DOI:10.1145/2980976

Article development led by acmqueue
queue.acm.org

The operations side of the story.

BY THOMAS A. LIMONCELLI

10 Optimizations on Linear Search

A FRIEND WAS asked the following question during a job interview: What is the fastest algorithm to find the largest number in an unsorted array?

The catch, of course, is that the data is unsorted. Because of that, each item must be examined; thus, the best algorithm would require $O(N)$ comparisons, where N is the number of elements. Any computer scientist knows this. For that reason, the fastest algorithm will be a linear search through the list.

End of story.

All the computer scientists may leave the room now.

Are all the computer scientists gone?

Good!

Now let's talk about the operational answer to this question.

System administrators (DevOps engineers or SREs or whatever your title) must deal with the operational aspects of computation, not just the theoretical aspects. Operations is where the rubber hits the road. As a result, operations people see things from a different perspective and can realize opportunities outside of the basic $O()$ analysis.

Let's look at the operational aspects of the problem of trying to improve something that is theoretically optimal already.

1. Don't Optimize Code that is Fast Enough

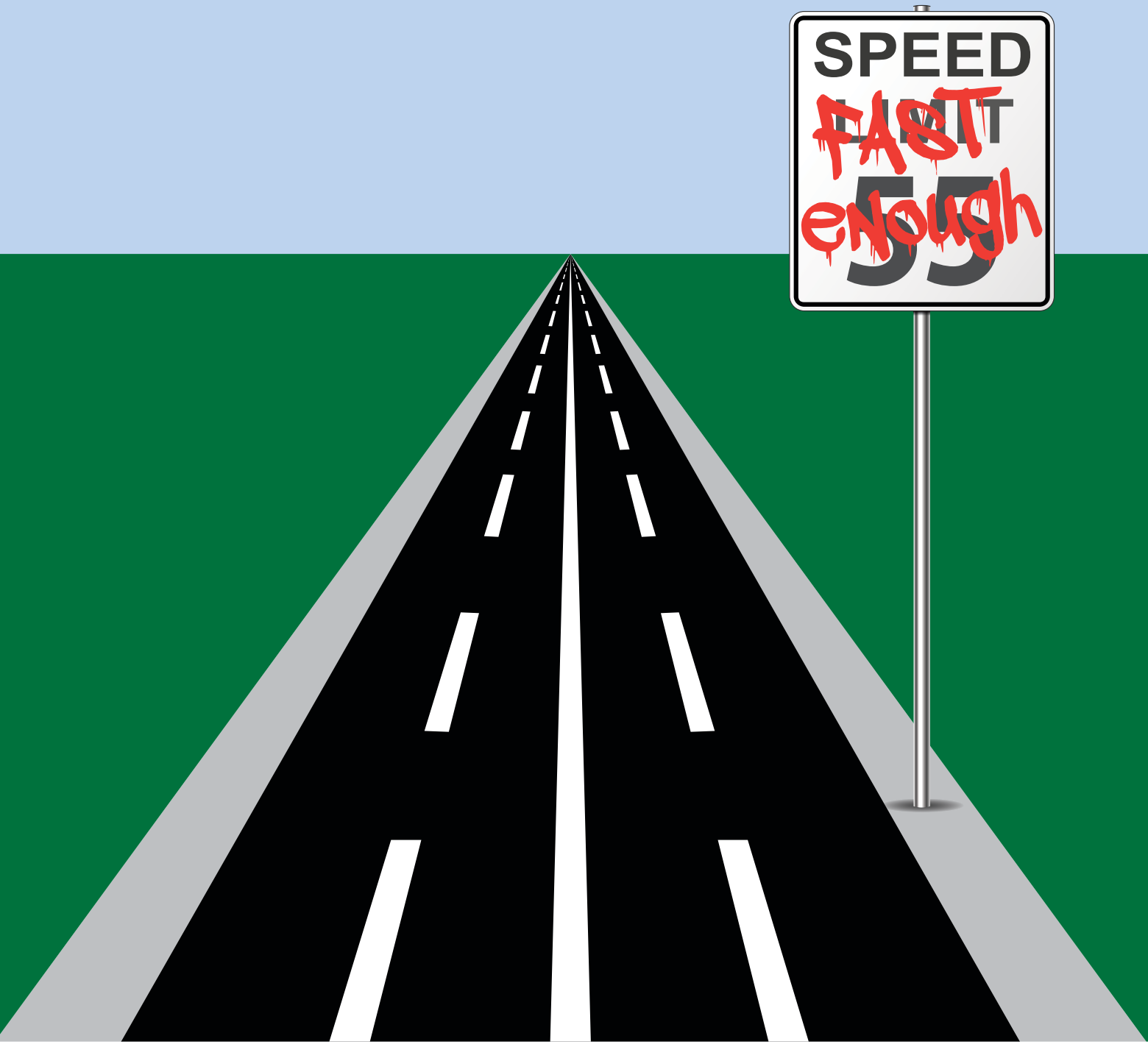
The first optimization comes from deciding to optimize time and not the algorithm itself. First, ask whether the code is fast enough already. If it is, you can optimize your time by not optimizing this code at all. This requires a definition of *fast enough*.

Suppose 200ms and under is *fast enough*. Anything that takes less than 200ms is perceived to be instantaneous by the human brain. Therefore, any algorithm that can complete the task in less than 200ms is usually good enough for interactive software.

Donald Knuth famously wrote that premature optimization is the root of all evil. Optimized solutions are usually more complex than the solutions they replace; therefore, you risk introducing bugs into the system. A bird in hand is worth two in the bush. Why add complexity when you don't have to?

My biggest concern with premature optimization is it is a distraction from other, more important work. Your time is precious and finite. Time spent on a premature optimization is time that could be spent on more important work.

Prioritizing your work is not about deciding in what order you will do the items on your to-do list. Rather, it is deciding which items on your to-do list will be intentionally dropped on the floor. I have 100 things I would like to do this week. I am going to complete only about 10 of them. How I prioritize my work determines which 90 tasks won't get done. I repeat this process ev-



IMAGERY FROM SHUTTERSTOCK.COM

ery week. One of the best time-management skills you can develop is to learn to let go of that 90%.

In the case of the interview question, whether optimizing is worthwhile relates to the number of data items. It isn't worth optimizing if only a small amount of data is involved. I imagine that if, during the interview, my friend had asked, "How many elements in the list?" the interviewer would have told him that it doesn't matter. From a theoretical point of view, it doesn't; from an operational point of view, however, it makes *all* the difference.

Deciding if an optimization is worth your time requires a quick back-of-the-envelope estimate to determine what kinds of improvements are possible, how long they might take to be achieved, and if the optimization will result in a return on investment. The ability to use rough estimates to decide whether or not an engineering task is worthwhile may be one of the most important tools in a system administrator's toolbox.

If *small* is defined to mean any amount of data that can be processed in under 200ms, then you would be surprised at how big *small* can be.

I conducted some simple benchmarks in Go to find how much data can be processed in 200ms. A linear search can scan 13 million elements in less than 200ms on a three-year-old MacBook laptop, and 13 million is no small feat.

This linear search might be buggy, however. It is five lines long and, not to

brag, but I can pack a lot of bugs into five lines. What if I were to leverage code that has been heavily tested instead? Most languages have a built-in sort function that has been tested far more than any code I've ever written. I could find the max by sorting the list and picking the last element. That would be lazy and execute more slowly than a linear search, but it would be very reliable. A few simple benchmarks found that on the same old laptop, this "lazy algorithm" could sort 700,000 elements and still be under the 200ms mark.

What about smaller values of *N*?

If *N* = 16,000, then the entire dataset fits in the L1 cache of the CPU, assuming the CPU was made in this decade. This means the CPU can scan the data so fast it will make your hair flip. If *N* = 64,000, then the data will fit in a modern L2 cache, and your hair may still do interesting things. If the computer wasn't made in this decade, I would recommend that my friend reconsider working for this company.

If *N* is less than 100, then the lazy algorithm runs imperceptibly fast. In fact, you could repeat the search on demand rather than storing the value, and unless you were running the algorithm thousands of times, the perceived time would be negligible.

The algorithms mentioned so far are satisfactory until *N* = 700,000 if we are lazy and *N* = 13,000,000 if we aren't; 13 million 32-bit integers (about 52MB) is hardly small by some standards. Yet, in terms of human perception, it can be searched instantly.

If my friend had known these benchmark numbers, he could have had some fun during the interview, asking the interviewer to suggest a large value of *N*, and replying, "What? I don't get out of bed for less than 13 million integers!" (Of course, this would probably have cost him the job.)

2. Use SIMD Instructions

Most modern CPUs have SIMD (single instruction, multiple data) instructions that let you repeat the same operation over a large swath of memory. They are able to do this very quickly because they benefit from more efficient memory access and parallel operations.

According to one simple benchmark (http://stackoverflow.com/a/2743040/71978), a 2.67GHz Core i7 saw a 7–8x improvement by using SIMD instructions where *N* = 100,000. If the amount of data exceeded the CPU's cache size, the benefit dropped to 3.5x.

With SIMD, *small* becomes about 45 million elements, or about 180MB.

3. Work in Parallel

Even if *N* is larger than the *small* quantity, you can keep within your 200ms time budget by using multiple CPUs. Each CPU core can search a shard of the data. With four CPU cores, *small* becomes 4*N*, or nearly 200 million items.

When I was in college, the study of parallel programming was hypothetical because we didn't have access to computers with more than one CPU. In fact, I didn't think I would ever be lucky enough to access a machine with such a fancy architecture. Boy, was I wrong! Now I have a phone with eight CPU cores, one of which, I believe, is dedicated exclusively to crushing candy.

Parallel processing is now the norm, not the exception. Code should be written to take advantage of this.

4. Hide Calculation in Another Function

The search for the max value can be hidden in other work. For example, earlier in the process the data is loaded into memory. Why not have that code also track the max value as it iterates through the data? If the data is being loaded from disk, the time spent waiting for I/O will dominate, and the additional comparison will be, essentially, free.

If the data is being read from a text file, the work to convert ASCII digits to 32-bit integers is considerably more than tracking the largest value seen so far. Adding max-value tracking would be "error in the noise" of any benchmarks. Therefore, it is essentially free.

You might point out that this violates the SoC (separation of concerns) principle. The method that loads data from the file should just load data from a file. Nothing else. Having it also track the maximum value along the way adds complexity. True, but we've already decided the added complexity is worth the benefit.

Where will this end? If the `LoadDataFromFile()` method also calculates the max value, what's to stop us from adding other calculations? Should it also calculate the min, count, total, and average? Obviously not. If you have the count and total, then you can calculate the average yourself.

5. Maintain the Max Along the Way

What if the max value cannot be tracked as part of loading the dataset? Perhaps you don't control the method that loads the data. If you are using an off-the-shelf JSON (JavaScript Object Notation) parser, adding the ability to track the max value would be very difficult. Perhaps the data is modified after being loaded, or it is generated in place.

In such situations I would ask why the data structure holding the data isn't doing the tracking itself. If data is only added, never removed or changed, the data structure can easily track the largest value seen so far. The need for a linear search has been avoided altogether.

If items are being removed and changed, more sophisticated data structures are required. A heap makes the highest value accessible in O(1) time. The data can be kept in the original order but in a heap or other index on the side. You will then always have fast access to the highest value, though you will suffer from additional overhead maintaining the indexes.

6. Hide Long Calculations from Users

Maybe the process can't be made any faster, but the delay can be hidden from the user.

One good place to hide the calculation is when waiting for user input. You don't need the entire processing power of the computer to ask "Are you sure?" and then wait for a response. Instead, you can use that time to perform calculations, and no one will be the wiser.

One video-game console manufacturer requires games to have some kind of user interaction within a few seconds of starting. Sadly, most games need more time than that to load and initialize. To meet the vendor's requirement, most games first load and display a title screen, then ask users to click a button to start the game. What users don't realize is that while they are sitting in awe of the amazing title screen, the game is finishing its preparations.

Get Out of Your Silo

Before discussing the remaining optimizations, let's discuss the value of thinking more globally about the problem. Many optimizations come from end-to-end thinking. Rather than optimizing the code itself, we should look at the entire system for inspiration.

To do this requires something scary: talking to people. Now, I understand that a lot of us go into this business because we like machines more than people, but the reality is that operations is a team sport.

Sadly, often the operations team is put in a silo, expected to work issues out on their own without the benefit of talking to the people who created the system. This stems from the days when one company created software and sold it on floppy disks. The operations people were in a different silo from the developers because they were literally in a different company. System administrators' only access to developers at the other company was through customer support, whose job it was to insulate developers from talking to customers directly. If that ever did happen, it was called an escalation, an industry term that means that a customer accidentally got the support he or she paid for. It is something that the software industry tries to prevent at all costs.

Most (or at least a growing proportion of) IT operations, however, deal with software that is developed in-house. In that situation there is very little excuse to have developers and operations in separate silos. In fact, they should talk to each other and collaborate. There should be a name for this kind of collaboration between developers and operations ... and there is: DevOps.

If your developers and operations teams are still siloed away from each other, then your business model hasn't changed since software was sold on floppy disks. This is ironic since your company probably didn't exist when floppy disks were in use. What's wrong with this picture?

Get out of your silo and talk to people. Take a walk down the hallway and introduce yourself to the developers in your company. Have lunch with them. Indulge in your favorite after-work beverage together. If you are a manager who requires operations and developers to communicate only through "proper channels" involving committees and product management chains, get out of their way.

Many optimizations come from end-to-end thinking. Rather than optimizing the code itself, we should look at the entire system for inspiration.

Once operations has forged a relationship with developers, it is easier to ask important questions, such as: How is the data used? What is it needed for and why?

This kind of social collaboration is required to develop the end-to-end thinking that makes it possible to optimize code, processes, and organizations. Every system has a bottleneck. If you optimize upstream of the bottleneck, you are simply increasing the size of the backlog waiting at the bottleneck. If you optimize downstream of the bottleneck, you are adding capacity to part of a system that is starved for work. If you stay within your silo, you'll never know enough to identify the actual bottleneck.

Getting out of your silo opens the door to optimizations such as our last four examples.

7. Use a "Good Enough" Value Instead

Is the maximum value specifically needed, or is an estimate good enough?

Perhaps the calculation can be avoided entirely.

Often an estimate is sufficient, and there are many creative ways to calculate one. Perhaps the max value from the previous dataset is good enough.

Perhaps the max value is being used to preallocate memory or other resources. Does this process really need to be fine-tuned every time the program runs? Might it be sufficient to adjust the allocations only occasionally—perhaps in response to resource monitoring or performance statistics?

If you are dealing with a *small* amount of data (using the earlier definition of small), perhaps preallocating resources is overkill. If you are dealing with large amounts of data, perhaps preallocating resources is unsustainable and needs to be reengineered before it becomes dangerous.

8. Seek Inspiration from the Upstream Processes

Sometimes we can get a different perspective by examining the inputs.

Where is the data coming from?

I once observed a situation where a developer was complaining that an

operation was very slow. His solution was to demand a faster machine. The sysadmin who investigated the issue found that the code was downloading millions of data points from a database on another continent. The network between the two hosts was very slow. A faster computer would not improve performance.

The solution, however, was not to build a faster network, either. Instead, we moved the calculation to be closer to the data. Rather than download the data and do the calculation, the sysadmin recommended changing the SQL query to perform the calculation at the database server. Instead of downloading millions of data points, now we were downloading the single answer.

This solution seems obvious but eluded the otherwise smart developer. How did that happen? Originally, the data was downloaded because it was processed and manipulated many different ways for many different purposes. Over time, however, these other purposes were eliminated until only one purpose remained. In this case the issue was not calculating the max value, but simply counting the number of data points, which SQL is very good at doing for you.

9. Seek Inspiration from The Downstream Processes

Another solution is to look at what is done with the data later in the process. Does some other processing step sort the data? If so, the max value doesn't need to be calculated. You can simply sort the data earlier in the process and take the last value.

You wouldn't know this was possible unless you took the time to talk with people and understand the end-to-end flow of the system.

Once I was on a project where data flowed through five different stages, controlled by five different teams. Each stage took the original data and sorted it. The data didn't change between stages, but each team made a private copy of the entire dataset so they could sort it. Because they had not looked outside their silos, they didn't realize how much wasted effort this entailed.

By sorting the data earlier in the flow, the entire process became much faster. One sort is faster than five.

10. Question the Question

When preparing this column I walked around the New York office of Stack Overflow and asked my coworkers if they had ever been in a situation where calculating the max value was a bottleneck worth optimizing.

The answer I got was a resounding no.

One developer pointed out that calculating the max is usually something done infrequently, often once per program run. Optimization effort should be spent on tasks done many times.

A developer with a statistics background stated that the max is useless. For most datasets it is an outlier and should be ignored. What *are* useful to him are the top *N* items, which presents an entirely different algorithmic challenge.

Another developer pointed out that anyone dealing with large amounts of data usually stores it in a database, and databases can find the max value very efficiently. In fact, he asserted, maintaining such data in a homegrown system is a waste of effort at best and negligent at worst. Thinking you can maintain a large dataset safely with homegrown databases is hubris.

Most database systems can determine the max value very quickly because of the indexes they maintain. If the system cannot, it isn't the system administrator's responsibility to rewrite the database software, but to understand the situation well enough to facilitate a discussion among the developers, vendors, and whoever else is required to find a better solution.

Conclusion: Find Another Question

This brings me to my final point. Maybe the interview question posed at the beginning of this column should be retired. It might be a good logic problem for a beginning programmer, but it is not a good question to use when interviewing system administrators because it is not a realistic situation.

A better question would be to ask job candidates to describe a situation where they optimized an algorithm. You can then listen to their story for signs of operational brilliance.

I would like to know that the candidates determined ahead of time what would be considered good enough. Did they talk with stakeholders to determine whether the improvement was needed, how much improvement was needed, and how they would know if the optimization was achieved? Did they determine how much time and money were worth expending on the optimization? Optimizations that require an infinite budget are not nearly as useful as one would think.

I would look to see if they benchmarked the system before and after, not just one or the other or not at all. I would like to see that they identified a specific problem, rather than just randomly tuning parts until they got better results. I would like to see that they determined the theoretical optimum as a yardstick against which all results were measured.

I would pay careful attention to the size of the improvement. Was the improvement measured, or did it simply "feel faster?" Did the candidates enhance performance greatly or just squeeze a few additional percentage points out of the existing system? I would be impressed if they researched academic papers to find better algorithms.

I would be most impressed, however, if they looked at the bigger picture and found a way to avoid doing the calculation entirely. In operations, often the best improvements come not from adding complexity, but by eliminating processes altogether. C

Related articles on queue.acm.org

You're Doing It Wrong
Poul-Henning Kamp
http://queue.acm.org/detail.cfm?id=1814327

You Don't Know Jack about Network Performance
Kevin Fall and Steve McCanne
http://queue.acm.org/detail.cfm?id=1066069

Why Writing Your Own Search Engine is Hard
Anna Patterson
http://queue.acm.org/detail.cfm?id=988407

Thomas A. Limoncelli is a site reliability engineer at Stack Overflow Inc. in New York City. His books include *The Practice of Cloud Administration* (http://the-cloud-book.com), *The Practice of System and Network Administration* (http://the-sysadmin-book.com), and *Time Management for System Administrators*. He blogs at EverythingSysadmin.com and tweets at @YesThatTom.