimensional big-O

Most of the time, people don't realize that there's more than one variable at work. For example, in a string-search algorithm, your algorithm may take time `O([length of text] + [length of query])`, i.e. it is linear in two variables like `O(N+M)`. Other more naive algorithms may be `O([length of text]*[length of query])` or `O(N*M)`. Ignoring multiple variables is one of the most common oversights I see in algorithm analysis, and can handicap you when designing an algorithm.

The whole story

Keep in mind that big-O is not the whole story. You can drastically speed up some algorithms by using caching, making them cache-oblivious, avoiding bottlenecks by working with RAM instead of disk, using parallelization, or doing work ahead of time -- these techniques are often *independent* of the order-of-growth "big-O" notation, though you will often see the number of cores in the big-O notation of parallel algorithms.

Also keep in mind that due to hidden constraints of your program, you might not really care about asymptotic behavior. You may be working with a bounded number of values, for example:

- If you're sorting something like 5 elements, you don't want to use the speedy `O(N log(N))` quicksort; you want to use insertion sort, which happens to perform well on small inputs. These situations often come up in divide-and-conquer algorithms, where you split up the problem into smaller and smaller subproblems, such as recursive sorting, fast Fourier transforms, or matrix multiplication.

- If some values are effectively bounded due to some hidden fact (e.g. the average human name is softly bounded at perhaps 40 letters, and human age is softly bounded at around 150). You can also impose bounds on your input to effectively make terms constant.

In practice, even among algorithms which have the same or similar asymptotic performance, their relative merit

may actually be driven by other things, such as: other performance factors (quicksort and mergesort are both `O(N log(N))`, but quicksort takes advantage of CPU caches); non-performance considerations, like ease of implementation; whether a library is available, and how reputable and maintained the library is.

Programs will also run slower on a 500MHz computer vs 2GHz computer. We don't really consider this as part of the resource bounds, because we think of the scaling in terms of machine resources (e.g. per clock cycle), not per real second. However, there are similar things which can 'secretly' affect performance, such as whether you are running under emulation, or whether the compiler optimized code or not. These might make some basic operations take longer (even relative to each other), or even speed up or slow down some operations asymptotically (even relative to each other). The effect may be small or large between different implementation and/or environment. Do you switch languages or machines to eke out that little extra work? That depends on a hundred other reasons (necessity, skills, coworkers, programmer productivity, the monetary value of your time, familiarity, workarounds, why not assembly or GPU, etc...), which may be more important than performance.

The above issues, like the effect of the choice of which programming language is used, are almost never considered as part of the constant factor (nor should they be); yet one should be aware of them because *sometimes* (though rarely) they may affect things. For

example in cpython, the native priority queue implementation is asymptotically non-optimal (`O(log(N))` rather than `O(1)` for your choice of insertion or find-min); do you use another implementation? Probably not, since the C implementation is probably faster, and there are probably other similar issues elsewhere. There are tradeoffs; sometimes they matter and sometimes they don't.

(edit: The "plain English" explanation ends here.)

Math addenda

For completeness, the precise definition of big-O notation is as follows: `f(x) ∈ O(g(x))` *means that "f is asymptotically upper-bounded by const*g": ignoring everything below some finite value of x, there exists a constant such that* `|f(x)| ≤ const * |g(x)|`*. (The other symbols are as follows: just like* `O` *means ≤,* `Ω` *means ≥. There are lowercase variants:* `o` *means <, and* `ω` *means >.)* `f(x) ∈ θ(g(x))` *means both* `f(x) ∈ O(g(x))` *and* `f(x) ∈ Ω(g(x))` *(upper- and lower-bounded by g): there exists some constants such that f will always lie in the "band" between* `const1*g(x)` *and* `const2*g(x)`*. It is the strongest asymptotic statement you can make and roughly equivalent to* `==`*. (Sorry, I elected to delay the mention of the absolute-value symbols until now, for clarity's sake; especially because I have never seen negative values come up in a computer science context.)*

People will often use `= O(...)`, which is perhaps the more correct 'comp-sci' notation, and entirely legitimate to use; "f = O(...)" is read "f is order ... / f is xxx-bounded by ..." and is thought of as "f is some expression whose asymptotics are ...". I was taught to use the more rigorous `∈ O(...)`. `∈` means "is an element of" (still read as before). In this particular case, `O(N²)` contains elements like { `2 N²`, `3 N²`, `1/2 N²`, `2 N² + log(N)`, `- N² + N^1.9`, ...} and is infinitely large, but it's still a set.

O and Ω are not symmetric (n = O(n²), but n² is not O(n)), but Θ is symmetric, and (since these relations are all transitive and reflexive) Θ, therefore, is symmetric and transitive and reflexive, and therefore partitions the set of all functions into equivalence classes. An equivalence class is a set of things that we consider to be the same. That is to say, given any function you can think of, you can find a canonical/unique 'asymptotic representative' of the class (by generally taking the limit... I think); just like you can group all integers into odds or evens, you can group all functions with Θ into x-ish, log(x)^2-ish, etc... by basically ignoring smaller terms (but sometimes you might be stuck with more complicated functions which are separate classes unto themselves).

The `=` notation might be the more common one and is even used in papers by world-renowned computer scientists. Additionally, it is often the case that in a casual setting, people will say `O(...)` when they mean `Θ(...)`; this is technically true since the set of things `Θ(exactlyThis)` is a subset of `O(noGreaterThanThis)` ... and it's easier to type. ;-)

edited Sep 8, 2021 at 11:33

answered Jul 8, 2011 at 4:46

ninjagecko
90.9k ● 24 ● 142 ● 152

---

33   An excellent mathematical answer, but the OP asked for a plain English answer. This level of mathematical description isn't required to understand the answer, though for people particularly mathematically minded it may be a lot simpler to understand than "plain English". However the OP asked for the latter. – El Zorko Jul 2, 2013 at 22:19

45   Presumably people other than the OP might have an interest in the answers to this question. Isn't that the guiding principle of the site? – Casey Feb 3, 2014 at 18:34

7   While I can maybe see why people might skim my answer and think it is too mathy (especially the "math is the new plain english" snide remarks, since removed), the original question asks about big-O which is about functions, so I attempt to be explicit and talk about functions in a way that complements the plain-English intuition. The math here can often be glossed over, or understood with a highschool math background. I do feel that people may look at the Math Addenda at the end though, and assume that is part of the answer, when it is merely there to see what the *real* math looks like. – ninjagecko Apr 3, 2015 at 4:39

5   This is a fantastic answer; much better IMO than the one with the most votes. The "math" required doesn't go beyond what's needed to understand the expressions in the parentheses after the "O," which no reasonable explanation that uses any examples can avoid. – Dave Abrahams Apr 27, 2016 at 15:52

256

EDIT: Quick note, this is almost certainly confusing [Big O notation](#) (which is an upper bound) with Theta notation (which is both an upper and lower bound). In my experience this is actually typical of discussions in non-academic settings. Apologies for any confusion caused.

In one sentence: As the size of your job goes up, how much longer does it take to complete it?

Obviously that's only using "size" as the input and "time taken" as the output — the same idea applies if you want to talk about memory usage etc.

Here's an example where we have N T-shirts which we want to dry. We'll *assume* it's incredibly quick to get them in the drying position (i.e. the human interaction is negligible). That's not the case in real life, of course...

- Using a washing line outside: assuming you have an infinitely large back yard, washing dries in O(1) time. However much you have of it, it'll get the same sun and fresh air, so the size doesn't affect the drying time.

- Using a tumble dryer: you put 10 shirts in each load, and then they're done an hour later. (Ignore the actual numbers here — they're irrelevant.) So drying 50 shirts takes *about* 5 times as long as drying 10 shirts.

- Putting everything in an airing cupboard: If we put everything in one big pile and just let general warmth do it, it will take a long time for the middle shirts to get dry. I wouldn't like to guess at the detail, but I suspect this is at least $O(N^2)$ — as you increase the wash load, the drying time increases faster.

One important aspect of "big O" notation is that it *doesn't* say which algorithm will be faster for a given size. Take a hashtable (string key, integer value) vs an array of pairs (string, integer). Is it faster to find a key in the hashtable or an element in the array, based on a string? (i.e. for the array, "find the first element where the string part matches the given key.") Hashtables are generally amortised (~= "on average") $O(1)$ — once they're set up, it should take about the same time to find an entry in a 100 entry table as in a 1,000,000 entry table. Finding an element in an array (based on content rather than index) is linear, i.e. $O(N)$ — on average, you're going to have to look at half the entries.

Does this make a hashtable faster than an array for lookups? Not necessarily. If you've got a very small collection of entries, an array may well be faster — you may be able to check all the strings in the time that it takes to just calculate the hashcode of the one you're

looking at. As the data set grows larger, however, the hashtable will eventually beat the array.

Share Improve this answer

Follow

edited Nov 8, 2011 at 6:15

answered Jan 28, 2009 at 11:16

Jon Skeet
1.5m ● 889 ● 9.3k ● 9.3k

---

6 A hashtable requires an algorithm to run to calculate the index of the actual array (depending on the implementation). And an array just have O(1) because it's just an adress. But this has nothing to do with the question, just an observation :) – Filip Ekberg Jan 28, 2009 at 11:29

---

7 jon's explanation has very much todo with the question i think. it's exactly how one could explain it to some mum, and she would eventually understand it i think :) i like the clothes example (in particular the last, where it explains the exponential growth of complexity) – Johannes Schaub - litb Jan 28, 2009 at 11:32

---

4 Filip: I'm not talking about address an array by index, I'm talking about finding a matching entry in an array. Could you reread the answer and see if that's still unclear? – Jon Skeet Jan 28, 2009 at 11:35

---

3 @Filip Ekberg I think you're thinking of a direct-address table where each index maps to a key directly hence is O(1), however I believe Jon is talking about an unsorted array of key/val pairs which you have to search through linearly. – ljs Jul 29, 2011 at 9:41

---

2 @RBT: No, it's not a binary look-up. It can get to the right hash *bucket* immediately, just based on a transformation from hash code to bucket index. After that, finding the right hash

code in the bucket may be linear or it may be a binary search... but by that time you're down to a very small proportion of the total size of the dictionary. – Jon Skeet Nov 17, 2016 at 7:57

137

Big O describes an upper limit on the growth behaviour of a function, for example the runtime of a program, when inputs become large.

Examples:

- O(n): If I double the input size the runtime doubles

- $O(n^2)$: If the input size doubles the runtime quadruples

- O(log n): If the input size doubles the runtime increases by one

- $O(2^n)$: If the input size increases by one, the runtime doubles

The input size is usually the space in bits needed to represent the input.

Share Improve this answer

Follow

edited Jan 11, 2014 at 11:11

answered Jan 28, 2009 at 11:23

starblue

56.7k ● 14 ● 99 ● 153

7   incorrect! for example O(n): If I double the input size the runtime will multiply to finite non zero constant. I mean O(n) = O(n + n) – arena-ru May 16, 2010 at 11:33

7   I'm talking about the f in f(n) = O(g(n)), not the g as you seem to understand. – starblue Aug 6, 2010 at 12:30

I upvoted, but the last sentence doesn't contribute much I feel. We don't often talk about "bits" when discussing or measuring Big(O). – cdiggins Sep 5, 2011 at 16:41

5   You should add an example for O(n log n).
– Christoffer Hammarström Sep 22, 2011 at 15:50

That's not so clear, essentially it behaves a little worse than O(n). So if n doubles, the runtime is multiplied by a factor somewhat larger than 2. – starblue Sep 23, 2011 at 6:44

▲

112

▼

🔖

🕘

Big O notation is most commonly used by programmers as an approximate measure of how long a computation (algorithm) will take to complete expressed as a function of the size of the input set.

Big O is useful to compare how well two algorithms will scale up as the number of inputs is increased.

More precisely Big O notation is used to express the asymptotic behavior of a function. That means how the function behaves as it approaches infinity.

In many cases the "O" of an algorithm will fall into one of the following cases:

- **O(1)** - Time to complete is the same regardless of the size of input set. An example is accessing an array element by index.

- **O(Log N)** - Time to complete increases roughly in line with the log2(n). For example 1024 items takes roughly twice as long as 32 items, because Log2(1024) = 10 and Log2(32) = 5. An example is finding an item in a [binary search tree](#) (BST).

- **O(N)** - Time to complete that scales linearly with the size of the input set. In other words if you double the number of items in the input set, the algorithm takes roughly twice as long. An example is counting the number of items in a linked list.

- **O(N Log N)** - Time to complete increases by the number of items times the result of Log2(N). An example of this is [heap sort](#) and [quick sort](#).

- **O(N^2)** - Time to complete is roughly equal to the square of the number of items. An example of this is [bubble sort](#).

- **O(N!)** - Time to complete is the factorial of the input set. An example of this is the [traveling salesman problem brute-force solution](#).

Big O ignores factors that do not contribute in a meaningful way to the growth curve of a function as the input size increases towards infinity. This means that constants that are added to or multiplied by the function are simply ignored.

cdiggins, what if i have O(N/2) complexity , should it be O(N) or O(N/2), for example what the complexity if i will loop over half string. – Melad Basilius May 12, 2017 at 15:08

1   @Melad This is an example of a constant (0.5) being multiplied to the function. This is ignored as it is considered to have a meaningful effect for very large values of N. – cdiggins May 18, 2017 at 19:47 ✏️

i like this answer because of its conneting each BigO to some examples such as data structures or algorithms common problems – danfromisrael Dec 27, 2022 at 4:15

91

Big O is just a way to "Express" yourself in a common way, "How much time / space does it take to run my code?".

You may often see O(n), O(n^2), O(nlogn) and so forth, all these are just ways to show; How does an algorithm change?

O(n) means Big O is n, and now you might think, "What is n!?" Well "n" is the amount of elements. Imaging you want to search for an Item in an Array. You would have to look on Each element and as "Are you the correct element/item?" in the worst case, the item is at the last

index, which means that it took as much time as there are items in the list, so to be generic, we say "oh hey, n is a fair given amount of values!".

So then you might understand what "n^2" means, but to be even more specific, play with the thought you have a simple, the simpliest of the sorting algorithms; bubblesort. This algorithm needs to look through the whole list, for each item.

My list

1. 1

2. 6

3. 3

The flow here would be:

- Compare 1 and 6, which is biggest? Ok 6 is in the right position, moving forward!

- Compare 6 and 3, oh, 3 is less! Let's move that, Ok the list changed, we need to start from the begining now!

This is O n^2 because, you need to look at all items in the list there are "n" items. For each item, you look at all items once more, for comparing, this is also "n", so for every item, you look "n" times meaning n*n = n^2

I hope this is as simple as you want it.

But remember, Big O is just a way to experss yourself in the manner of time and space.

edited Oct 19, 2014 at 20:47

ROMANIA_engineer
56.5k ● 30 ● 208 ● 205

answered Jan 28, 2009 at 11:14

Filip Ekberg
36.3k ● 20 ● 128 ● 184

▲

58

▼

🔖

🕓

Big O describes the fundamental scaling nature of an algorithm.

There is a lot of information that Big O does not tell you about a given algorithm. It cuts to the bone and gives only information about the scaling nature of an algorithm, specifically how the resource use (think time or memory) of an algorithm scales in response to the "input size".

Consider the difference between a steam engine and a rocket. They are not merely different varieties of the same thing (as, say, a Prius engine vs. a Lamborghini engine) but they are dramatically different kinds of propulsion systems, at their core. A steam engine may be faster than a toy rocket, but no steam piston engine will be able to achieve the speeds of an orbital launch vehicle. This is because these systems have different scaling characteristics with regards to the relation of fuel required ("resource usage") to reach a given speed ("input size").

Why is this so important? Because software deals with problems that may differ in size by factors up to a trillion. Consider that for a moment. The ratio between the speed necessary to travel to the Moon and human walking speed is less than 10,000:1, and that is absolutely tiny compared to the range in input sizes software may face. And because software may face an astronomical range in input sizes there is the potential for the Big O complexity of an algorithm, it's fundamental scaling nature, to trump any implementation details.

Consider the canonical sorting example. Bubble-sort is $O(n^2)$ while merge-sort is $O(n \log n)$. Let's say you have two sorting applications, application A which uses bubble-sort and application B which uses merge-sort, and let's say that for input sizes of around 30 elements application A is 1,000x faster than application B at sorting. If you never have to sort much more than 30 elements then it's obvious that you should prefer application A, as it is much faster at these input sizes. However, if you find that you may have to sort ten million items then what you'd expect is that application B actually ends up being thousands of times faster than application A in this case, entirely due to the way each algorithm scales.

Share   Improve this answer

Follow

Here is the plain English bestiary I tend to use when explaining the common varieties of Big-O

In all cases, prefer algorithms higher up on the list to those lower on the list. However, the cost of moving to a more expensive complexity class varies significantly.

O(1):

No growth. Regardless of how big as the problem is, you can solve it in the same amount of time. This is somewhat analogous to broadcasting where it takes the same amount of energy to broadcast over a given distance, regardless of the number of people that lie within the broadcast range.

O(log n):

This complexity is the same as **O(1)** except that it's just a little bit worse. For all practical purposes, you can consider this as a very large constant scaling. The difference in work between processing 1 thousand and 1 billion items is only a factor six.

O(n):

The cost of solving the problem is proportional to the size of the problem. If your problem doubles in size, then the cost of the solution doubles. Since most problems have to be scanned into the computer in some way, as data entry,

disk reads, or network traffic, this is generally an affordable scaling factor.

O(*n* log *n*):

This complexity is very similar to **O(*n*)**. For all practical purposes, the two are equivalent. This level of complexity would generally still be considered scalable. By tweaking assumptions some **O(*n* log *n*)** algorithms can be transformed into **O(*n*)** algorithms. For example, bounding the size of keys reduces sorting from **O(*n* log *n*)** to **O(*n*)**.

O(n^2):

Grows as a square, where *n* is the length of the side of a square. This is the same growth rate as the "network effect", where everyone in a network might know everyone else in the network. Growth is expensive. Most scalable solutions cannot use algorithms with this level of complexity without doing significant gymnastics. This generally applies to all other polynomial complexities - **O(n^k)** - as well.

O(2^n):

Does not scale. You have no hope of solving any non-trivially sized problem. Useful for knowing what to avoid, and for experts to find approximate algorithms which are in **O(n^k)**.

2   Could you please consider a different analogy for O(1)? The engineer in me wants to pull out a discussion about RF impedance due to obstructions. – johnwbyrd Mar 10, 2016 at 23:02

Big O is a measure of how much time/space an algorithm uses relative to the size of its input.

If an algorithm is O(n) then the time/space will increase at the same rate as its input.

If an algorithm is O(n^2) then the time/space increase at the rate of its input squared.

and so on.

Share   Improve this answer

Follow

2   It's not about space. It's about complexity which means time. – S.Lott Jan 28, 2009 at 11:35

14   I have always believed it can be about time OR space. but not about both at the same time. – Rocco Jan 28, 2009 at 12:58

9   Complexity most definitely can be about space. Have a look at this: en.wikipedia.org/wiki/PSPACE – Tom Crockett Aug 8, 2010 at 15:58

4   This answer is the most "plain" one here. Previous ones actually assume readers know enough to understand them but writers are not aware of it. They think theirs are simple and plain, which are absolutely not. Writing a lot text with pretty format and making fancy artificial examples that are hard to non-CS people is not plain and simple, it is just attractive to stackoverflowers who are mostly CS people to up vote. Explaining CS term in plain English needs nothing about code and math at all. +1 for this answer though it is still not good enough. – W.Sun May 29, 2013 at 12:36 ✏

This answer makes the (common) error of assuming that f=O(g) means that f and g are proportional. – Paul Hankin Apr 3, 2015 at 4:38

▲

38

▼

🔖

🕓

It is very difficult to measure the speed of software programs, and when we try, the answers can be very complex and filled with exceptions and special cases. This is a big problem, because all those exceptions and special cases are distracting and unhelpful when we want to compare two different programs with one another to find out which is "fastest".

As a result of all this unhelpful complexity, people try to describe the speed of software programs using the smallest and least complex (mathematical) expressions

possible. These expressions are very very crude approximations: Although, with a bit of luck, they will capture the "essence" of whether a piece of software is fast or slow.

Because they are approximations, we use the letter "O" (Big Oh) in the expression, as a convention to signal to the reader that we are making a gross oversimplification. (And to make sure that nobody mistakenly thinks that the expression is in any way accurate).

If you read the "Oh" as meaning "on the order of" or "approximately" you will not go too far wrong. (I think the choice of the Big-Oh might have been an attempt at humour).

The only thing that these "Big-Oh" expressions try to do is to describe how much the software slows down as we increase the amount of data that the software has to process. If we double the amount of data that needs to be processed, does the software need twice as long to finish it's work? Ten times as long? In practice, there are a very limited number of big-Oh expressions that you will encounter and need to worry about:

The good:

- `O(1)` **Constant**: The program takes the same time to run no matter how big the input is.
- `O(log n)` **Logarithmic**: The program run-time increases only slowly, even with big increases in the size of the input.

The bad:

- `O(n)` **Linear**: The program run-time increases proportionally to the size of the input.

- `O(n^k)` **Polynomial**: - Processing time grows faster and faster - as a polynomial function - as the size of the input increases.

... and the ugly:

- `O(k^n)` **Exponential** The program run-time increases very quickly with even moderate increases in the size of the problem - it is only practical to process small data sets with exponential algorithms.

- `O(n!)` **Factorial** The program run-time will be longer than you can afford to wait for anything but the very smallest and most trivial-seeming datasets.

Share   Improve this answer

Follow

answered May 29, 2013 at 13:51

[William Payne](#)

3,285 ● 3 ● 25 ● 25

---

5   I've also heard the term Linearithmic - `O(n log n)` which would be considered good. – Jason Down May 29, 2013 at 18:45

▲

37

What is a plain English explanation of Big O? With as little formal definition as possible and simple mathematics.

A Plain English Explanation of the *Need* for Big-O Notation:

When we program, we are trying to solve a problem. What we code is called an algorithm. Big O notation allows us to compare the worse case performance of our algorithms in a standardized way. Hardware specs vary over time and improvements in hardware can reduce the time it takes an algorithms to run. But replacing the hardware does not mean our algorithm is any better or improved over time, as our algorithm is still the same. So in order to allow us to compare different algorithms, to determine if one is better or not, we use Big O notation.

A Plain English Explanation of *What* Big O Notation is:

Not all algorithms run in the same amount of time, and can vary based on the number of items in the input, which we'll call n. Based on this, we consider the worse case analysis, or an upper-bound of the run-time as n get larger and larger. We must be aware of what n is, because many of the Big O notations reference it.

Share   Improve this answer

Follow

Ok, my 2cents.

Big-O, is **rate of increase** of resource consumed by program, w.r.t. problem-instance-size

Resource : Could be total-CPU time, could be maximum RAM space. By default refers to CPU time.

Say the problem is "Find the sum",

```
int Sum(int*arr,int size){
    int sum=0;
    while(size-->0)
       sum+=arr[size];

    return sum;
}
```

problem-instance= {5,10,15} ==> problem-instance-size = 3, iterations-in-loop= 3

problem-instance= {5,10,15,20,25} ==> problem-instance-size = 5 iterations-in-loop = 5

For input of size "n" the program is growing at speed of "n" iterations in array. Hence Big-O is N expressed as O(n)

Say the problem is "Find the Combination",

```
    void Combination(int*arr,int size)
    { int outer=size,inner=size;
      while(outer -->0) {
        inner=size;
```

```
            while(inner -->0)
               cout<<arr[outer]<<"-"<<arr[inner]<<endl;
        }
      }
```

problem-instance= {5,10,15} ==> problem-instance-size = 3, total-iterations = 3*3 = 9

problem-instance= {5,10,15,20,25} ==> problem-instance-size = 5, total-iterations= 5*5 =25

For input of size "n" the program is growing at speed of "n*n" iterations in array. Hence Big-O is N^2 expressed as $O(n^2)$

Share   Improve this answer

Follow

3   `while (size-->0)` I hope this wouldn't ask again. – mr5 Jun 18, 2013 at 14:41

A simple straightforward answer can be:

31

Big O represents the worst possible time/space for that algorithm. The algorithm will never take more space/time above that limit. Big O represents time/space complexity in the extreme case.

29

Big O notation is a way of describing the upper bound of an algorithm in terms of space or running time. The n is the number of elements in the the problem (i.e size of an array, number of nodes in a tree, etc.) We are interested in describing the running time as n gets big.

When we say some algorithm is O(f(n)) we are saying that the running time (or space required) by that algorithm is always lower than some constant times f(n).

To say that binary search has a running time of O(logn) is to say that there exists some constant c which you can multiply log(n) by that will always be larger than the running time of binary search. In this case you will always have some constant factor of log(n) comparisons.

In other words where g(n) is the running time of your algorithm, we say that g(n) = O(f(n)) when g(n) <= c*f(n) when n > k, where c and k are some constants.

28

"*What is a plain English explanation of Big O? With as little formal definition as possible and simple mathematics.*"

Such a beautifully simple and short question seems at least to deserve an equally short answer, like a student might receive during tutoring.

> Big O notation simply tells how much time* an algorithm can run within, in terms of *only the amount of input data**.*

(*in a wonderful, *unit-free* sense of time!)
(**which is what matters, because people will *always want more*, whether they live today or tomorrow)

Well, what's so wonderful about Big O notation if that's what it does?

- Practically speaking, Big O analysis is *so useful and important* because Big O puts the focus squarely on the algorithm's *own* complexity and completely *ignores* anything that is merely a proportionality constant—like a JavaScript engine, the speed of a

CPU, your Internet connection, and all those things which become quickly become as laughably outdated as a Model *T*. Big O focuses on performance only in the way that matters equally as much to people living in the present or in the future.

- Big O notation also shines a spotlight directly on the most important principle of computer programming/engineering, the fact which inspires all good programmers to keep thinking and dreaming: the only way to achieve results beyond the slow forward march of technology is to *invent a better algorithm*.

Share   Improve this answer

Follow

edited Aug 24, 2013 at 6:50

answered Aug 15, 2013 at 1:57

Joseph Myers

6,542 ● 28 ● 36

---

5   Being asked to explain something mathematical without mathematics is always a personal challenge to me, as a bona fide Ph.D. mathematician and teacher who believes that such a thing is actually possible. And being a programmer as well, I hope that no one minds that I found answering this particular question, without mathematics, to be a challenge that was completely irresistible. – Joseph Myers Aug 15, 2013 at 2:09

Algorithm example (Java):

26

```java
public boolean search(/* for */Integer K,/* in */List<
{
    for(/* each */Integer i:/* in */L)
    {
        if(i == K)
        {
            return true;
        }
    }

    return false;
}
```

Algorithm description:

- This algorithm searches a list, item by item, looking for a key,

- Iterating on each item in the list, if it's the key then return True,

- If the loop has finished without finding the key, return False.

Big-O notation represents the upper-bound on the Complexity (Time, Space, ..)

To find The Big-O on Time Complexity:

- Calculate how much time (regarding input size) the worst case takes:

- Worst-Case: the key doesn't exist in the list.

- Time(Worst-Case) = 4n+1

- Time: O(4n+1) = O(n) | in Big-O, constants are neglected

- O(n) ~ Linear

There's also Big-Omega, which represent the complexity of the Best-Case:

- Best-Case: the key is the first item.

- Time(Best-Case) = 4

- Time: $\Omega(4) = O(1)$ ~ Instant\Constant

Share   Improve this answer

Follow

answered Mar 23, 2013 at 15:19

Khaled.K
5,940 ● 1 ● 34 ● 52

---

2   Where does your constant 4 comes from? – Rod Jul 4, 2014 at 13:34

---

2   @Rod iterator init, iterator comparison, iterator read, key comparison.. I think `C` would be better – Khaled.K Jul 6, 2014 at 11:04

▲

23

▼

🔖

Big O notation is a way of describing how quickly an algorithm will run given an arbitrary number of input parameters, which we'll call "n". It is useful in computer science because different machines operate at different speeds, and simply saying that an algorithm takes 5 seconds doesn't tell you much because while you may be running a system with a 4.5 Ghz octo-core processor, I

may be running a 15 year old, 800 Mhz system, which could take longer regardless of the algorithm. So instead of specifying how fast an algorithm runs in terms of time, we say how fast it runs in terms of number of input parameters, or "n". By describing algorithms in this way, we are able to compare the speeds of algorithms without having to take into account the speed of the computer itself.

Share Improve this answer

Follow

Big O

20

$f(x) = O(g(x))$ when x goes to a (for example, a = +∞) means that there is a function k such that:

1. $f(x) = k(x)g(x)$

2. k is bounded in some neighborhood of a (if a = +∞, this means that there are numbers N and M such that for every x > N, $|k(x)| < M$).

In other words, in plain English: $f(x) = O(g(x))$, x → a, means that in a neighborhood of a, f decomposes into the product of g and some bounded function.

Small o

By the way, here is for comparison the definition of small o.

$f(x) = o(g(x))$ when x goes to a means that there is a function k such that:

1. $f(x) = k(x)g(x)$

2. $k(x)$ goes to 0 when x goes to a.

Examples

- $\sin x = O(x)$ when $x \to 0$.

- $\sin x = O(1)$ when $x \to +\infty$,

- $x^2 + x = O(x)$ when $x \to 0$,

- $x^2 + x = O(x^2)$ when $x \to +\infty$,

- $\ln(x) = o(x) = O(x)$ when $x \to +\infty$.

Attention! The notation with the equal sign "=" uses a "fake equality": it is true that $o(g(x)) = O(g(x))$, but false that $O(g(x)) = o(g(x))$. Similarly, it is ok to write "$\ln(x) = o(x)$ when $x \to +\infty$", but the formula "$o(x) = \ln(x)$" would make no sense.

More examples

- $O(1) = O(n) = O(n^2)$ when $n \to +\infty$ (but not the other way around, the equality is "fake"),

- $O(n) + O(n^2) = O(n^2)$ when $n \to +\infty$

- $O(O(n^2)) = O(n^2)$ when $n \to +\infty$

- $O(n^2)O(n^3) = O(n^5)$ when $n \to +\infty$

Here is the Wikipedia article:

https://en.wikipedia.org/wiki/Big_O_notation

Share   Improve this answer

Follow

3   You are stating "Big O" and "Small o" without explainy what they are, introducing lots of mathematical concepts without telling why they are important and the link to wikipedia may be in this case too obvious for this kind of question. – ASX Jan 11, 2014 at 14:54

@AditSaxena What do you mean "without explaining what they are"? I exactly explained what they are. That is, "big O" and "small o" are nothing by themselves, only a formula like "f(x) = O(g(x))" has a meaning, which i explained (in plain English, but without defining of course all the necessary things from a Calculus course). Sometimes "O(f(x))" is viewed as the class (actually the set) of all the functions "g(x)" such that "g(x) = O(f(x))", but this is an extra step, which is not necessary for understanding the basics.
– Alexey Jan 11, 2014 at 18:34 ✏

Well, ok, there are words that are not plain English, but it is inevitable, unless i would have to include all necessary definitions from Mathematical Analysis. – Alexey Jan 11, 2014 at 18:38 ✏

2   Hi #Alexey, please have a look at accepted answer: it is long but it is well constructed and well formatted. It starts with a

simple definition with no mathematical background needed. While doing so he introduce thre 3 "technical" words which he explains immediately (relative, representation, complexity). This goes on step by step while digging into this field. – ASX Jan 12, 2014 at 22:44

2   Big O is used for understanding asymptotic behavior of algorithms for the same reason it is used for understanding asymptotic behavior of functions (asymptotic behavior is the behavior near infinity). It is a convenient notation for comparing a complicated function (the actual time or space the algorithm takes) to simple ones (anything simple, usually a power function) near infinity, or near anything else. I only explained what it is (gave the definition). How to compute with big O is a different story, maybe i'll add some examples, since you are interested. – Alexey Jan 13, 2014 at 17:23 ✎

▲

14

▼

🔖

🕑

You want to know all there is to know of big O? So do I.

So to talk of big O, I will use words that have just one beat in them. One sound per word. Small words are quick. You know these words, and so do I. We will use words with one sound. They are small. I am sure you will know all of the words we will use!

Now, let's you and me talk of work. Most of the time, I do not like work. Do you like work? It may be the case that you do, but I am sure I do not.

I do not like to go to work. I do not like to spend time at work. If I had my way, I would like just to play, and do fun things. Do you feel the same as I do?

Now at times, I do have to go to work. It is sad, but true. So, when I am at work, I have a rule: I try to do less work. As near to no work as I can. Then I go play!

So here is the big news: the big O can help me not to do work! I can play more of the time, if I know big O. Less work, more play! That is what big O helps me do.

Now I have some work. I have this list: one, two, three, four, five, six. I must add all things in this list.

Wow, I hate work. But oh well, I have to do this. So here I go.

One plus two is three… plus three is six... and four is... I don't know. I got lost. It is too hard for me to do in my head. I don't much care for this kind of work.

So let's not do the work. Let's you and me just think how hard it is. How much work would I have to do, to add six numbers?

Well, let's see. I must add one and two, and then add that to three, and then add that to four… All in all, I count six adds. I have to do six adds to solve this.

Here comes big O, to tell us just how hard this math is.

Big O says: we must do six adds to solve this. One add, for each thing from one to six. Six small bits of work... each bit of work is one add.

Well, I will not do the work to add them now. But I know how hard it would be. It would be six adds.

Oh no, now I have more work. Sheesh. Who makes this kind of stuff?!

Now they ask me to add from one to ten! Why would I do that? I did not want to add one to six. To add from one to ten… well… that would be even more hard!

How much more hard would it be? How much more work would I have to do? Do I need more or less steps?

Well, I guess I would have to do ten adds… one for each thing from one to ten. Ten is more than six. I would have to work that much more to add from one to ten, than one to six!

I do not want to add right now. I just want to think on how hard it might be to add that much. And, I hope, to play as soon as I can.

To add from one to six, that is some work. But do you see, to add from one to ten, that is more work?

Big O is your friend and mine. Big O helps us think on how much work we have to do, so we can plan. And, if we are friends with big O, he can help us choose work that is not so hard!

Now we must do new work. Oh, no. I don't like this work thing at all.

The new work is: add all things from one to n.

Wait! What is n? Did I miss that? How can I add from one to n if you don't tell me what n is?

Well, I don't know what n is. I was not told. Were you? No? Oh well. So we can't do the work. Whew.

But though we will not do the work now, we can guess how hard it would be, if we knew n. We would have to add up n things, right? Of course!

Now here comes big O, and he will tell us how hard this work is. He says: to add all things from one to N, one by one, is O(n). To add all these things, [I know I must add n times.][1] That is big O! He tells us how hard it is to do some type of work.

To me, I think of big O like a big, slow, boss man. He thinks on work, but he does not do it. He might say, "That work is quick." Or, he might say, "That work is so slow and hard!" But he does not do the work. He just looks at the work, and then he tells us how much time it might take.

I care lots for big O. Why? I do not like to work! No one likes to work. That is why we all love big O! He tells us how fast we can work. He helps us think of how hard work is.

Uh oh, more work. Now, let's not do the work. But, let's make a plan to do it, step by step.

They gave us a deck of ten cards. They are all mixed up: seven, four, two, six… not straight at all. And now... our job is to sort them.

Ergh. That sounds like a lot of work!

How can we sort this deck? I have a plan.

I will look at each pair of cards, pair by pair, through the deck, from first to last. If the first card in one pair is big and the next card in that pair is small, I swap them. Else, I go to the next pair, and so on and so on... and soon, the deck is done.

When the deck is done, I ask: did I swap cards in that pass? If so, I must do it all once more, from the top.

At some point, at some time, there will be no swaps, and our sort of the deck would be done. So much work!

Well, how much work would that be, to sort the cards with those rules?

I have ten cards. And, most of the time -- that is, if I don't have lots of luck -- I must go through the whole deck up to ten times, with up to ten card swaps each time through the deck.

Big O, help me!

Big O comes in and says: for a deck of n cards, to sort it this way will be done in O(N squared) time.

Why does he say n squared?

Well, you know n squared is n times n. Now, I get it: n cards checked, up to what might be n times through the deck. That is two loops, each with n steps. That is n squared much work to be done. A lot of work, for sure!

Now when big O says it will take O(n squared) work, he does not mean n squared adds, on the nose. It might be some small bit less, for some case. But in the worst case, it will be near n squared steps of work to sort the deck.

Now here is where big O is our friend.

Big O points out this: as n gets big, when we sort cards, the job gets MUCH MUCH MORE HARD than the old just-add-these-things job. How do we know this?

Well, if n gets real big, we do not care what we might add to n or n squared.

For big n, n squared is more large than n.

Big O tells us that to sort things is more hard than to add things. O(n squared) is more than O(n) for big n. That means: if n gets real big, to sort a mixed deck of n things MUST take more time, than to just add n mixed things.

Big O does not solve the work for us. Big O tells us how hard the work is.

I have a deck of cards. I did sort them. You helped. Thanks.

Is there a more fast way to sort the cards? Can big O help us?

Yes, there is a more fast way! It takes some time to learn, but it works... and it works quite fast. You can try it too, but take your time with each step and do not lose your place.

In this new way to sort a deck, we do not check pairs of cards the way we did a while ago. Here are your new rules to sort this deck:

One: I choose one card in the part of the deck we work on now. You can choose one for me if you like. (The first time we do this, "the part of the deck we work on now" is the whole deck, of course.)

Two: I splay the deck on that card you chose. What is this splay; how do I splay? Well, I go from the start card down, one by one, and I look for a card that is more high than the splay card.

Three: I go from the end card up, and I look for a card that is more low than the splay card.

Once I have found these two cards, I swap them, and go on to look for more cards to swap. That is, I go back to step Two, and splay on the card you chose some more.

At some point, this loop (from Two to Three) will end. It ends when both halves of this search meet at the splay card. Then, we have just splayed the deck with the card

you chose in step One. Now, all the cards near the start are more low than the splay card; and the cards near the end are more high than the splay card. Cool trick!

Four (and this is the fun part): I have two small decks now, one more low than the splay card, and one more high. Now I go to step one, on each small deck! That is to say, I start from step One on the first small deck, and when that work is done, I start from step One on the next small deck.

I break up the deck in parts, and sort each part, more small and more small, and at some time I have no more work to do. Now this may seem slow, with all the rules. But trust me, it is not slow at all. It is much less work than the first way to sort things!

What is this sort called? It is called Quick Sort! That sort was made by a man called [C. A. R. Hoare](#) and he called it Quick Sort. Now, Quick Sort gets used all the time!

Quick Sort breaks up big decks in small ones. That is to say, it breaks up big tasks in small ones.

Hmmm. There may be a rule in there, I think. To make big tasks small, break them up.

This sort is quite quick. How quick? Big O tells us: this sort needs $O(n \log n)$ work to be done, in the mean case.

Is it more or less fast than the first sort? Big O, please help!

The first sort was O(n squared). But Quick Sort is O(n log n). You know that n log n is less than n squared, for big n, right? Well, that is how we know that Quick Sort is fast!

If you have to sort a deck, what is the best way? Well, you can do what you want, but I would choose Quick Sort.

Why do I choose Quick Sort? I do not like to work, of course! I want work done as soon as I can get it done.

How do I know Quick Sort is less work? I know that O(n log n) is less than O(n squared). The O's are more small, so Quick Sort is less work!

Now you know my friend, Big O. He helps us do less work. And if you know big O, you can do less work too!

You learned all that with me! You are so smart! Thank you so much!

Now that work is done, let's go play!

---

[1]: There is a way to cheat and add all the things from one to n, all at one time. Some kid named Gauss found this out when he was eight. I am not that smart though, so don't ask me how he did it.

▲

13

▼

🔖

🕘

Not sure I'm further contributing to the subject but still thought I'd share: I once found [this blog post](#) to have some quite helpful (though very basic) explanations & examples on Big O:

Via examples, this helped get the bare basics into my tortoiseshell-like skull, so I think it's a pretty descent 10-minute read to get you headed in the right direction.

Share   Improve this answer

Follow

@William ...and people tend to die of old age, species go extinct, planets turn barren etc. – Priidu Neemre May 30, 2013 at 5:52

▲

12

▼

I've more simpler way to understand the time complexity he most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N as N approaches infinity. In general you can think of it like this:

```
statement;
```

Is constant. The running time of the statement will not change in relation to N

```
for ( i = 0; i < N; i++ )
   statement;
```

Is linear. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for ( i = 0; i < N; i++ )
{
for ( j = 0; j < N; j++ )
   statement;
}
```

Is quadratic. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while ( low <= high )
{
 mid = ( low + high ) / 2;
 if ( target < list[mid] )
 high = mid - 1;
 else if ( target > list[mid] )
   low = mid + 1;
 else break;
}
```

Is logarithmic. The running time of the algorithm is proportional to the number of times N can be divided by 2. This is because the algorithm divides the working area in half with each iteration.

```
void quicksort ( int list[], int left, int right )
{
  int pivot = partition ( list, left, right );
  quicksort ( list, left, pivot - 1 );
  quicksort ( list, pivot + 1, right );
}
```

Is N * log (N). The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic. There are other Big O measures such as cubic, exponential, and square root, but they're not nearly as common. Big O notation is described as O ( ) where is the measure. The quicksort algorithm would be described as O (N * log (N)).

Note: None of this has taken into account best, average, and worst case measures. Each would have its own Big O notation. Also note that this is a VERY simplistic explanation. Big O is the most common, but it's also more complex that I've shown. There are also other notations such as big omega, little o, and big theta. You probably

won't encounter them outside of an algorithm analysis course.

- See more at: [Here](#)

Share   Improve this answer

Follow

▲

12

▼

> What is a plain English explanation of "Big O" notation?

Very Quick Note:

The O in "Big O" refers to as "Order"(or precisely "order of")
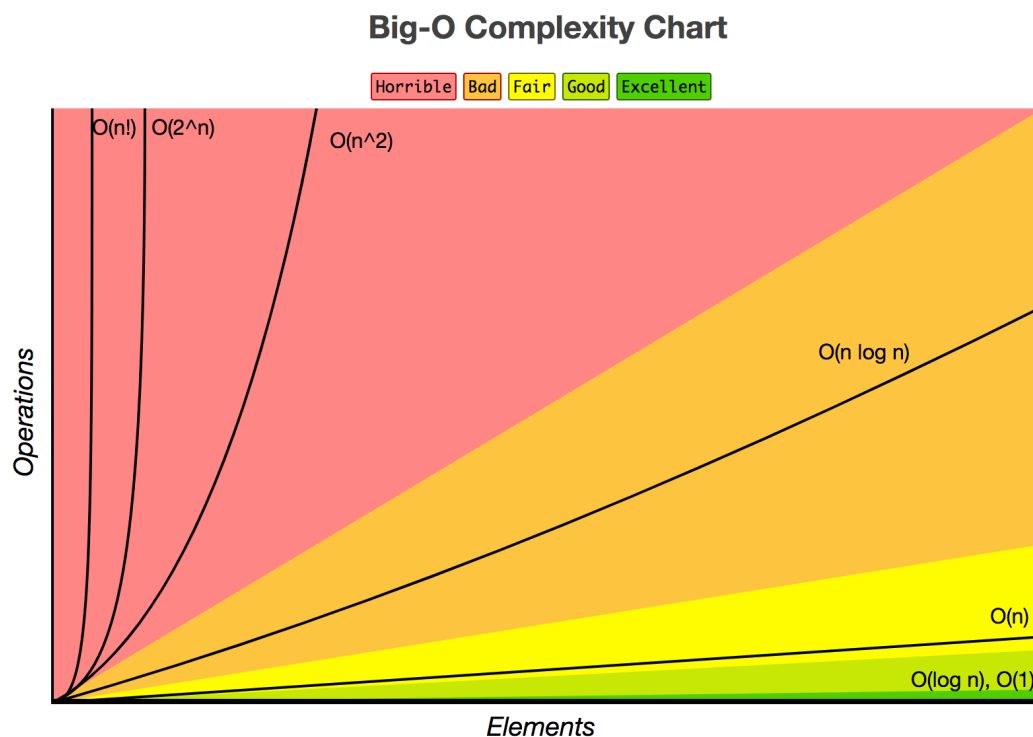so you could get its idea literally that it's used to order something to compare them.

- **"Big O" does two things:**

  1. Estimates how many steps of the method your computer applies to accomplish a task.

  2. Facilitate the process to compare with others in order to determine whether it's good or not?

  3. "Big O' achieves the above two with standardized `Notations`.

- **There are seven most used notations**

  1. O(1), means your computer gets a task done with `1` step, it's excellent, Ordered No.1

  2. O(logN), means your computer complete a task with `logN` steps, its good, Ordered No.2

  3. O(N), finish a task with `N` steps, its fair, Order No.3

  4. O(NlogN), ends a task with `O(NlogN)` steps, it's not good, Order No.4

  5. O(N^2), get a task done with `N^2` steps, it's bad, Order No.5

  6. O(2^N), get a task done with `2^N` steps, it's horrible, Order No.6

  7. O(N!), get a task done with `N!` steps, it's terrible, Order No.7

Big-O Complexity Chart

Horrible | Bad | Fair | Good | Excellent

O(n!) O(2^n)   O(n^2)

O(n log n)

O(n)

O(log n), O(1)

Operations

Elements

Suppose you get notation `O(N^2)`, not only you are clear the method takes N*N steps to accomplish a task, also you see that it's not good as `O(NlogN)` from its ranking.

Please note the order at line end, just for your better understanding.There's more than 7 notations if all possibilities considered.

In CS, the set of steps to accomplish a task is called algorithms.
In Terminology, Big O notation is used to describe the performance or complexity of an algorithm.

In addition, Big O establishes the worst-case or measure the Upper-Bound steps.
You could refer to Big-Ω (Big-Omega) for best case.

[Big-Ω (Big-Omega) notation (article) | Khan Academy](#)

- **Summary**
  "Big O" describes the algorithm's performance and evaluates it.

  or address it formally, "Big O" classifies the algorithms and standardize the comparison process.

Share   Improve this answer   edited Apr 13, 2018 at 13:53

Follow

12

Assume we're talking about an algorithm **A**, which should do something with a dataset of size **n**.

Then `O( <some expression X involving n> )` means, in simple English:

> **If you're unlucky when executing A, it might take as much as X(n) operations to complete.**

As it happens, there are certain functions (think of them as *implementations* of **X(n)**) that tend to occur quite often. These are well known and easily compared (Examples: `1`, `Log N`, `N`, `N^2`, `N!`, etc..)

By comparing these when talking about **A** and other algorithms, it is easy to rank the algorithms according to the number of operations they *may* (worst-case) require to complete.

In general, our goal will be to find or structure an algorithm **A** in such a way that it will have a function `X(n)` that returns as low a number as possible.

Share   Improve this answer

Follow

answered Oct 25, 2013 at 15:11

Kjartan
19.1k ● 16 ● 74 ● 99

If you have a suitable notion of infinity in your head, then there is a very brief description:

> Big O notation tells you the cost of solving an infinitely large problem.

And furthermore

> Constant factors are negligible

If you upgrade to a computer that can run your algorithm twice as fast, big O notation won't notice that. Constant factor improvements are too small to even be noticed in the scale that big O notation works with. Note that this is an intentional part of the design of big O notation.

Although anything "larger" than a constant factor can be detected, however.

When interested in doing computations whose size is "large" enough to be considered as approximately infinity, then big O notation is approximately the cost of solving your problem.

If the above doesn't make sense, then you don't have a compatible intuitive notion of infinity in your head, and you should probably disregard all of the above; the only way I know to make these ideas rigorous, or to explain them if they aren't already intuitively useful, is to first

teach you big O notation or something similar. (although, once you well understand big O notation in the future, it may be worthwhile to revisit these ideas)

Share   Improve this answer

Follow

Say you order Harry Potter: Complete 8-Film Collection [Blu-ray] from Amazon and download the same film collection online at the same time. You want to test which method is faster. The delivery takes almost a day to arrive and the download completed about 30 minutes earlier. Great! So it's a tight race.

What if I order several Blu-ray movies like The Lord of the Rings, Twilight, The Dark Knight Trilogy, etc. and download all the movies online at the same time? This time, the delivery still take a day to complete, but the online download takes 3 days to finish. For online shopping, the number of purchased item (input) doesn't affect the delivery time. The output is constant. We call this **O(1)**.

For online downloading, the download time is directly proportional to the movie file sizes (input). We call this **O(n)**.

From the experiments, we know that online shopping scales better than online downloading. It is very important

to understand big O notation because it helps you to analyze the **scalability** and **efficiency** of algorithms.

Note: Big O notation represents the **worst-case scenario** of an algorithm. Let's assume that **O(1)** and **O(n)** are the worst-case scenarios of the example above.

Reference : http://carlcheo.com/compsci

Share Improve this answer

Follow

answered Dec 6, 2015 at 6:01

raaz
12.5k ● 22 ● 65 ● 82

▲

10

▼

🔖

🕘

Definition :- Big O notation is a notation which says how a algorithm performance will perform if the data input increases.

When we talk about algorithms there are 3 important pillars Input , Output and Processing of algorithm. Big O is symbolic notation which says if the data input is increased in what rate will the performance vary of the algorithm processing.

I would encourage you to see this youtube video which explains Big O Notation in depth with code examples.
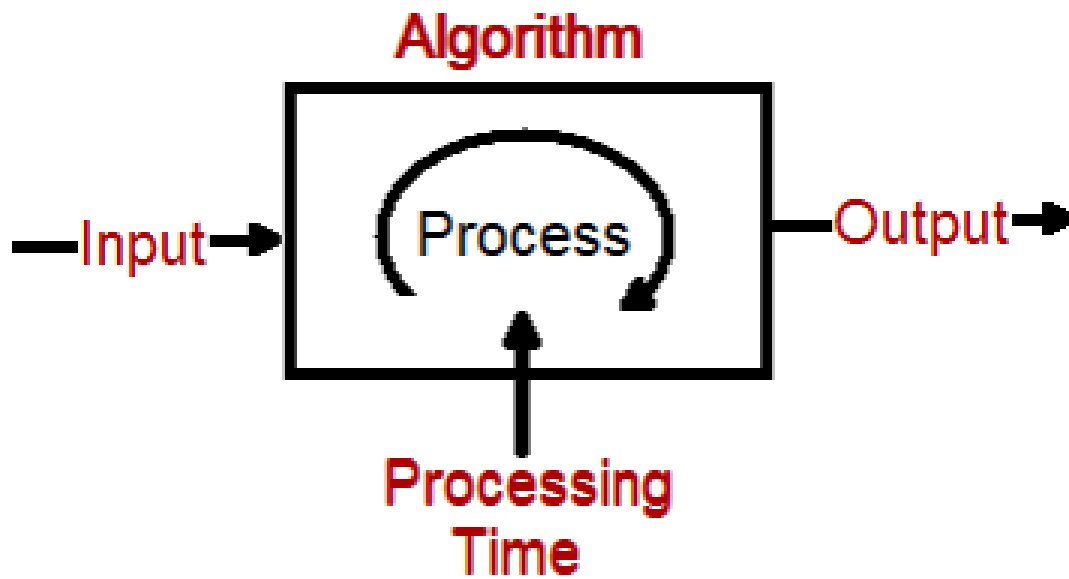
So for example assume that a algorithm takes 5 records and the time required for processing the same is 27 seconds. Now if we increase the records to 10 the algorithm takes 105 seconds.

In simple words the time taken is square of the number of records. We can denote this by **O(n ^ 2)**. This symbolic representation is termed as Big O notation.

Now please note the units can be anything in inputs it can be bytes , bits number of records , the performance can be measured in any unit like second , minutes , days and so on. So its not the exact unit but rather the relationship.

$$5 \text{ Rec} \longrightarrow 27 \text{ Sec}$$

$$10 \text{ Rec} \longrightarrow 105 \text{ Sec}$$

$$\text{Time} \cong (\text{Rec})^2$$

$$O(\text{Rec}^2)$$

$$O(n^2)$$

For example look at the below function "Function1" which takes a collection and does processing on the first record. Now for this function the performance will be same irrespective you put 1000 , 10000 or 100000 records. So we can denote it by **O(1)**.

```
void Function1(List<string> data)
{
string str = data[0];
}
```

Now see the below function "Function2()". In this case the processing time will increase with number of records. We can denote this algorithm performance using **O(n)**.

```
void Function2(List<string> data)
    {
        foreach(string str in data)
        {
            if (str == "shiv")
            {
                return;
            }
        }
    }
```

When we see a Big O notation for any algorithm we can classify them in to three categories of performance :-

1. Log and constant category :- Any developer would love to see their algorithm performance in this category.

2. Linear :- Developer will not want to see algorithms in this category , until its the last option or the only option left.

3. Exponential :- This is where we do not want to see our algorithms and a rework is needed.

So by looking at Big O notation we categorize good and bad zones for algorithms.
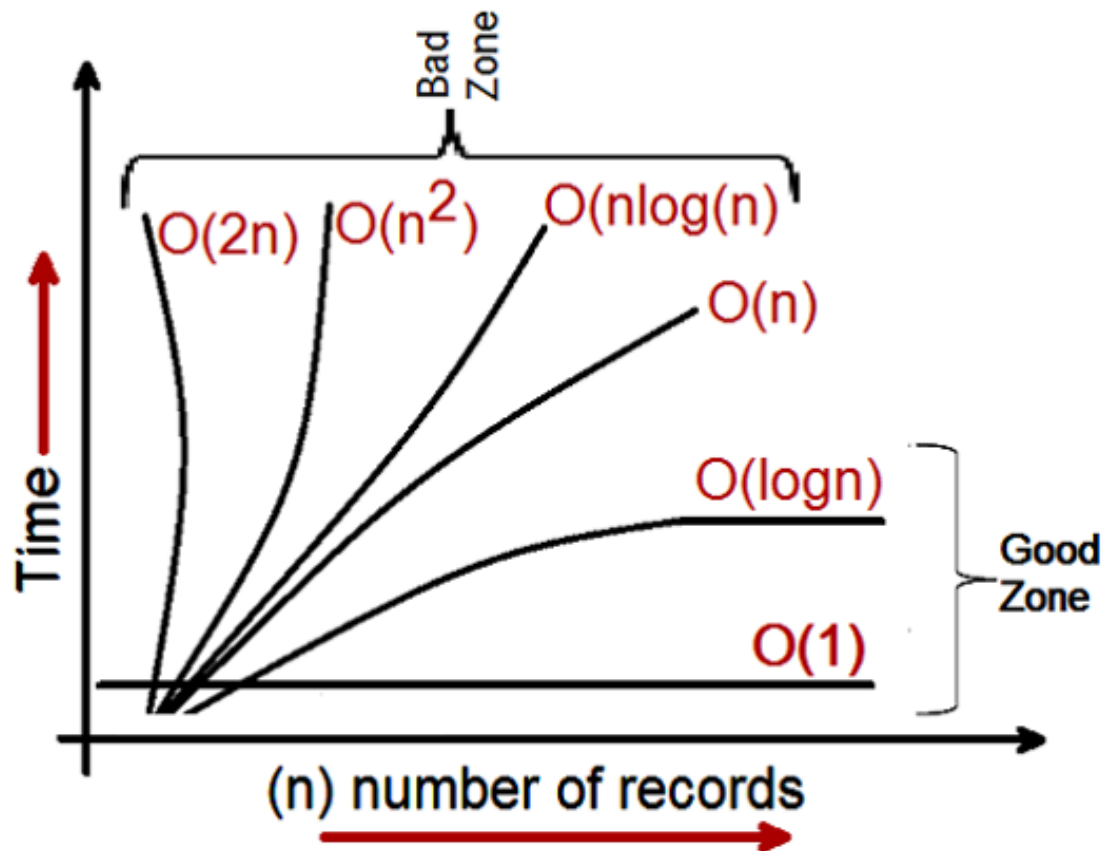
I would recommend you to watch this 10 minutes video which discusses Big O with sample code

https://www.youtube.com/watch?v=k6kxtzICG_g

Share Improve this answer   edited May 11, 2018 at 9:43

Follow

answered May 11, 2018 at 8:33

Shivprasad Koirala

28.5k ● 7 ● 87 ● 75

Simplest way to look at it (in plain English)

9

We are trying to see how the number of input parameters, affects the running time of an algorithm. If the running

time of your application is proportional to the number of input parameters, then it is said to be in Big O of n.

The above statement is a good start but not completely true.

A more accurate explanation (mathematical)

Suppose

```
n=number of input parameters

T(n)= The actual function that expresses the
running time of the algorithm as a function of n

c= a constant

f(n)= An approximate function that expresses the
running time of the algorithm as a function of n
```

Then as far as Big O is concerned, the approximation f(n) is considered good enough as long as the below condition is true.

```
lim      T(n) ≤ c×f(n)
n→∞
```

The equation is read as As n approaches infinity, T of n, is less than or equal to c times f of n.

If the above equation is true, then the running time of the algorithm is said to be in Big Oh of n

In big O notation this is written as

```
T(n)∈O(n)
```

This is read as T of n is in big O of n.

Back to English

Based on the mathematical definition above, if you say your algorithm is a Big O of n, it means it is a function of n (number of input parameters) **or faster**. If your algorithm is Big O of n, then it is also automatically the Big O of n square.

Big O of n means my algorithm runs at least as fast as this. You cannot look at Big O notation of your algorithm and say its slow. You can only say its fast.

Check [this](#) out for a video tutorial on Big O from UC Berkley. It is actually a simple concept. If you hear professor Shewchuck (aka God level teacher) explaining it, you will say "Oh that's all it is!".

Share   Improve this answer

Follow

Preface

8

algorithm: procedure/formula for solving a problem

How do analyze algorithms and how can we compare algorithms against each other?

example: you and a friend are asked to create a function to sum the numbers from 0 to N. You come up with f(x) and your friend comes up with g(x). Both functions have the same result, but a different algorithm. In order to objectively compare the efficiency of the algorithms we use **Big-O notation**.

Big-O notation: describes *how quickly runtime will grow relative to the input as the input get arbitrarily large.*

3 key takeaways:

1. Compare *how quickly runtime grows* **NOT** *compare exact runtimes* (depends on hardware)

2. Only concerned with runtime grow relative to the input **(n)**

3. As *n* gets arbitrarily large, focus on the terms that will grow the fastest as n gets large (think infinity) AKA

asymptotic analysis

Space complexity: aside from time complexity, we also care about space complexity (how much memory/space an algorithm uses). Instead of checking the time of operations, we check the size of the allocation of memory.

Share   Improve this answer

Follow

8

I found a really great explanation about big O notation especially for a someone who's not much into mathematics.

https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

> Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.
>
> Anyone who's read Programming Pearls or any other Computer Science books and doesn't have a grounding in Mathematics will have hit a wall when they reached chapters that mention O(N log N) or other seemingly crazy syntax. Hopefully

this article will help you gain an understanding of the basics of Big O and Logarithms.

As a programmer first and a mathematician second (or maybe third or fourth) I found the best way to understand Big O thoroughly was to produce some examples in code. So, below are some common orders of growth along with descriptions and examples where possible.

O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements) {
    return elements[0] == null;
}
```

O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O

notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

```csharp
bool ContainsValue(IList<string> elements, string
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```

O(N²)

$O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.

```csharp
bool ContainsDuplicates(IList<string> elements) {
    for (var outer = 0; outer < elements.Count; ou
    {
        for (var inner = 0; inner < elements.Count
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]
        }
    }
```

```
        return false;
    }
```

O(2^N)

O(2^N) denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an O(2^N) function is exponential - starting off very shallow, then rising meteorically. An example of an O(2^N) function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number) {
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number
}
```

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe

element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as O(log N). The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

Share Improve this answer

Follow

🔥 **Highly active question**. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.