Some Big-Os are Bigger Than Others

Big-O notation is often used to compare algorithms. Sergey Ignatchenko reminds us that asymptotic limits might not be generally applicable.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Often, when speaking about algorithms, we say things such as, "Hey, this algorithm is O(n^2); it is not as good as that one, which is O(n)". However, as with any bold and simple statement, there are certain very practical applicability limits, which define when the statement can be taken for granted, and where it cannot. Let's take a closer look at Big-O notation and at the real meaning behind it.

Definition

Mathematically, Big-O can be defined as follows:

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n > 0, \forall n \geq n0 : 0 \leq f(n) \leq cg(n)\}$$

In other words, $f(n) \in O(g(n))$ if and only if there exist positive constants c and $n0$ such that for all $n \geq n0$, the inequality $0 \leq g(n) \leq cg(n)$ is satisfied. We say that $f(n)$ is Big O of $g(n)$, or that $g(n)$ is an *asymptotic upper bound* for $f(n)$ [CMPS102].

In addition to O(n), there is a similar notation of $\Theta(n)$; technically, the difference between the two is that O(n) is an upper bound, and $\Theta(n)$ is a 'tight upper bound'. Actually, whenever we're speaking about Big-Os, we usually actually mean Big-Thetas.

For the purposes of this article, I prefer the following plain-English sorta-definition [Wiki.Big-O]:

1. '$T(n)$ is O(n^{100})' means $T(n)$ grows asymptotically no faster than n^{100}
2. '$T(n)$ is O(n^2)' means $T(n)$ grows asymptotically no faster than n^2
3. '$T(n)$ is $\Theta(n^2)$' means $T(n)$ grows asymptotically as fast as n^2

Note that (as O(n) is just an upper bound), all three statements above can stand for the very same function $T(n)$; moreover, whenever statement #3 stands, both #1 and #2 also stand.

In our day-to-day use of O(n), we tend to use the tightest bound, i.e. for the function which satisfies all three statements above, usually we'd say "it is O(n^2)", while actually meaning $\Theta(n^2)$.

Admittedly, the preliminaries above are short and probably insufficient if you haven't dealt with Big-Os before; if you need an introduction into the world of complexities and Big-Os, you may want to refer, for example, to [Orr14].

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He currently holds the position of Security Researcher and writes for a software blog (http://ithare.com). Sergey can be contacted at sergey@ignatchenko.com

Everything is O(1)

One interesting observation about O(n) notations is that:

> *Strictly speaking, for real-world computers, every algorithm which completes in a finite time can be said to be O(1)*

This observation follows from the very definition above. As all real-world computers have finite addressable data (2^{64}, 2^{256}, and the number of atoms in the observable universe are finite), then for any finite-time algorithm there is a finite upper bound of time MAXT (as there is only a finite number of states it can possibly go through without looping). As soon as we know this MAXT, we can say that for the purposes of our definition above, c is MAXT, and then the inequality in the definition stands, technically making ANY real-world finite-time algorithm an O(1).

It should be mentioned that this observation is more than one singular peculiarity. Instead, it demonstrates one very important property of the Big-O notation: *strictly speaking, all Big-Os apply ONLY to infinite-size input sets*. In practice, this can be relaxed, but we still need to note that

> *Big-O asymptotic makes sense ONLY for 'large enough' sets.*

Arguing about O(n) vs O(n^2) complexity for a set of size 1 is pretty pointless. But what about size 2? Size 64? Size 1048576? Or more generally:

> *What is the typical size when the difference between different Big-Os starts to dominate performance in the real world?*

Big-O notation itself is of no help in this regard (neither is it intended to be). To answer this question, we'll need to get our heads out of the mathematical sand, and deal with the much more complicated real world.

History: early CPUs

Ancient times

Historically, Big-O notation, as applied to algorithm complexity, goes back at least 50 years. At that time, most CPUs were very simple, and more importantly,

> *Memory access times were considered comparable to CPU operation times*

If we take the MIX machine [Knuth] (which assigns execution times 'typical of vintage-1970 computers'), we'll see that ADD, SUB, etc. – as well as all LOAD/STORES – are counted as 2 CPU clocks, MUL is counted as 10 CPU clocks, and the longest (besides I/O and floating-point) operation DIV is counted as 12 CPU clocks. At the same time, memory-copying MIX operation MOVE goes at the rate of two CPU clocks per word being moved. In short:

> *In 1970's computers – and in the literature of the time too – memory accesses were considered to have roughly the same CPU cost as calculations.*

As this was the time that most O(n) notation (as applied to computer science) was developed, it led to quite a number of assumptions and common practices. One of the most important assumptions (which has since become a 'silent assumption') is that we can estimate complexity by

This discrepancy in speed growth between CPU core and memory latencies has caused Very Significant changes to the CPU architecture

simply counting the number of operations. Back in ancient times, this worked pretty well; it follows both from the MIX timings above, and from data on emerging micro-CPUs such as the i8080 (which had R-R ops at 4 clocks, and R-M ops at 6 clocks). However, as CPUs were developed, the cost difference of different ops has increased A LOT, while the assumption (being a 'silent' assumption) has never really been revised.

Modern CPUs: from 1 to 100 clocks and beyond

From about the 80s till now, the situation has changed in a very significant way. Since about the early 80s, micro-CPU clocks speeds grew from ~4MHz to ~4GHz (i.e. 1000-fold), while memory latencies only improved from about 250ns to about 10ns, i.e. only 25-fold[1] ☹. This discrepancy in speed growth between CPU core and memory latencies has caused Very Significant changes to CPU architecture; in particular, LOTS of caches were deployed between the core and main RAM, to make things work more smoothly.

Currently, a typical x64 CPU has three levels of cache, with typical access times being 4, 12 and 40 clocks for L1, L2 and L3 caches respectively. Main RAM access costs 100–150 clocks (and up to 300 clocks if we're speaking about multi-socket configurations with memory on a different NUMA node). At the same time, the costs of simple ALU operations (such as ADD etc.) have dropped below 1 CPU clock.[2]

This is a Big Fat Contrast with the assumptions used back in the 1970s; now the difference because of unfortunate 'jumps' over (uncached at the time) memory can lead to a 100×+ (!) performance difference. However, it is still O(1) and is rarely taken into account during performance analysis ☹.

Real world experiment

Theory is all very well, but what about a little practical experiment to support it? Let's take a look at a short and simple program (Listing 1).

Trivial, right? Now let's see how this program performs in two cases: first, when run 'as is', and second, when we replace `list<int>` in line `(*)` with `vector<int>`.

As we can see from Listing 1, between moments t0 and t1 we're only traversing our lists or vectors, and each inner iteration is O(1). Therefore, the whole thing between t1 and t0 is clearly O(N*M*P) = O(n) – both for list and vector. Now, let's see what is the *real-world* difference between these two algorithms with exactly the same Big-O asymptotic.

WARNING: VIEWER DISCRETION ADVISED. The results below can be unsuitable for some of the readers, and are intended for mature developer audiences only. Author, translator, and Overload *disclaim all the responsibility for all the effects resulting from reading further, including, but not limited to: broken jaws due to jaw dropping, popped eyes, and being bored to death because it is well-known to the reader.* (See Table 1.)

```
constexpr int N=100;
constexpr int M=5000;
constexpr int P=5000;

using Container = list<int>;//(*)

int main() {
  //creating containers
  Container c[M];
  srand(time(0));
  for(int i=0;i<M;++i) {
    for(int j=0;j<P;++j) {
      Container& cc = c[rand()%M];
      cc.push_back(rand());
    }
  }

  clock_t t0 = clock();
  //running over them N times
  int sum = 0;
  for(int i=0;i<N;++i) {
    for(int j=0;j<M;++j) {
      for(int it: c[j]) {
        sum += it;
      }
    }
  }

  clock_t t1 = clock();
  cout << sum << '\n';
  cout << "t=" << (t1-t0) << '\n';
  return 0;
}
```

Listing 1

As we can see, Big-O(n) for `list<>` *has been observed to be up to 780× bigger than intuitively the same Big-O(n) for* `vector<>`*.*

And that was NOT a specially constructed case of swapping or something – it was just an in-memory performance difference, pretty much without context switching or any other special scenarios; in short – this can easily happen in a pretty much any computing environment. *BTW, you should be able to reproduce similar results yourself on your own box, please just don't forget to use at least some optimization, as debug overhead tends to mask the performance difference; also note that drastically reducing M and/or P will lead to different cache patterns, with results being very different.*

In trying to explain this difference, we'll need to get into educated-guesswork area. For MSVC and gcc, the performance difference between `vector<>` and `list<>` is pretty much in line with the difference between typical cached access times (single-digit clocks) and typical uncached access times (100–150 clocks). As access patterns for

1. I don't want to get into discussion whether it's really 10× or 100× – in any case, it is MUCH less than 1000×.
2. Statistically, of course.

any evidence we've got represents only a small subset of all the possible experiments

Box 1	clang -O2 N=100,M=5000,P=5000 RESULT: **vector<> is 434x faster**	clang -O3 -march=native N=100,M=5000,P=5000 RESULT: **vector<> is 498x faster**	clang -O3 -march=native N=100,M=1000,P=1000 RESULT: **vector<> is 780x faster**[a]
Box 2	MSVC Release N=100,M=1000,P=1000 RESULT: **vector<> is 116x faster**	MSVC Release N=100,M=5000,P=5000 RESULT: **vector<> is 147x faster**	gcc -O2 N=100,M=1000,P=1000 RESULT: **vector<> is 120x faster**

a. This result is probably attributed to **`vector<>`** with M=1000 and P=1000 fitting into L3 cache on this box.

Table 1

`vector<>` are expected to use CPU prefetch fully and **`list<>`** under the patterns in Listing 1 is pretty much about random access to memory, which cannot be cached due to the size, this 100–150× difference in access times can be expected to translate into 100–150× difference in performance.

For clang, however, the extra gain observed is not that obvious. My somewhat-educated guess here is that clang manages to get more from parallelization over linear-accessible **`vector<>`**, while this optimization is inapplicable to **`list<>`**. In short – when going along the **`vector<>`**, the compiler and/or CPU 'know' where exactly the next **`int`** resides, so they can fetch it in advance, and can process it in parallel too. When going along the **`list<>`**, it is NOT possible to fetch the next item until the pointer is dereferenced (and under the circumstances, it takes a looooong while to dereference this pointer).

On the other hand, it should be noted that an exact explanation of the performance difference is not that important for the purposes of this article. The only thing which matters is that we've taken two 'reasonably good' (i.e. NOT deliberately poorly implemented) algorithms, both having exactly the same Big-O asymptotic, and easily got 100×-to-780× performance difference between the two.

Potential difference: 1000x as a starting point

As we can see, depending on the compiler, results above vary greatly; however, what is clear, is that

> *These days, real-world difference between two algorithms with exactly the same Big-O asymptotic behaviour, can easily exceed 500×*

In other words: we've got very practical evidence that the difference can be over 500×. On the other hand, any evidence we've got represents only a small subset of all the possible experiments. Still, lacking any further evidence at the moment, the following assumption looks fairly reasonable.

> *With modern CPUs, if we have two algorithms (neither of which being specially anti-optimized, and both intuitively perceived to have the same performance at least by some developers), the performance difference between them can be anywhere from 0.001× to 1000×.*

Consequences

Now let's take a looking at the same thing from a bit different perspective. The statement above can be rephrased into the following:

> *with modern CPUs, unknown constants (those traditionally ignored in Big-O notation) may easily reach 1000.*

In turn, this means that while O(n2) is still worse than O(n) *for large values of n*, for n's around 1000 or so we can easily end up in a situation when:

- algo1() is $O(n^2)$, but T(algo1(n)) is actually $1* n^2$
- algo2() is O(n), but T(algo2(n)) is actually 1000*n
- then $\forall$ n < 1000, $T(algo1(n)) = n^2 < 1000*n = T(algo2(n))$

This observation is nothing new, and it was obviously obvious to the Founding Fathers back in the 1970s; what's new is the *real-world difference* in those constants, which has grown Very Significantly since that time, and can now reach 1000×. In other words,

> *the maximum size when $O(n^2)$ can be potentially faster than O(n), has grown to a very significant n~=1000*

If we're comparing two algos, one with complexity of O(n) and another with complexity of O(log n), then similar analysis will look as follows:

- algo1() is O(n), but T(algo1(n)) is actually $1* n$
- algo2() is O(log n), but T(algo2(n)) is actually $1000*\log_2(n)$
- then $\forall$ n < 13746, $T(algo1(n)) = n < 1000*\log_2(n) = T(algo2(n))$

In other words,

> *the maximum size when O(n) can be potentially faster than O(log n), is currently even larger, in the over-10K range*

Summary

> *All animals are equal, but some animals are more equal than others.*
> ~ George Orwell, Animal Farm

To summarize the findings above:

- Big-O notation still stands ;-)
  - All Big-Os are Big, but some Big-Os are bigger than others

If in doubt – test it! Real-world results can be very different from intuitive expectations

- On modern CPUs, the performance difference between two reasonably good algos with the same Big-O can easily reach over 500× (that's for usual scenarios, without swapping or context switch thrashing)

- Big-O asymptotic can still be used for *large enough* values of *n*. However, what qualifies as *large enough* for this purpose, has changed over the years
  - In particular, for modern CPUs, even if the asymptotic difference is 'large' (such as comparing $O(n^2)$ to O(n)), then the advantage of O(n) SHOULD NOT be taken as granted for sizes < 1000 or so
  - If the asymptotic difference is 'smaller' (such as comparing O(n) to O(log n)), then the advantage of O(log n) SHOULD NOT be taken as granted for sizes < 10000 or so
  - Starting from n=10000, we can usually still expect that the better Big-O asymptotic will dominate performance in practice
  - If in doubt – test it! Real-world results can be Very Different from intuitive expectations (and still Very Different from estimates based on pure Big-O asymptotic).

References

[CMPS102] CMPS 102, 'Introduction to Analysis of Algorithms', University of California, https://classes.soe.ucsc.edu/cmps102/Spring04/TantaloAsymp.pdf

[Knuth] Donald E. Knuth, *The Art of Computer Programming*, Vol.1, section 1.3.1

[Loganberry04] David 'Loganberry' Buttery, 'Frithaes! – an Introduction to Colloquial Lapine', http://bitsnbobstones.watershipdown.org/lapine/overview.html

[Orr14] Roger Orr, 'Order Notation in Practice', *Overload* #124

[Wiki.Big-O] https://en.wikipedia.org/wiki/Big_O_notation

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.