Optimizing Go string operations with practical examples

Alex Bledea

Photo by Chad Kirchoff on Unsplash

I'm going to show you how I took a very simple program and made it run almost 5 times faster, with very minimal adjustments. You may know that I'm doing Advent of Code, and I'll share my solution from day 2, and how I was able to improve its speed. (and if you're not participating in the contest, you definitely should!)

Just to give some background, the contest favors coding speed over solution efficiency, so in order to solve the problem quickly I wrote some working code as fast as I can, without thinking of any optimization. Most of the time, that will run almost instantly, at least for a good part of the problems. The creator of the contests mentions that every problem has a solution which takes no longer than 15 seconds on 10-year-old hardware.

> For those curious, my contest times for this problem were 10:34, rank 1426 and 12:56, rank 1071.

For that reason, I find it interesting to start from my original solutions and find ways to improve it, either through better algorithms or language-specific techniques. This is a very good exercise which helps me improve both my abstract thinking and my language knowledge. In addition to that, it's a great idea after solving the problem to browse the subreddit and read not only other people's approaches and solutions, but also what other languages can do.

> For this article, I'll be talking about lower-level langauge optimizations, while the core solution will stay the same. I want to showcase how very small changes that would often be considered "stylistic" can significantly affect your program's performance. These optimizations are specific to Go, but many of the concepts can be translated to different langauges.

So let's start with the problem statement, which I'll simplify for brevity. Your input consists of a bunch of games, each having this format:

```
Game 1: 4 blue, 7 red, 5 green; 3 blue, 4 red, 5 green; 3 red, 11 green
```

```
Game 2: 2 blue, 8 red, 1 green; 15 blue, 2 green, 8 red;
Game 3: 1 red, 6 green, 4 blue
```

The game has an index, followed by the rounds that have been played, separated by `;`. What you're required to do for part (a) is:

- find all games in which *every round* has at most 12 red, 13 green and 14 blue;
- add up all those games number.

For example, in `Game 1`, no round exceeded the limit for red, green, blue. However, in `Game 2`, the second round exceeded the maximum limit for blue. In the case above, the games which satisfy the condition are the first and the third, meaning the solution would be `1+3=4`

Since part (b) is similar to part (a) and the same optimizations apply, I will only be discussing the changes for part (a).

Now, let's get to the fun stuff. We can quickly build up an idea for some algorithm, with the following steps:

- start with a counter;
- read each line individually;
- find the section corresponding to the scores (after `Game x:`);
- get all individual rounds (`x green, x red, x blue`);
- check if any individual round does not satisfy the condition;
- if all individual rounds are good, increment the counter with the game number (which in this case is the same as the line number, simplifying the problem a bit).

That is the pseudocode we'll keep in mind. The core algorithm is not difficult at all; most of the effort will be invested in parsing the strings. Let's take a look at my first working contest solution (which I refactored here for readability, the actual solution is in the linked commit hash):

```
func part1() int {
 sumOfGameIndices := 0




 for gameIndex, line := range inputLines {


  parts := strings.Split(line, ": ")
  game := parts[1]


  rounds := strings.Split(game, "; ")

  isPossible := false
  for _, round := range rounds {
   red, green, blue := findScores(round)
   if isRoundPossible(red, green, blue) {
    isPossible = true
   } else {
    isPossible = false



    break
   }
  }
  if isPossible {
   sumOfGameIndices += gameIndex + 1
  }
 }
 return sumOfGameIndices
}
```

The reason why I parsed the input in an array of strings with the lines is to streamline the benchmarks

> implementations. It is of course possible and more efficient to parse each line after reading it using its file descriptor, but that may involve background noise in your benchmarks like system calls or memory allocations. For example, Go's *`bufio.Scanner`* buffered file reader does a lot of allocations if using `scanner.Text()`, polluting benchmarks.
>
> Of course, reading the file into an array of strings also allocates, maybe even more, and uses much more memory, but does not interfere wtih benchmarks.

Let's also reveal the `findScores()` function:

```
func findScores(round string) (int, int, int) {


 var red, green, blue int

 individualScores := strings.Split(round, ", ")
 for _, score := range individualScores {

  scoreWithColor := strings.Split(score, " ")
  scoreStr := scoreWithColor[0]
  color := scoreWithColor[1]


  score, _ := strconv.Atoi(scoreStr)
  if color == "red" {
   red += score
  }
  if color == "green" {
   blue += score
  }
  if color == "blue" {
   green += score
  }
 }
 return red, blue, green
}
```

And finally, the simple `isRoundPossible()` implementation:

```
func isRoundPossible(red, green, blue int) bool {
 return red <= 12 && green <= 13 && blue <= 14
}
```

There isn't really much you could do to improve the core algorithm here, it's essentially just adding the numbers from the file. However, can you spot some (perhaps stylistic) implementation details that make this program slower?

With the code fresh in mind, let's put up some benchmarks to monitor the program's resources (the `ret` and `r` dance is to avoid dead code compiler optimization).

```
var ret int
func BenchmarkPart1(b *testing.B) {
 r := 0
 for n := 0; n < b.N; n++ {
  r = part1()
 }
 ret = r
}
```

And below are the results. I ran the benchmark a few times, after a fresh operating system restart (which improved the average execution time by about 1000ns):

```
goos: windows
goarch: amd64
cpu: AMD Ryzen 7 5800H with Radeon Graphics
BenchmarkPart1-16        29626      78729 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        29956      79863 ns/op    51248 B/op     1383 allocs/op
```

```
BenchmarkPart1-16        30465      79246 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        30079      79018 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        30069      79062 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        29917      79491 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        30338      79184 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        30518      79395 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        30187      79296 ns/op    51248 B/op     1383 allocs/op
BenchmarkPart1-16        30144      79508 ns/op    51248 B/op     1383 allocs/op
PASS
ok   command-line-arguments 66.533s
```

I will just interpret the relevant bits and avoid the Go benchmark specifics. We can see that the program took on average around 79,000 nanoseconds to execute (that is ~0.08 miliseconds!), allocated 50KB on the heap during every execution, spread across 1383 total allocations.

At first glance, this doesn't really tell us anything about the program since we don't have something to compare it to, but the number of memory allocations stands out a bit. If we omit processing and storing the file in an array of strings (which is not benchmarked), it looks like we're doing quite a few allocations when running the solution. That seems a bit odd; we're just reading some numbers from an input string and doing some calculations. The numbers could easily live on the stack, and the input string is on the heap already. It doesn't feel like we would need any more heap allocations!

> Heap allocations are expensive operations, much more expensive than stack allocations. There are many resources out there which explain why (just Google heap vs stack allocations), but the explanation I like the most comes from this stackoverflow post. Overly simplified, a stack allocation is just a few CPU instructions, while a heap allocation implementation has about 10,000 lines of C code)

But where are all these allocations coming from? Something stands out: we're doing a lot of `strings.Split()` to parse this input. This method is also common enough across multiple languages that it seems natural to approach the input parsing this way. It could be a starting point.

Of course, that is just an assumption. However, a CPU profile of the program confirms the suspicion (I had to run the program 20,000 times in a loop because it was completing too fast to even retrieve a single CPU sample):

This part of the profile graph shows that more than 70% of the program's execution was spent inside the `strings.Split()` function. If we think about it conceptually, a string splitting function needs to find some indices and store the split parts in an array (which many times involves a memory allocation). That is shown above by the `strings.Index()` and `runtime.makeslice()` functions. The program spent close to half its execution time just allocating memory for the string slices.

In fact, if we do a memory profile, we see that *all allocations* are coming from the `strings.Split()` function (the other branches are specific to the runtime or the profiler):

Now, *do we really need* the `strings.Split()` function? The content is in the input line already, we just need to find it. If you think about it, we only need the `strings.Index()` part of the split function, to find the parts of the string that are relevant to us. That should not involve any memory allocations (the CPU profile proves that, but I didn't include that part of the image since it was too big) and usually standard libraries implements it in a really efficient way (in Go it boils down to the Rabin-Karp algorithm, and even assembly instructions in some cases).

So let's see how it looks like to search directly for the data we need, without splitting the input. We trimmed the `Game x:` prefix by splitting at first:

```
parts := strings.Split(line, ": ")
game := parts[1]
```

Instead, we could simply find the starting index of the game content, and slice from there:

```
gameContentIndex := strings.Index(line, ": ") + 2
game := line[gameContentIndex:]
```

Alternatively, it's possible to use `strings.Cut()` to achieve the same result in a single line of code (thanks to Risky Pribadi for pointing it out).

```
gameContentHeader, game, _ := strings.Cut(line, ": ")
```

> In Go, strings are immutable and therefore slicing will reference the underlying byte content of the existing string. This already exists in memory (either on the stack or heap) and will not require a new allocation.

Previously, we've also got the content of the individual rounds using the splitting function, since rounds were separated by ;. But the content is there already, we don't have to create a new array to store it again. Instead, we could approach it this way:

```
eog := 0


for eod != -1 {


 eod = strings.Index(game, "; ")



 var round string
 if eod == -1 {


  round = game
 } else {


  round = game[:eod]
  game = game[eod+2:]
 }
  red, green, blue := findScores(round)

}
```

> Tip: instead of modifying the `game` variable directly, using an auxiliary variable to store the remaining game content like `remainingGameContent` would make the code easier to read.

The last place using the `strings.Split()` is the `findScores` function, and multiple times. The profiles show that this function consumes a lot of resources. Let's quickly remember what it did:

```
func findScores(round string) (int, int, int) {


 var red, green, blue int

 individualScores := strings.Split(round, ", ")
 for _, score := range individualScores {

  scoreWithColor := strings.Split(score, " ")
  scoreStr := scoreWithColor[0]
  color := scoreWithColor[1]


}
return red, blue, green
```

Again, we know the drill. For a string that just stores at most 3 values, and for getting the color and score for each value, there's no need to create extra slices. We can follow the same pattern and change the code to:

```
func findScores(round string) (int, int, int) {
 var red, green, blue int


 var eoc int
```

```
 for eoc != -1 {


  var currentScoreWithColor string
  eoc = strings.Index(round, ", ")
  if eoc == -1 {
   currentScoreWithColor = round
  } else {
   currentScoreWithColor = round[:eoc]
   round = round[eoc+2:]
  }



  space := strings.Index(currentScoreWithColor, " ")
  scoreStr := currentScoreWithColor[:space]
  color := currentScoreWithColor[space+1:]


 }
 return red, blue, green
}
```

> The same tip from above applies here as well, for the `round` variable. Also, it's worth mentioning that the conversion logic is efficient and doesn't allocate, done via `strconv.Atoi()` and not `fmt.Sprintf()`. The latter is more readable and convenient but far less efficient.

With all this, we've pretty much removed any reference to `strings.Split()` and refactored the entire logic using `strings.Index()`. Now the question is, did this actually work? Is the program more efficient? Let's examine the benchmarks:

```
goos: windows
goarch: amd64
cpu: AMD Ryzen 7 5800H with Radeon Graphics
BenchmarkPart1-16       142279      16891 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       142938      16806 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       144451      16772 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       142868      16891 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       143073      16892 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       142743      16741 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       143288      16890 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       143026      16880 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       140206      16799 ns/op        0 B/op        0 allocs/op
BenchmarkPart1-16       142514      16938 ns/op        0 B/op        0 allocs/op
```

That's quite a difference. The new code has effectively **no heap allocations** and the execution time had been reduced to an average of 16,800 nanoseconds from an initial 79,000. That's an almost 5 times boost in performance! And we didn't make any changes to the core algorithm, the differences being mostly in the style of the code.

On an ending note, I have to say that I did these benchmarks for fun. I don't want to say that you can just follow a similar approach and get a 5x performance out of your code right away. The moment I finished solving this problem, I knew it was a great candidate for this article, and therefore the example is a bit extreme. I just wanted to illustrate how useful it can sometimes be to understand the language you're working with in improving your program's performance. Our changes were relatively minimal, and didn't involve using any other dependencies than the standard library every language has.