go-perfbook/performance.md at master · dgryski/go-perfbook

Writing and Optimizing Go code

This document outlines best practices for writing high-performance Go code.

While some discussions will be made for making individual services faster (caching, etc), designing performant distributed systems is beyond the scope of this work. There are already good texts on monitoring and distributed system design. Optimizing distributed systems encompasses an entirely different set of research and design trade-offs.

This book is split into different sections:

Basic tips for writing not-slow software
CS 101-level stuff

Tips for writing fast software
Go-specific sections on how to get the best from Go

Advanced tips for writing *really* fast software
For when your optimized code isn't fast enough

We can summarize these three sections as:

"Be reasonable"

"Be deliberate"

"Be dangerous"

When and Where to Optimize

I'm putting this first because it's really the most important step. Should you even be doing this at all?

Every optimization has a cost. Generally, this cost is expressed in terms of code complexity or cognitive load -- optimized code is rarely simpler than the unoptimized version.

But there's another side that I'll call the economics of optimization. As a programmer, your time is valuable. There's the opportunity cost of what else you could be working on for your project, which bugs to fix, which features to add. Optimizing things is fun, but it's not always the right task to choose. Performance is a feature, but so is shipping, and so is correctness.

Choose the most important thing to work on. Sometimes it's not an actual CPU optimization, but a user-experience one. Something as simple as adding a progress bar, or making a page more responsive by doing computation in the background after rendering the page.

Sometimes this will be obvious: an hourly report that completes in three hours is probably less useful than one that completes in less than one.

Just because something is easy to optimize doesn't mean it's worth optimizing. Ignoring low-hanging fruit is a valid development strategy.

Think of this as optimizing *your* time.

You get to choose what to optimize and when to optimize. You can move the slider between "Fast Software" and "Fast Deployment"

People hear and mindlessly repeat "premature optimization is the root of all evil", but they miss the full context of the quote.

> "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%."
>
> -- Knuth

Add: https://www.youtube.com/watch?time_continue=429&v=3WBaY61c9sE

don't ignore the easy optimizations

more knowledge of algorithms and data structures makes more optimizations "easy" or "obvious"

Should you optimize?

> Yes, but only if the problem is important, the program is genuinely too slow, and there is some expectation that it can be made faster while maintaining correctness, robustness, and clarity."
>
> -- The Practice of Programming, Kernighan and Pike

Premature optimization can also hurt you by tying you into certain decisions. The optimized code can be harder to modify if requirements change and harder to throw away (sunk-cost fallacy) if needed.

BitFunnel performance estimation has some numbers that make this trade-off explicit. Imagine a hypothetical search engine needing 30,000 machines across multiple data centers. These machines have a cost of approximately $1,000 USD per year. If you can double the speed of the software, this can save the company $15M USD per year. Even a single developer spending an entire year to improve performance by only 1% will pay for itself.

In the vast majority of cases, the size and speed of a program is not a concern. The easiest optimization is not having to do it. The second easiest optimization is just buying faster hardware.

Once you've decided you're going to change your program, keep reading.

How to Optimize

Optimization Workflow

Before we get into the specifics, let's talk about the general process of optimization.

Optimization is a form of refactoring. But each step, rather than improving some aspect of the source code (code duplication, clarity, etc), improves some aspect of the performance: lower CPU, memory usage, latency, etc. This improvement generally comes at the cost of readability. This means that in addition to a comprehensive set of unit tests (to ensure your changes haven't broken anything), you also need a good set of benchmarks to ensure your changes are having the desired effect on performance. You must be able to verify that your change really *is* lowering CPU. Sometimes a change you thought would improve performance will actually turn out to have a zero or negative change. Always make sure you undo your fix in these cases.

What is the best comment in source code you have ever encountered? - Stack Overflow:

```
//
// Dear maintainer:
//
// Once you are done trying to 'optimize' this routine,
// and have realized what a terrible mistake that was,
// please increment the following counter as a warning
// to the next guy:
//
// total_hours_wasted_here = 42
//
```

The benchmarks you are using must be correct and provide reproducible numbers on representative workloads. If individual runs have too high a variance, it will make small improvements more difficult to spot. You will need to use benchstat or equivalent statistical tests and won't be able just to eyeball it. (Note that using statistical tests is a good idea anyway.) The steps to run the benchmarks should be documented, and any custom scripts and tooling should be committed to the repository with instructions for how to run them. Be mindful of large benchmark suites that take a long time to run: it will make the development iterations slower.

Note also that anything that can be measured can be optimized. Make sure you're measuring the right thing.

The next step is to decide what you are optimizing for. If the goal is to improve CPU, what is an acceptable

speed? Do you want to improve the current performance by 2x? 10x? Can you state it as "a problem of size N in less than time T"? Are you trying to reduce memory usage? By how much? How much slower is acceptable for what change in memory usage? What are you willing to give up in exchange for lower space requirements?

Optimizing for service latency is a trickier proposition. Entire books have been written on how to performance test web servers. The primary issue is that for a single function, performance is fairly consistent for a given problem size. For webservices, you don't have a single number. A proper web-service benchmark suite will provide a latency distribution for a given reqs/second level. This talk gives a good overview of some of the issues: "How NOT to Measure Latency" by Gil Tene

TODO: See the later section on optimizing web services

The performance goals must be specific. You will (almost) always be able to make something faster. Optimizing is frequently a game of diminishing returns. You need to know when to stop. How much effort are you going to put into getting the last little bit of work. How much uglier and harder to maintain are you willing to make the code?

Dan Luu's previously mentioned talk on BitFunnel performance estimation shows an example of using rough calculations to determine if your target performance figures are reasonable.

Simon Eskildsen has a talk from SRECon covering this topic in more depth: Advanced Napkin Math: Estimating System Performance from First Principles

Finally, Jon Bentley's "Programming Pearls" has a chapter titled "The Back of the Envelope" covering Fermi problems. Sadly, these kinds of estimation skills got a bad wrap thanks to their use in Microsoft style "puzzle interview questions" in the 1990s and early 2000s.

For greenfield development, you shouldn't leave all benchmarking and performance numbers until the end. It's easy to say "we'll fix it later", but if performance is really important it will be a design consideration from the start. Any significant architectural changes required to fix performance issues will be too risky near the deadline. Note that *during* development, the focus should be on reasonable program design, algorithms, and data structures. Optimizing at lower-levels of the stack should wait until later in the development cycle when a more complete view of the system performance is available. Any full-system profiles you do while the system is incomplete will give a skewed view of where the bottlenecks will be in the finished system.

TODO: How to avoid/detect "Death by 1000 cuts" from poorly written software. Solution: "Premature pessimization is the root of all evil". This matches with my Rule 1: Be deliberate. You don't need to write every line of code to be fast, but neither should by default do wasteful things.

> "Premature pessimization is when you write code that is slower than it needs to be, usually by asking for unnecessary extra work, when equivalently complex code would be faster and should just naturally flow out of your fingers."

-- Herb Sutter

Benchmarking as part of CI is hard due to noisy neighbours and even different CI boxes if it's just you. Hard to gate on performance metrics. A good middle ground is to have benchmarks run by the developer (on appropriate hardware) and included in the commit message for commits that specifically address performance. For those that are just general patches, try to catch performance degradations "by eye" in code review.

TODO: how to track performance over time?

Write code that you can benchmark. Profiling you can do on larger systems. Benchmarking you want to test isolated pieces. You need to be able to extract and setup sufficient context that benchmarks test enough and are representative.

The difference between what your target is and the current performance will also give you an idea of where to start. If you need only a 10-20% performance improvement, you can probably get that with some implementation tweaks and smaller fixes. If you need a factor of 10x or more, then just replacing a multiplication with a left-shift isn't going to cut it. That's probably going to call for changes up and down your stack, possibly redesigning large portions of the system with these performance goals in mind.

Good performance work requires knowledge at many different levels, from system design, networking, hardware (CPU, caches, storage), algorithms, tuning, and debugging. With limited time and resources, consider which level will give the most improvement: it won't always be an algorithm or program tuning.

In general, optimizations should proceed from top to bottom. Optimizations at the system level will have more impact than expression-level ones. Make sure you're solving the problem at the appropriate level.

This book is mostly going to talk about reducing CPU usage, reducing memory usage, and reducing latency. It's good to point out that you can very rarely do all three. Maybe CPU time is faster, but now your program uses more memory. Maybe you need to reduce memory space, but now the program will take longer.

Amdahl's Law tells us to focus on the bottlenecks. If you double the speed of routine that only takes 5% of the runtime, that's only a 2.5% speedup in total wall-clock. On the other hand, speeding up routine that takes 80% of the time by only 10% will improve runtime by almost 8%. Profiles will help identify where time is actually spent.

When optimizing, you want to reduce the amount of work the CPU has to do. Quicksort is faster than bubble sort because it solves the same problem (sorting) in fewer steps. It's a more efficient algorithm. You've reduced the work the CPU needs to do in order to accomplish the same task.

Program tuning, like compiler optimizations, will generally make only a small dent in the total runtime. Large wins will almost always come from an algorithmic change or data structure change, a fundamental shift in how your program is organized. Compiler technology improves, but slowly. Proebsting's Law says compilers double in performance every 18 *years*, a stark contrast with the (slightly misunderstood interpretation) of Moore's Law

that doubles processor performance every 18 *months*. Algorithmic improvements work at larger magnitudes. Algorithms for mixed integer programming improved by a factor of 30,000 between 1991 and 2008. For a more concrete example, consider this breakdown of replacing a brute force geo-spatial algorithm described in an Uber blog post with more specialized one more suited to the presented task. There is no compiler switch that will give you an equivalent boost in performance.

TODO: Optimizing floating point FFT and MMM algorithm differences in gttse07.pdf

A profiler might show you that lots of time is spent in a particular routine. It could be this is an expensive routine, or it could be a cheap routine that is just called many many times. Rather than immediately trying to speed up that one routine, see if you can reduce the number of times it's called or eliminate it completely. We'll discuss more concrete optimization strategies in the next section.

The Three Optimization Questions:

Do we have to do this at all? The fastest code is the code that's never run.

If yes, is this the best algorithm.

If yes, is this the best *implementation* of this algorithm.

Concrete optimization tips

Jon Bentley's 1982 work "Writing Efficient Programs" approached program optimization as an engineering problem: Benchmark. Analyze. Improve. Verify. Iterate. A number of his tips are now done automatically by compilers. A programmer's job is to use the transformations compilers *can't* do.

There are summaries of the book:

http://www.crowl.org/lawrence/programming/Bentley82.html

http://www.geoffprewett.com/BookReviews/WritingEfficientPrograms.html

and the program tuning rules:

https://web.archive.org/web/20080513070949/http://www.cs.bell-labs.com/cm/cs/pearls/apprules.html

When thinking of changes you can make to your program, there are two basic options: you can either change your data or you can change your code.

Data Changes

Changing your data means either adding to or altering the representation of the data you're processing. From a performance perspective, some of these will end up changing the O() associated with different aspects of the data structure. This may even include preprocessing the input to be in a different, more useful format.

Ideas for augmenting your data structure:

Extra fields

The classic example of this is storing the length of a linked list in a field in the root node. It takes a bit more work to keep it updated, but then querying the length becomes a simple field lookup instead of an O(n) traversal. Your data structure might present a similar win: a bit of bookkeeping during some operations in exchange for some faster performance on a common use case.

Similarly, storing pointers to frequently needed nodes instead of performing additional searches. This covers things like the "backwards" links in a doubly-linked list to make node removal O(1). Some skip lists keep a "search finger", where you store a pointer to where you just were in your data structure on the assumption it's a good starting point for your next operation.

Extra search indexes

Most data structures are designed for a single type of query. If you need two different query types, having an additional "view" onto your data can be large improvement. For example, a set of structs might have a primary ID (integer) that you use to look up in a slice, but sometimes need to look up with a secondary ID (string). Instead of iterating over the slice, you can augment your data structure with a map either from string to ID or directly to the struct itself.

Extra information about elements

For example, keeping a bloom filter of all the elements you've inserted can let you quickly return "no match" for lookups. These need to be small and fast to not overwhelm the rest of the data structure. (If a lookup in your main data structure is cheap, the cost of the bloom filter will outweigh any savings.)

If queries are expensive, add a cache.

At a larger level, an in-process or external cache (like memcache) can help. It might be excessive for a single data structure. We'll cover more about caches below.

These sorts of changes are useful when the data you need is cheap to store and easy to keep up-to-date.

These are all clear examples of "do less work" at the data structure level. They all cost space. Most of the time if you're optimizing for CPU, your program will use more memory. This is the classic space-time trade-off.

It's important to examine how this tradeoff can affect your solutions -- it's not always straight-forward. Sometimes a small amount of memory can give a significant speed, sometimes the tradeoff is linear (2x memory usage == 2x performance speedup), sometimes it's significantly worse: a huge amount of memory gives only a small speedup. Where you need to be on this memory/performance curve can affect what algorithm choices are reasonable. It's not always possible to just tune an algorithm parameter. Different memory usages might be completely different algorithmic approaches.

Lookup tables also fall into this space-time trade-off. A simple lookup table might just be a cache of previously requested computations.

If the domain is small enough, the *entire* set of results could be precomputed and stored in the table. As an example, this could be the approach taken for a fast popcount implementation, where by the number of set bits in byte is stored in a 256-entry table. A larger table could store the bits required for all 16-bit words. In this case, they're storing exact results.

A number of algorithms for trigonometric functions use lookup tables as a starting point for a calculation.

If your program uses too much memory, it's also possible to go the other way. Reduce space usage in exchange for increased computation. Rather than storing things, calculate them every time. You can also compress the data in memory and decompress it on the fly when you need it.

If the data you're processing is on disk, instead of loading everything into RAM, you could create an index for the pieces you need and keep that in memory, or pre-process the file into smaller workable chunks.

Small Memory Software is a book available online covering techniques for reducing the space used by your programs. While it was originally written targeting embedded developers, the ideas are applicable for programs on modern hardware dealing with huge amounts of data.

Rearrange your data

Eliminate structure padding. Remove extra fields. Use a smaller data type.

Change to a slower data structure

Simpler data structures frequently have lower memory requirements. For example, moving from a pointer-heavy tree structure to use slice and linear search instead.

Custom compression format for your data

Compression algorithms depend very heavily on what is being compressed. It's best to choose one that suites your data. If you have []byte, the something like snappy, gzip, lz4, behaves well. For floating point data there is go-tsz for time series and fpc for scientific data. Lots of research has been done around compressing integers, generally for information retrieval in search engines. Examples include delta encoding and varints to more complex schemes involving Huffman encoded xor-differences. You can also come up with custom compression formats optimized for exactly your data.

Do you need to inspect the data or can it stay compressed? Do you need random access or only streaming? If you need access to individual entries but don't want to decompress the entire thing, you can compress the data in smaller blocks and keep an index indicating what range of entries are in each block. Access to a single entry just needs to check the index and unpack the smaller data block.

If your data is not just in-process but will be written to disk, what about data migration or adding/removing

fields. You'll now be dealing with raw []byte instead of nice structured Go types, so you'll need unsafe and to consider serialization options.

We will talk more about data layouts later.

Modern computers and the memory hierarchy make the space/time trade-off less clear. It's very easy for lookup tables to be "far away" in memory (and therefore expensive to access) making it faster to just recompute a value every time it's needed.

This also means that benchmarking will frequently show improvements that are not realized in the production system due to cache contention (e.g., lookup tables are in the processor cache during benchmarking but always flushed by "real data" when used in a real system. Google's Jump Hash paper in fact addressed this directly, comparing performance on both a contended and uncontended processor cache. (See graphs 4 and 5 in the Jump Hash paper)

TODO: how to simulate a contended cache, show incremental cost TODO: sync.Map as a Go-ish example of cache-contention addressing

Another aspect to consider is data-transfer time. Generally network and disk access is very slow, and so being able to load a compressed chunk will be much faster than the extra CPU time required to decompress the data once it has been fetched. As always, benchmark. A binary format will generally be smaller and faster to parse than a text one, but at the cost of no longer being as human readable.

For data transfer, move to a less chatty protocol, or augment the API to allow partial queries. For example, an incremental query rather than being forced to fetch the entire dataset each time.

Algorithmic Changes

If you're not changing the data, the other main option is to change the code.

The biggest improvement is likely to come from an algorithmic change. This is the equivalent of replacing bubble sort (`O(n^2)`) with quicksort (`O(n log n)`) or replacing a linear scan through an array (`O(n)`) with a binary search (`O(log n)`) or a map lookup (`O(1)`).

This is how software becomes slow. Structures originally designed for one use is repurposed for something it wasn't designed for. This happens gradually.

It's important to have an intuitive grasp of the different big-O levels. Choose the right data structure for your problem. You don't have to always shave cycles, but this just prevents dumb performance issues that might not be noticed until much later.

The basic classes of complexity are:

O(1): a field access, array or map lookup

Advice: don't worry about it (but keep in mind the constant factor.)

O(log n): binary search

Advice: only a problem if it's in a loop

O(n): simple loop

Advice: you're doing this all the time

O(n log n): divide-and-conquer, sorting

Advice: still fairly fast

O(n*m): nested loop / quadratic

Advice: be careful and constrain your set sizes

Anything else between quadratic and subexponential

Advice: don't run this on a million rows

O(b ^ n), O(n!): exponential and up

Advice: good luck if you have more than a dozen or two data points

Link: http://bigocheatsheet.com

Let's say you need to search through of an unsorted set of data. "I should use a binary search" you think, knowing that a binary search is O(log n) which is faster than the O(n) linear scan. However, a binary search requires that the data is sorted, which means you'll need to sort it first, which will take O(n log n) time. If you're doing lots of searches, then the upfront cost of sorting will pay off. On the other hand, if you're mostly doing lookups, maybe having an array was the wrong choice and you'd be better off paying the O(1) lookup cost for a map instead.

Being able to analyze your problem in terms of big-O notation also means you can figure out if you're already at the limit for what is possible for your problem, and if you need to change approaches in order to speed things up. For example, finding the minimum of an unsorted list is `O(n)`, because you have to look at every single item. There's no way to make that faster.

If your data structure is static, then you can generally do much better than the dynamic case. It becomes easier to build an optimal data structure customized for exactly your lookup patterns. Solutions like minimal perfect hashing can make sense here, or precomputed bloom filters. This also make sense if your data structure is "static" for long enough and you can amortize the up-front cost of construction across many lookups.

Choose the simplest reasonable data structure and move on. This is CS 101 for writing "not-slow software". This should be your default development mode. If you know you need random access, don't choose a linked-

list. If you know you need in-order traversal, don't use a map. Requirements change and you can't always guess the future. Make a reasonable guess at the workload.

http://daslab.seas.harvard.edu/rum-conjecture/

Data structures for similar problems will differ in when they do a piece of work. A binary tree sorts a little at a time as inserts happen. A unsorted array is faster to insert but it's unsorted: at the end to "finalize" you need to do the sorting all at once.

When writing a package to be used by others, avoid the temptation to optimize upfront for every single use case. This will result in unreadable code. Data structures by design are effectively single-purpose. You can neither read minds nor predict the future. If a user says "Your package is too slow for this use case", a reasonable answer might be "Then use this other package over here". A package should "do one thing well".

Sometimes hybrid data structures will provide the performance improvement you need. For example, by bucketing your data you can limit your search to a single bucket. This still pays the theoretical cost of O(n), but the constant will be smaller. We'll revisit these kinds of tweaks when we get to program tuning.

Two things that people forget when discussion big-O notation:

One, there's a constant factor involved. Two algorithms which have the same algorithmic complexity can have different constant factors. Imagine looping over a list 100 times vs just looping over it once. Even though both are O(n), one has a constant factor that's 100 times higher.

These constant factors are why even though merge sort, quicksort, and heapsort all O(n log n), everybody uses quicksort because it's the fastest. It has the smallest constant factor.

The second thing is that big-O only says "as n grows to infinity". It talks about the growth trend, "As the numbers get big, this is the growth factor that will dominate the run time." It says nothing about the actual performance, or how it behaves with small n.

There's frequently a cut-off point below which a dumber algorithm is faster. A nice example from the Go standard library's `sort` package. Most of the time it's using quicksort, but it has a shell-sort pass then insertion sort when the partition size drops below 12 elements.

For some algorithms, the constant factor might be so large that this cut-off point may be larger than all reasonable inputs. That is, the O(n^2) algorithm is faster than the O(n) algorithm for all inputs that you're ever likely to deal with.

This also means you need to know representative input sizes, both for choosing the most appropriate algorithm and for writing good benchmarks. 10 items? 1000 items? 1000000 items?

This also goes the other way: For example, choosing to use a more complicated data structure to give you O(n) scaling instead of O(n^2), even though the benchmarks for small inputs got slower. This also applies to most

lock-free data structures. They're generally slower in the single-threaded case but more scalable when many threads are using it.

The memory hierarchy in modern computers confuses the issue here a little bit, in that caches prefer the predictable access of scanning a slice to the effectively random access of chasing a pointer. Still, it's best to begin with a good algorithm. We will talk about this in the hardware-specific section.

TODO: extending last paragraph, mention O() notation is an model where each operation has fixed cost. That's a wrong assumption on modern hardware.

> The fight may not always go to the strongest, nor the race to the fastest, but that's the way to bet. -- Rudyard Kipling

Sometimes the best algorithm for a particular problem is not a single algorithm, but a collection of algorithms specialized for slightly different input classes. This "polyalgorithm" quickly detects what kind of input it needs to deal with and then dispatches to the appropriate code path. This is what the sorting package mentioned above does: determine the problem size and choose a different algorithm. In addition to combining quicksort, shell sort, and insertion sort, it also tracks recursion depth of quicksort and calls heapsort if necessary. The `string` and `bytes` packages do something similar, detecting and specializing for different cases. As with data compression, the more you know about what your input looks like, the better your custom solution can be. Even if an optimization is not always applicable, complicating your code by determining that it's safe to use and executing different logic can be worth it.

This also applies to subproblems your algorithm needs to solve. For example, being able to use radix sort can have a significant impact on performance, or using quickselect if you only need a partial sort.

Sometimes rather than specialization for your particular task, the best approach is to abstract it into a more general problem space that has been well-studied by researchers. Then you can apply the more general solution to your specific problem. Mapping your problem into a domain that already has well-researched implementations can be a significant win.

Similarly, using a simpler algorithm means that tradeoffs, analysis, and implementation details are more likely to be more studied and well understood than more esoteric or exotic and complex ones.

Simpler algorithms can also be faster. These two examples are not isolated cases https://go-review.googlesource.com/c/crypto/+/169037 https://go-review.googlesource.com/c/go/+/170322/

TODO: notes on algorithm selection

TODO: improve worst-case behaviour at slight cost to average runtime linear-time regexp matching

While most algorithms are deterministic, there are a class of algorithms that use randomness as a way to simplify otherwise complex decision making step. Instead of having code that does the Right Thing, you use randomness to select a probably not *bad* thing. For example, a treap is a probabilistically balanced binary tree.

Each node has a key, but also is assigned a random value. When inserting into the tree, the normal binary tree insertion path is followed but the nodes also obey the heap property based on each nodes randomly assigned weight. This simpler approach replaces otherwise complicated tree rotating solutions (like AVL and Red Black trees) but still maintains a balanced tree with O(log n) insert/lookup "with high probability. Skip lists are another similar, simple data structure that uses randomness to produce "probably" O(log n) insertion and lookups.

Similarly, choosing a random pivot for quicksort can be simpler than a more complex median-of-medians approach to finding a good pivot, and the probability that bad pivots are continually (randomly) chosen and degrading quicksort's performance to O(n^2) is vanishingly small.

Randomized algorithms are classed as either "Monte Carlo" algorithms or "Las Vegas" algorithms, after two well known gambling locations. A Monte Carlo algorithm gambles with correctness: it might output a wrong answer (or in the case of the above, an unbalanced binary tree). A Las Vegas algorithm always outputs a correct answer, but might take a very long time to terminate.

Another well-known example of a randomized algorithm is the Miller-Rabin primality testing algorithm. Each iteration will output either "not prime" or "maybe prime". While "not prime" is certain, the "maybe prime" is correct with probability at least 1/2. That is, there are non-primes for which "maybe prime" will still be output. By running many iterations of Miller-Rabin, we can make the probability of failure (that is, outputing "maybe prime" for a composite number) as small as we'd like. If it passes 200 iterations, then we can say the number is composite with probability at most 1/(2^200).

Another area where randomness plays a part is called "The power of two random choices". While initially the research was applied to load balancing, it turned out to be widely applicable to a number of selection problems. The idea is that rather than trying to find the best selection out of a group of items, pick two at random and select the best from that. Returning to load balancing (or hash table chains), the power of two random choices reduces the expected load (or hash chain length) from O(log n) items to O(log log n) items. For more information, see The Power of Two Random Choices: A Survey of Techniques and Results

randomized algorithms: other caching algorithms statistical approximations (frequently depend on sample size and not population size)

TODO: batching to reduce overhead: https://lemire.me/blog/2018/04/17/iterating-in-batches-over-data-structures-can-be-much-faster/

TODO: - Algorithm Design Manual: http://algorist.com/algorist.html - How To Solve It By Computer - to what extent is this a "how to write algorithms" book? If you're going to change the code to speed it up, by definition you're writing new algorithms. Soo... maybe?

Benchmark Inputs

Real-world inputs rarely match the theoretical "worst case". Benchmarking is vital to understanding how your system behaves in production.

You need to know what class of inputs your system will be seeing once deployed, and your benchmarks must use instances pulled from that same distribution. As we've seen, different algorithms make sense at different input sizes. If your expected input range is <100, then your benchmarks should reflect that. Otherwise, choosing an algorithm which is optimal for n=10^6 might not be the fastest.

Be able to generate representative test data. Different distributions of data can provoke different behaviours in your algorithm: think of the classic "quicksort is O(n^2) when the data is sorted" example. Similarly, interpolation search is O(log log n) for uniform random data, but O(n) worst case. Knowing what your inputs look like is the key to both representative benchmarks and for choosing the best algorithm. If the data you're using to test isn't representative of real workloads, you can easily end up optimizing for one particular data set, "overfitting" your code to work best with one specific set of inputs.

This also means your benchmark data needs to be representative of the real world. Using purely randomized inputs may skew the behaviour of your algorithm. Caching and compression algorithms both exploit skewed distributions not present in random data and so will perform worse, while a binary tree will perform better with random values as they will tend to keep the tree balanced. (This is the idea behind a treap, by the way.)

On the other hand, consider the case of testing a system with a cache. If your benchmark input consists only a single query, then every request will hit the cache giving potentially a very unrealistic view of how the system will behave in the real world with a more varied request pattern.

Also, note that some issues that are not apparent on your laptop might be visible once you deploy to production and are hitting 250k reqs/second on a 40 core server. Similarly, the behaviour of the garbage collector during benchmarking can misrepresent real-world impact. There are (rare) cases where a microbenchmark will show a slow-down, but real-world performance improves. Microbenchmarks can help nudge you in the right direction but being able to fully test the impact of a change across the entire system is best.

Writing good benchmarks can be difficult.

https://timharris.uk/misc/five-ways.pdf

Use geometric mean to compare groups of benchmarks.

https://www.cse.unsw.edu.au/~cs9242/current/papers/Fleming_Wallace_86.pdf

Evaluating Benchmark Accuracy:

http://www.brendangregg.com/blog/2018-06-30/benchmarking-checklist.html

Program Tuning

Program tuning used to be an art form, but then compilers got better. So now it turns out that compilers can optimize straight-forward code better than complicated code. The Go compiler still has a long way to go to match gcc and clang, but it does mean that you need to be careful when tuning and especially when upgrading Go versions that your code doesn't become "worse". There are definitely cases where tweaks to work around the lack of a particular compiler optimization became slower once the compiler was improved.

My RC6 cipher implementation had a 10% speed up for the inner loop just by switching to `encoding/binary` and `math/bits` instead of my hand-rolled versions.

Similarly, the `compress/bzip2` package was sped by switching to simpler code the compiler was better able to optimize

If you are working around a specific runtime or compiler code generation issue, always document your change with a link to the upstream issue. This will allow you to quickly revisit your optimization once the bug is fixed.

Fight the temptation to cargo cult folklore-based "performance tips", or even over-generalize from your own experience. Each performance bug needs to be approached on its own merits. Even if something has worked previously, make sure to profile to ensure the fix is still applicable. Your previous work can guide you, but don't apply previous optimizations blindly.

Program tuning is an iterative process. Keep revisiting your code and seeing what changes can be made. Ensure you're making progress at each step. Frequently one improvement will enable others to be made. (Now that I'm not doing A, I can simplify B by doing C instead.) This means you need to keep looking at the entire picture and not get too obsessed with one small set of lines.

Once you've settled on the right algorithm, program tuning is the process of improving the implementation of that algorithm. In Big-O notation, this is the process of reducing the constants associated with your program.

All program tuning is either making a slow thing fast, or doing a slow thing fewer times. Algorithmic changes also fall into these categories, but we're going to be looking at smaller changes. Exactly how you do this varies as technologies change.

Making a slow thing fast might be replacing SHA1 or `hash/fnv1` with a faster hash function. Doing a slow thing fewer times might be saving the result of the hash calculation of a large file so you don't have to do it multiple times.

Keep comments. If something doesn't need to be done, explain why. Frequently when optimizing an algorithm you'll discover steps that don't need to be performed under some circumstances. Document them. Somebody else might think it's a bug and needs to be put back.

> Empty programs gives the wrong answer in no time at all.
>
> It's easy to be fast if you don't have to be correct.

"Correctness" can depend on the problem. Heuristic algorithms that are mostly-right most of the time can be fast, as can algorithms which guess and improve allowing you to stop when you hit an acceptable limit.

Cache common cases:

We're all familiar with memcache, but there are also in-process caches. Using an in-process cache saves the cost of both the network call and the cost of serialization. On the other hand, this increases GC pressure as there is more memory to keep track of. You also need to consider eviction strategies, cache invalidation, and thread-safety. An external cache will generally handle eviction for you, but cache invalidation remains a problem. Thread-safety can also be an issue with external caches as it becomes effectively shared mutable state either between different goroutines in the same service or even different service instances if the external cache is shared.

A cache saves information you've just spent time computing in the hopes that you'll be able to reuse it again soon and save the computation time. A cache doesn't need to be complex. Even storing a single item -- the most recently seen query/response -- can be a big win, as seen in the `time.Parse()` example below.

With caches it's important to compare the cost (in terms of actual wall-clock and code complexity) of your caching logic to simply refetching or recomputing the data. The more complex algorithms that give higher hit rates are generally not cheap themselves. Randomized cache eviction is simple and fast and can be effective in many cases. Similarly, randomized cache *insertion* can limit your cache to only popular items with minimal logic. While these may not be as effective as the more complex algorithms, the big improvement will be adding a cache in the first place: choosing exactly which caching algorithm gives only minor improvements.

It's important to benchmark your choice of cache eviction algorithm with real-world traces. If in the real world repeated requests are sufficiently rare, it can be more expensive to keep cached responses around than to simply recompute them when needed. I've had services where testing with production data showed even an optimal cache wasn't worth it. we simply did't have sufficient repeated requests to make the added complexity of a cache make sense.

Your expected cache hit ratio is important. You'll want to export the ratio to your monitoring stack. Changing ratios will show a shift in traffic. Then it's time to revisit the cache size or the expiration policy.

A large cache can increase GC pressure. At the extreme (little or no eviction, caching all requests to an expensive function) this can turn into memoization

Program tuning:

Program tuning is the art of iteratively improving a program in small steps. Egon Elbre lays out his procedure:

Come up with a hypothesis as to why your program is slow.

Come up with N solutions to solve it

Try them all and keep the fastest.

Keep the second fastest just in case.

Repeat.

Tunings can take many forms.

If possible, keep the old implementation around for testing.

If not possible, generate sufficient golden test cases to compare output to.

"Sufficient" means including edge cases, as those are the ones likely to get affected by tuning as you aim to improve performance in the general case.

Exploit a mathematical identity:
Note that implementing and optimizing numerical calculations is almost its own field
https://github.com/golang/go/commit/ed6c6c9c11496ed8e458f6e0731103126ce60223

https://gist.github.com/dgryski/67e6a7ff94c3a1add30eb26ec0ad8b0f

multiplication with addition

use WolframAlpha, Maxima, sympy and similar to specialize, optimize or create lookup-tables

(Also, https://users.ece.cmu.edu/~franzf/papers/gttse07.pdf)

moving from floating point math to integer math

or mandelbrot removing sqrt, or lttb removing abs, `a < b/c` => `a * c < b`

consider different number representations: fixed-point, floating-point, (smaller) integers,

fancier: integers with error accumulators (e.g. Bresenham's line and circle), multi-base numbers / redundant number systems

"pay only for what you use, not what you could have used"
zero only part of an array, rather than the whole thing

best done in tiny steps, a few statements at a time

cheap checks before more expensive checks:
e.g., strcmp before regexp, (q.v., bloom filter before query) "do expensive things fewer times"

common cases before rare cases i.e., avoid extra tests that always fail

unrolling still effective: https://play.golang.org/p/6tnySwNxG6O
code size. vs branch test overhead

using offsets instead of slice assignment can help with bounds checks, data dependencies, and code gen (less to copy in inner loop).

remove bounds checks and nil checks from loops: https://go-review.googlesource.com/c/go/+/151158

other tricks for the prove pass

this is where pieces of Hacker's Delight fall

Many folklore performance tips for tuning rely on poorly optimizing compilers and encourage the programmer to do these transformations by hand. Compilers have been using shifts instead of multiplying or dividing by a power of two for 15 years now -- nobody should be doing that by hand. Similarly, hoisting invariant calculations out of loops, basic loop unrolling, common sub-expression elimination and many others are all done automatically by gcc and clang and the like. Go's compiler does many of these and continues to improve. As always, benchmark before committing to the new version.

The transformations the compiler can't do rely on you knowing things about the algorithm, about your input data, about invariants in your system, and other assumptions you can make, and factoring that implicit knowledge into removing or altering steps in the data structure.

Every optimization codifies an assumption about your data. These *must* be documented and, even better, tested for. These assumptions are going to be where your program crashes, slows down, or starts returning incorrect data as the system evolves.

Program tuning improvements are cumulative. 5x 3% improvements is a 15% improvement. When making optimizations, it's worth it to think about the expected performance improvement. Replacing a hash function with a faster one is a constant factor improvement.

Understanding your requirements and where they can be altered can lead to performance improvements. One issue that was presented in the #performance Gophers Slack channel was the amount of time that was spent creating a unique identifier for a map of string key/value pairs. The original solution was to extract the keys, sort them, and pass the resulting string to a hash function. The improved solution we came up was to individually hash the keys/values as they were added to the map, then xor all these hashes together to create the identifier.

Here's an example of specialization.

Let's say we're processing a massive log file for a single day, and each line begins with a time stamp.

```
Sun  4 Mar 2018 14:35:09 PST <.........................>
```

For each line, we're going to call `time.Parse()` to turn it into a epoch. If profiling shows us `time.Parse()` is the bottleneck, we have a few options to speed things up.

The easiest is to keep a single-item cache of the previously seen time stamp and the associated epoch. As long

as our log file has multiple lines for a single second, this will be a win. For the case of a 10 million line log file, this strategy reduces the number of expensive calls to `time.Parse()` from 10,000,000 to 86400 -- one for each unique second.

TODO: code example for single-item cache

Can we do more? Because we know exactly what format the timestamps are in *and* that they all fall in a single day, we can write custom time parsing logic that takes this into account. We can calculate the epoch for midnight, then extract hour, minute, and second from the timestamp string -- they'll all be in fixed offsets in the string -- and do some integer math.

TODO: code example for string offset version

In my benchmarks, this reduced the time parsing from 275ns/op to 5ns/op. (Of course, even at 275 ns/op, you're more likely to be blocked on I/O and not CPU for time parsing.)

The general algorithm is slow because it has to handle more cases. Your algorithm can be faster because you know more about your problem. But the code is more closely tied to exactly what you need. It's much more difficult to update if the time format changes.

Optimization is specialization, and specialized code is more fragile to change than general purpose code.

The standard library implementations need to be "fast enough" for most cases. If you have higher performance needs you will probably need specialized implementations.

Profile regularly to ensure to track the performance characteristics of your system and be prepared to re-optimize as your traffic changes. Know the limits of your system and have good metrics that allow you to predict when you will hit those limits.

When the usage of your application changes, different pieces may become hotspots. Revisit previous optimizations and decide if they're still worth it, and revert to more readable code when possible. I had one system that I had optimized process startup time with a complex set of mmap, reflect, and unsafe. Once we changed how the system was deployed, this code was no longer required and I replaced it with much more readable regular file operations.

TODO(dgryski): hash function work should fall here; manually inlining, removing structs, unrolling loops, removing bounds checks

Optimization workflow summary

All optimizations should follow these steps:

determine your performance goals and confirm you are not meeting them

profile to identify the areas to improve.

This can be CPU, heap allocations, or goroutine blocking.

benchmark to determine the speed up your solution will provide using the built-in benchmarking framework (http://golang.org/pkg/testing/)
Make sure you're benchmarking the right thing on your target operating system and architecture.

profile again afterwards to verify the issue is gone

use https://godoc.org/golang.org/x/perf/benchstat or https://github.com/codahale/tinystat to verify that a set of timings are 'sufficiently' different for an optimization to be worth the added code complexity.

use https://github.com/tsenart/vegeta for load testing http services (+ other fancy ones: k6, fortio, fbender)
if possible, test ramp-up/ramp-down in addition to steady-state load

make sure your latency numbers make sense

TODO: mention github.com/aclements/perflock as cpu noise reduction tool

The first step is important. It tells you when and where to start optimizing. More importantly, it also tells you when to stop. Pretty much all optimizations add code complexity in exchange for speed. And you can *always* make code faster. It's a balancing act.

Garbage Collection

You pay for memory allocation more than once. The first is obviously when you allocate it. But you also pay every time the garbage collection runs.

Reduce/Reuse/Recycle. -- @bboreham

Stack vs. heap allocations

What causes heap allocations?

Understanding escape analysis (and the current limitation)

/debug/pprof/heap , and -base

API design to limit allocations:
allow passing in buffers so caller can reuse rather than forcing an allocation

you can even modify a slice in place carefully while you scan over it

passing in a struct could allow caller to stack allocate it

reducing pointers to reduce gc scan times
pointer-free slices

maps with both pointer-free keys and values

GOGC

buffer reuse (sync.Pool vs or custom via go-slab, etc)

slicing vs. offset: pointer writes while GC is running need writebarrier: https://github.com/golang/go/commit/b85433975aedc2be2971093b6bbb0a7dc264c8fd
no writebarrier if writing to stack https://github.com/golang/go/commit/2140975ebde164ea1eaa70fc72775c03567f2bc9

use error variables instead of errors.New() / fmt.Errorf() at call site (performance or style? interface requires pointer, so it escapes to heap anyway)

use structured errors to reduce allocation (pass struct value), create string at error printing time

size classes

beware pinning larger allocation with smaller substrings or slices

Runtime and compiler

cost of calls via interfaces (indirect calls on the CPU level)

runtime.convT2E / runtime.convT2I

type assertions vs. type switches

defer

special-case map implementations for ints, strings
map for byte/uint16 not optimized; use a slice instead.

You can fake a float64-optimized with math.Float{32,64}{from,}bits, but beware float equality issues

https://github.com/dgryski/go-gk/blob/master/exact.go says 100x faster; need benchmarks

bounds check elimination

[]byte <-> string copies, map optimizations

two-value range will copy an array, use the slice instead:
https://play.golang.org/p/4b181zkB1O

https://github.com/mdempsky/rangerdanger

use string concatenation instead of fmt.Sprintf where possible; runtime has optimized routines for it

Unsafe

And all the dangers that go with it

Common uses for unsafe

mmap'ing data files
struct padding

but not always sufficiently faster to justify complexity/safety cost

but "off-heap", so ignored by gc (but so would a pointerless slice)

need to think about serialization format: how to deal with pointers, indexing (mph, index header)

speedy de-serialization

binary wire protocol to struct when you already have the buffer

string <-> slice conversion, []byte <-> []uint32, ...

int to bool unsafe hack (but cmov) (but != 0 is also branch-free)

padding:
https://dave.cheney.net/2015/10/09/padding-is-hard

http://www.catb.org/esr/structure-packing/#_go_and_rust

https://golang.org/ref/spec#Size_and_alignment_guarantees

https://github.com/dominikh/go-tools structlayout, structlayout-optimize

write tests for struct layout with unsafe.Offsetof to notice breakage from unsafe or asm

Common gotchas with the standard library

time.After() leaks until it fires; use t := NewTimer(); t.Stop() / t.Reset()

Reusing HTTP connections...; ensure the body is drained (issue #?)

rand.Int() and friends are 1) mutex protected and 2) expensive to create
consider alternate random number generation (go-pcgr, xorshift)

binary.Read and binary.Write use reflection and are slow; do it by hand. (https://github.com/conformal/yubikey/commit/613e3b04ae2eeb78e6a19636b8ff8e9106d2e7bc)

use strconv instead of fmt if possible

Use `strings.EqualFold(str1, str2)` instead of `strings.ToLower(str1) ==`

`strings.ToLower(str2)` or `strings.ToUpper(str1) == strings.ToUpper(str2)` to efficiently compare strings if possible.

...

Alternate implementations

Popular replacements for standard library packages:

encoding/json -> ffjson, easyjson, jingo (only encoder), etc

net/http
fasthttp (but incompatible API, not RFC compliant in subtle ways)

httprouter (has other features besides speed; I've never actually seen routing in my profiles)

regexp -> ragel (or other regular expression package)

serialization
encoding/gob -> https://github.com/alecthomas/go_serialization_benchmarks

protobuf -> https://github.com/gogo/protobuf

all serialization formats have trade-offs: choose one that matches what you need
Write heavy workload -> fast encoding speed

Read-heavy workload -> fast decoding speed

Other considerations: encoded size, language/tooling compatibility

tradeoffs of packed binary formats vs. self-describing text formats

database/sql -> has tradeoffs that affect performance
look for drivers that don't use it: jackx/pgx, crawshaw sqlite, ...

gccgo (benchmark!), gollvm (WIP)

container/list: use a slice instead (almost always)

cgo

cgo is not go -- Rob Pike

Performance characteristics of cgo calls

Tricks to reduce the costs: batching

Rules on passing pointers between Go and C

syso files (race detector, dev.boringssl)

Advanced Techniques

Techniques specific to the architecture running the code

introduction to CPU caches

performance cliffs

building intuition around cache-lines: sizes, padding, alignment

OS tools to view cache-misses (perf)

maps vs. slices

SOA vs AOS layouts: row-major vs. column major; when you have an X, do you need another X or do you need a Y?

temporal and spacial locality: use what you have and what's nearby as much as possible

reducing pointer chasing

explicit memory prefetching; frequently ineffective; lack of intrinsics means function call overhead (removed from runtime)

make the first 64-bytes of your struct count

branch prediction

remove branches from inner loops: if a { for { } } else { for { } } instead of for { if a { } else { } } benchmark due to branch prediction structure to avoid branch

if i % 2 == 0 { evens++ } else { odds++ }

counts[i & 1] ++ "branch-free code", benchmark; not always faster, but frequently harder to read TODO: ASCII class counts example, with benchmarks

sorting data can help improve performance via both cache locality and branch prediction, even taking into account the time it takes to sort

function call overhead: inliner is getting better

reduce data copies (including for repeated large lists of function params)

Comment about Jeff Dean's 2002 numbers (plus updates)

cpus have gotten faster, but memory hasn't kept up

TODO: little comment about code-aligment free optimization (or unoptimization)

Concurrency

Figure out which pieces can be done in parallel and which must be sequential

goroutines are cheap, but not free.

Optimizing multi-threaded code
false-sharing -> pad to cache-line size

true sharing -> sharding

Overlap with previous section on caches and false/true sharing

Lazy synchronization; it's expensive, so duplicating work may be cheaper

things you can control: number of workers, batch size

You need a mutex to protect shared mutable state. If you have lots of mutex contention, you need to either reduce the shared, or reduce the mutable. Two ways to reduce the shared are 1) shard the locks or 2) process independently and combine afterwards. To reduce mutable: well, make your data structure read-only. You can also reduce the time the data needs be shared by reducing the critical section -- hold the lock as little as needed. Sometimes a RWMutex will be sufficient, although note that they're slower but they allow multiple readers in.

If you're sharding the locks, be careful of shared cache-lines. You'll need to pad to avoid cache-line bouncing between processors.

var stripe [8]struct{ sync.Mutex; _ [7]uint64 } // mutex is 64-bits; padding fills the rest of the cacheline

Don't do anything expensive in your critical section if you can help it. This includes things like I/O (which are cheap but slow).

TODO: how to decompose problem for concurrency TODO: reasons parallel implementation might be slower (communication overhead, best algorithm is sequential, ...)

Assembly

Stuff about writing assembly code for Go

compilers improve; the bar is high

replace as little as possible to make an impact; maintenance cost is high

good reasons: SIMD instructions or other things outside of what Go and the compiler can provide

very important to benchmark: improvements can be huge (10x for go-highway) zero (go-speck/rc6/farm32), or

even slower (no inlining)

rebenchmark with new versions to see if you can delete your code yet
TODO: link to 1.11 patches removing asm code

always have pure-Go version (purego build tag): testing, arm, gccgo

brief intro to syntax

how to type the middle dot

calling convention: everything is on the stack, followed by the return values.
everything is on the stack, followed by the return values

this might change golang/go#18597

https://science.raphael.poss.name/go-calling-convention-x86-64.html

using opcodes unsupported by the asm (asm2plan9, but this is getting rarer)

notes about why inline assembly is hard: golang/go#26891

all the tooling to make this easier:
asmfmt: gofmt for assembly https://github.com/klauspost/asmfmt

c2goasm: convert assembly from gcc/clang to goasm https://github.com/minio/c2goasm

go2asm: convert go to assembly you can link https://rsc.io/tmp/go2asm

peachpy/avo: higher-level assembler in python (peachpy) or Go (avo)

differences of above

https://github.com/golang/go/wiki/AssemblyPolicy

Design of the Go Assembler: https://talks.golang.org/2016/asm.slide

Optimizing an entire service

Most of the time you won't be presented with a single CPU-bound routine. That's the easy case. If you have a service to optimize, you need to look at the entire system. Monitoring. Metrics. Log lots of things over time so you can see them getting worse and so you can see the impact your changes have in production.

tip.golang.org/doc/diagnostics.html

references for system design: SRE Book, practical distributed system design

extra tooling: more logging + analysis

The two basic rules: either speed up the slow things or do them less frequently.

distributed tracing to track bottlenecks at a higher level

query patterns for querying a single server instead of in bulk

your performance issues may not be your code, but you'll have to work around them anyway

https://docs.microsoft.com/en-us/azure/architecture/antipatterns/

Tooling

Introductory Profiling

This is a quick cheat-sheet for using the pprof tooling. There are plenty of other guides available on this. Check out https://github.com/davecheney/high-performance-go-workshop.

TODO(dgryski): videos?

Introduction to pprof
go tool pprof (and https://github.com/google/pprof)

Writing and running (micro)benchmarks
small, like unit tests

profile, extract hot code to benchmark, optimize benchmark, profile.

-cpuprofile / -memprofile / -benchmem

0.5 ns/op means it was optimized away -> how to avoid

tips for writing good microbenchmarks (remove unnecessary work, but add baselines)

How to read it pprof output

What are the different pieces of the runtime that show up

malloc, gc workers

runtime._ExternalCode

Macro-benchmarks (Profiling in production)
larger, like end-to-end tests

net/http/pprof, debug muxer

because it's sampling, hitting 10 servers at 100hz is the same as hitting 1 server at 1000hz

Using -base to look at differences

Memory options: -inuse_space, -inuse_objects, -alloc_space, -alloc_objects

Profiling in production; localhost+ssh tunnels, auth headers, using curl.

How to read flame graphs

Tracer

Look at some more interesting/advanced tooling

other tooling in /x/perf

perf (perf2pprof)

intel vtune / amd codexl / apple instruments

https://godoc.org/github.com/aclements/go-perf

Appendix: Implementing Research Papers

Tips for implementing papers: (For `algorithm` read also `data structure`)

Don't. Start with the obvious solution and reasonable data structures.

"Modern" algorithms tend to have lower theoretical complexities but high constant factors and lots of implementation complexity. One of the classic examples of this is Fibonacci heaps. They're notoriously difficult to get right and have a huge constant factor. There has been a number of papers published comparing different heap implementations on different workloads, and in general the 4- or 8-ary implicit heaps consistently come out on top. And even in the cases where Fibonacci heap should be faster (due to O(1) "decrease-key"), experiments with Dijkstra's depth-first search algorithm show it's faster when they use the straight heap removal and addition.

Similarly, treaps or skiplists vs. the more complex red-black or AVL trees. On modern hardware, the "slower" algorithm may be fast enough, or even faster.

> The fastest algorithm can frequently be replaced by one that is almost as fast and much easier to understand.
>
> -- Douglas W. Jones, University of Iowa

and

> Rule 3. Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy.
>
> Rule 4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures. -- "Notes on C Programming" (Rob Pike, 1989)

The added complexity has to be enough that the payoff is actually worth it. Another example is cache eviction algorithms. Different algorithms can have much higher complexity for only a small improvement in hit ratio. Of course, you may not be able to test this until you have a working implementation and have integrated it into your program.

Sometimes the paper will have graphs, but much like the trend towards publishing only positive results, these will tend to be skewed in favour of showing how good the new algorithm is.

Choose the right paper.

Look for the paper their algorithm claims to beat and implement that.

Frequently, earlier papers will be easier to understand and necessarily have simpler algorithms.

Not all papers are good.

Look at the context the paper was written in. Determine assumptions about the hardware: disk space, memory usage, etc. Some older papers make different tradeoffs that were reasonable in the 70s or 80s but don't necessarily apply to your use case. For example, what they determine to be "reasonable" memory vs. disk usage tradeoffs. Memory sizes are now orders of magnitude larger, and SSDs have altered the latency penalty for using disk. Similarly, some streaming algorithms are designed for router hardware, which can make it a pain to translate into software.

Make sure the assumptions the algorithm makes about your data hold.

This will take some digging. You probably don't want to implement the first paper you find.

Make sure you understand the algorithm. This sounds obvious, but it will be impossible to debug otherwise.

https://blizzard.cs.uwaterloo.ca/keshav/home/Papers/data/07/paper-reading.pdf

A good understanding may allow you to extract the key idea from the paper and possibly apply just that to your problem, which may be simpler than reimplementing the entire thing.

The original paper for a data structure or algorithm isn't always the best. Later papers may have better explanations.

Some papers release reference source code which you can compare against, but

academic code is almost universally terrible

beware licensing restrictions ("research purposes only")

beware bugs; edge cases, error checking, performance etc.

Also look out for other implementations on GitHub: they may have the same (or different!) bugs as yours.

Other resources on this topic:

https://www.youtube.com/watch?v=8eRx5Wo3xYA

http://codecapsule.com/2012/01/18/how-to-implement-a-paper/