Lab2

Lab 2: For Loops, Arrays, and Slices

Introduction

In Lab 1, you learned the basics of Go and TDD. Now it's time to level up! In this lab, you'll learn how to work with collections of data and repeat actions automatically.

What you'll learn:

- **For loops** - How to repeat actions efficiently
- **Arrays** - Working with fixed-size collections
- **Slices** - Using flexible, dynamic collections
- **Variadic functions** - Handling variable numbers of arguments
- **Benchmarking** - Measuring and optimizing code performance

How you'll learn: Following the Test-Driven Development (TDD) approach, you'll build a tournament scoring system step by step. Each concept will be introduced through practical, real-world examples where you write tests first, make them pass, and then refactor for better code quality.

Task Description

Throughout this lab, you'll build a tournament scoring system by completing the following tasks:

1. **Repeat Cheer** - Create a function that repeats text multiple times using for loops
2. **Calculate Totals** - Sum up player scores using arrays and slices
3. **Handle Multiple Players** - Process scores for any number of players with variadic functions

4. **Find Best Rounds** - Extract and analyze specific portions of data using slice slicing
5. **Optimize Performance** - Measure and improve code speed with benchmarking

Each task builds on the previous one, gradually introducing new concepts and techniques. Let's get started!

Prerequisites

- Go 1.21 or higher
- Linux environment (Ubuntu 22.04 recommended)
- Root/sudo access for testing

Step 0: Project Setup

First, let's set up our project. This is just like Lab 1.

```
mkdir tournament
cd tournament
go mod init github.com/yourusername/tournament
```

You'll see a `go.mod` file appear. That's Go's way of tracking your project and its dependencies.

Stage 1: The For Loop

Let's start simple. Your tournament needs crowd support, so let's create a function that repeats a cheer!

Step 1.1: Write the test first

Create a new file `repeat_test.go`:

```
package tournament

import"testing"

funcTestRepeat(t*testing.T){
   got:=Repeat("Go! ",3)
   want:="Go! Go! Go! "

if got!= want{
      t.Errorf("got %q but wanted %q", got, want)
}
}
```

What we're testing: A function that repeats text a specific number of times.

Step 1.2: Run the test (watch it fail)

```
gotest
```

You'll see:

```
./repeat_test.go:6:9: undefined: Repeat
FAIL    tournament [build failed]
```

The test is looking for something that doesn't exist yet.

Step 1.3: Create `repeat.go` with minimal code

```
package tournament

funcRepeat(textstring, countint)string{
```

```
return""
}
```

We're writing just enough code to make it compile, but not enough to pass the test.

Step 1.4: Run the test again

```
gotest
```

```
--- FAIL: TestRepeat (0.00s)
    repeat_test.go:9: got "" but wanted "Go! Go! Go! "
FAIL
FAIL    tournament    0.002s
```

Now it compiles, but the test fails because we're returning an empty string. Good!

Step 1.5: Make it pass with a for loop

```
package tournament

funcRepeat(textstring, countint)string{
var resultstring
for i:=0; i< count; i++{
        result= result+ text
}
return result
}
```

- `var result string` - Create an empty string to build our result
- `for i := 0; i < count; i++` - This is the for loop:

  - `i := 0` - Start with i at 0
  - `i < count` - Keep looping while i is less than count
  - `i++` - Add 1 to i after each loop
- Inside the loop: add text to result each time

Step 1.6: Run the test

```
gotest
```

```
PASS
ok      tournament    0.001s
```

Success! You've written your first for loop in Go.

Learning Outcomes:

- How to write a for loop with initialization, condition, and increment
- How to build strings by concatenating in a loop

Stage 2: Arrays - Fixed Size Collections

Now let's track game scores. A player completes exactly 5 rounds. We need to calculate their total score.

Step 2.1: Write the test

Create a new file `scores_test.go`:

```
package tournament

import"testing"

funcTestCalculateTotal(t*testing.T){
    scores:=[5]int{10,15,20,12,18}
    got:=CalculateTotal(scores)
    want:=75

if got!= want{
        t.Errorf("got %d but wanted %d", got, want)
}
}
```

- `[5]int` means "an array of exactly 5 integers"

Step 2.2: Run the test (watch it fail)

```
gotest
```

```
./scores_test.go:7:8: undefined: CalculateTotal
FAIL    tournament [build failed]
```

Step 2.3: Create the function in `scores.go`

```
package tournament

funcCalculateTotal(scores[5]int)int{
    total:=0
for i:=0; i<5; i++{
        total= total+ scores[i]
}
return total
}
```

- `scores[i]` accesses the element at position i

- `scores[0]` is the first element (10)
- `scores[4]` is the last element (18)
- We add each score to the total

Step 2.4: Run the test

```
gotest
```

```
PASS
ok      tournament    0.001s
```

Great! But there's a better way to write this in Go.

Step 2.5: Refactor with `range`

Go has a cleaner way to loop through arrays and slices:

```
package tournament

funcCalculateTotal(scores[5]int)int{
    total:=0
for_, score:=range scores{
        total+= score
}
return total
}
```

- `range scores` iterates through each element automatically
- Returns two values: index and value
- `_` means "ignore the index"
- `score` is the current element's value

- `total += score` is shorthand for `total = total + score`

Step 2.6: Run tests to confirm refactoring worked

```
gotest
```

```
PASS
ok      tournament    0.001s
```

Learning Outcomes:

- How to declare and use arrays with fixed size `[5]int`
- How to access array elements with indices
- The `range` keyword for cleaner iteration

Stage 3: Slices - Flexible Collections

Problem: What if games have different numbers of rounds? Some players play 3 rounds, others play 7. Arrays won't work because the size is fixed!

Solution: Slices! They're like arrays but can grow and shrink.

Step 3.1: Write tests for different sizes

Update your `scores_test.go`:

```
funcTestCalculateTotalSlice(t*testing.T){
    t.Run("with 5 scores",func(t*testing.T){
        scores:=[]int{10,15,20,12,18}
        got:=CalculateTotal(scores)
        want:=75

if got!= want{
```

```
          t.Errorf("got %d but wanted %d", got, want)
}
})

   t.Run("with 3 scores",func(t*testing.T){
      scores:=[]int{25,30,35}
      got:=CalculateTotal(scores)
      want:=90

if got!= want{
          t.Errorf("got %d but wanted %d", got, want)
}
})
}
```

Notice: `[]int` instead of `[5]int` - no number means it's a slice!

Step 3.2: Run the test (watch it fail)

```
gotest
```

```
./scores_test.go:11:15: cannot use scores (variable of type []int) as [5]int value
in argument to CalculateTotal
./scores_test.go:20:15: cannot use scores (variable of type []int) as [5]int valu
e in argument to CalculateTotal
FAIL    tournament [build failed]
```

The compiler is telling us we can't pass a slice to a function expecting an array. Let's fix it!

Step 3.3: Update the function to use slices

```
package tournament

funcCalculateTotal(scores[]int)int{
    total:=0
for_, score:=range scores{
        total+= score
}
return total
}
```

That's it! The code is exactly the same. The only change is the type signature: `[]int` instead of `[5]int`.

Step 3.4: Run the tests

```
gotest
```

```
PASS
ok      tournament    0.001s
```

Step 3.5: Add edge case tests

Let's test what happens with empty slices and single elements:

```
t.Run("with empty slice",func(t*testing.T){
    scores:=[]int{}
    got:=CalculateTotal(scores)
    want:=0

if got!= want{
        t.Errorf("got %d but wanted %d", got, want)
}
})

t.Run("with single score",func(t*testing.T){
    scores:=[]int{42}
    got:=CalculateTotal(scores)
    want:=42

if got!= want{
        t.Errorf("got %d but wanted %d", got, want)
}
})
```

Step 3.6: Run all tests

```
gotest
```

```
PASS
ok      tournament    0.002s
```

All tests pass! Our function handles edge cases automatically because `range` works with empty slices too.

Understanding slices:

```
scores:=[]int{10,20,30}

// Get the length
len(scores)// Returns 3

// Access elements (just like arrays)
scores[0]// 10
```

```
scores[1]// 20

// Add elements
scores=append(scores,40)
// Now scores = []int{10, 20, 30, 40}
```

Learning Outcomes:

- The difference between arrays `[5]int` and slices `[]int`
- How slices work with any number of elements
- Using `len()` to get the length of a slice
- Using `append()` to add elements to a slice

Stage 4: Multiple Players (Variadic Functions)

Now we have multiple players, each with their own set of scores. We need to calculate totals for all of them at once.

Step 4.1: Write the test

Add to `scores_test.go`:

```
funcTestCalculateTotals(t*testing.T){
    player1:=[]int{10,15,20}
    player2:=[]int{25,30}
    player3:=[]int{5,10,15,20}

    got:=CalculateTotals(player1, player2, player3)
    want:=[]int{45,55,50}

if!equal(got, want){
        t.Errorf("got %v but wanted %v", got, want)
}
}
```

```
// Helper function to compare slices
funcequal(a, b[]int)bool{
iflen(a)!=len(b){
returnfalse
}
for i:=range a{
if a[i]!= b[i]{
returnfalse
}
}
returntrue
}
```

We need a helper function `equal` because Go doesn't let you compare slices with `==`.

Step 4.2: Run the test (watch it fail)

```
gotest
```

```
./scores_test.go:66:9: undefined: CalculateTotals
FAIL    tournament [build failed]
```

Step 4.3: Create the variadic function

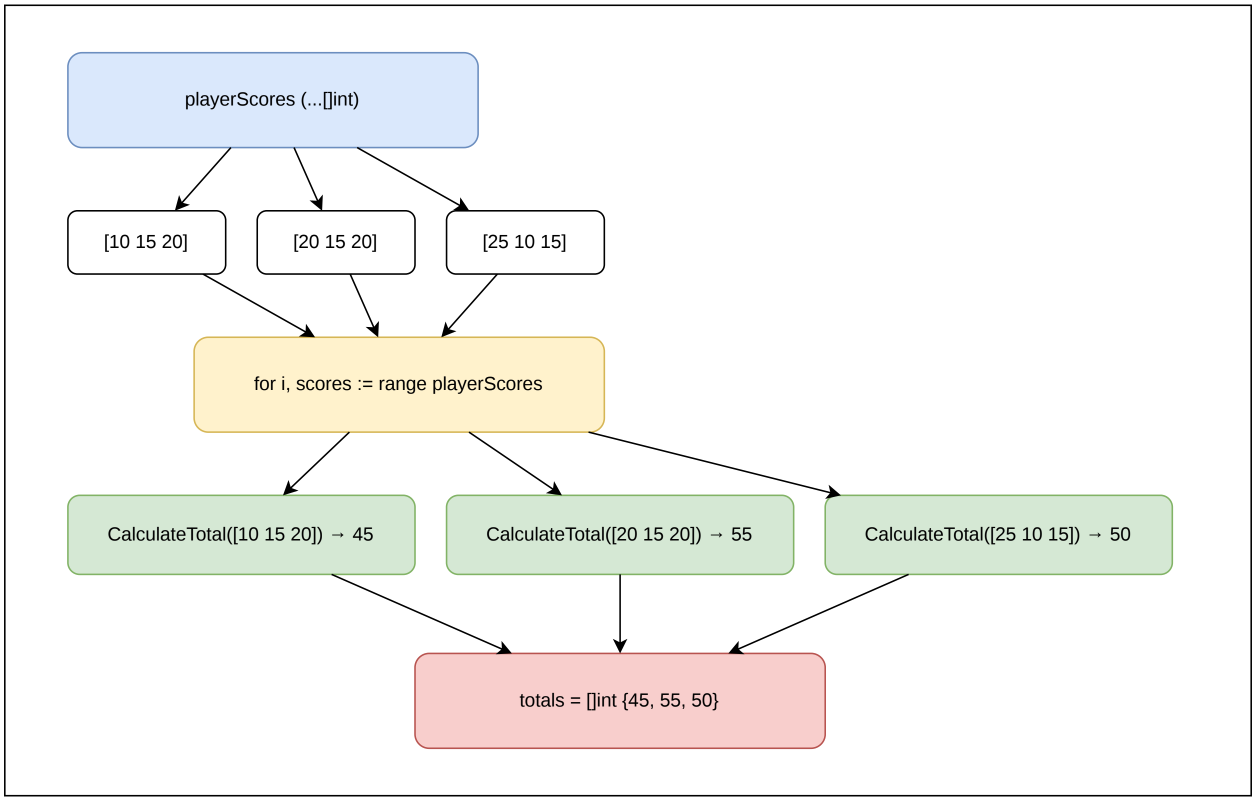
Add to `scores.go`:

```
funcCalculateTotals(playerScores...[]int)[]int{
    totals:=make([]int,len(playerScores))

for i, scores:=range playerScores{
        totals[i]=CalculateTotal(scores)
}

return totals
}
```

- `...[]int` means "accept any number of []int arguments" (variadic parameter)
- `make([]int, len(playerScores))` creates a slice with the right length
- `for i, scores := range playerScores` gives us both index and value
- We calculate each player's total and store it at index i


playerScores (...[]int)
[10 15 20]
[20 15 20]
[25 10 15]
for i, scores := range playerScores
CalculateTotal([10 15 20]) → 45
CalculateTotal([20 15 20]) → 55
CalculateTotal([25 10 15]) → 50
totals = []int {45, 55, 50}

Step 4.4: Run the test

```
gotest
```

```
PASS
ok      tournament    0.002s
```

Success! Now let's test edge cases.

Step 4.5: Add edge case tests

```
t.Run("single player",func(t*testing.T){
    player1:=[]int{100,200,300}

    got:=CalculateTotals(player1)
    want:=[]int{600}

if!equal(got, want){
        t.Errorf("got %v but wanted %v", got, want)
}
})

t.Run("no players",func(t*testing.T){
    got:=CalculateTotals()
    want:=[]int{}

iflen(got)!=len(want){
        t.Errorf("got %v but wanted empty slice", got)
}
})
```

Step 4.6: Run all tests

```
gotest
```

```
PASS
ok      tournament    0.002s
```

Understanding variadic functions:

You can call `CalculateTotals` with any number of arguments:

```
CalculateTotals(player1)// 1 player
CalculateTotals(player1, player2)// 2 players
CalculateTotals(player1, player2, player3)// 3 players
```

Learning Outcomes:

- How to create variadic functions with `...Type`
- Using `make()` to create slices with a specific length
- Using both index and value from `range`
- Writing helper functions for testing (like `equal`)
- Testing functions with different numbers of arguments

Common Mistake:

Don't confuse variadic parameters with slice of slices:

```
funcCalculateTotals(playerScores[][]int)// This is a slice of slices
funcCalculateTotals(playerScores...[]int)// This is variadic (correct!)
```

Try It Yourself:

1. Create a function `CalculateAverages` that returns average scores for each player
2. What happens if you pass an empty slice to a player? Test it!

Stage 5: Best Rounds (Slice Slicing)

Challenge: Each game has a "warm-up" round (the first round) that doesn't count. Find each player's best score from their actual rounds (excluding the first).

Step 5.1: Write the test

Add to `scores_test.go`:

```
funcTestBestRounds(t*testing.T){
    t.Run("normal cases",func(t*testing.T){
        player1:=[]int{5,10,15,20}// Warm-up: 5, best: 20
        player2:=[]int{30,25,35}// Warm-up: 30, best: 35

        got:=BestRounds(player1, player2)
        want:=[]int{20,35}

if!equal(got, want){
            t.Errorf("got %v but wanted %v", got, want)
}
})
}
```

Step 5.2: Run the test (watch it fail)

```
gotest
```

```
./scores_test.go:103:9: undefined: BestRounds
FAIL    tournament [build failed]
```

Step 5.3: Implement with slice slicing

Add to `scores.go`:

```
funcBestRounds(playerScores...[]int)[]int{
    bestScores:=make([]int,len(playerScores))

for i, scores:=range playerScores{
        bestScores[i]=findMaxFromActualRounds(scores)
}

return bestScores
}

funcfindMaxFromActualRounds(scores[]int)int{
// Handle edge cases
iflen(scores)==0||len(scores)==1{
return0
}

// Slice away the first element (warm-up round)
    actualRounds:= scores[1:]

// Find the maximum
    max:= actualRounds[0]
for_, score:=range actualRounds{
if score> max{
            max= score
}
}

return max
}
```

What's happening:

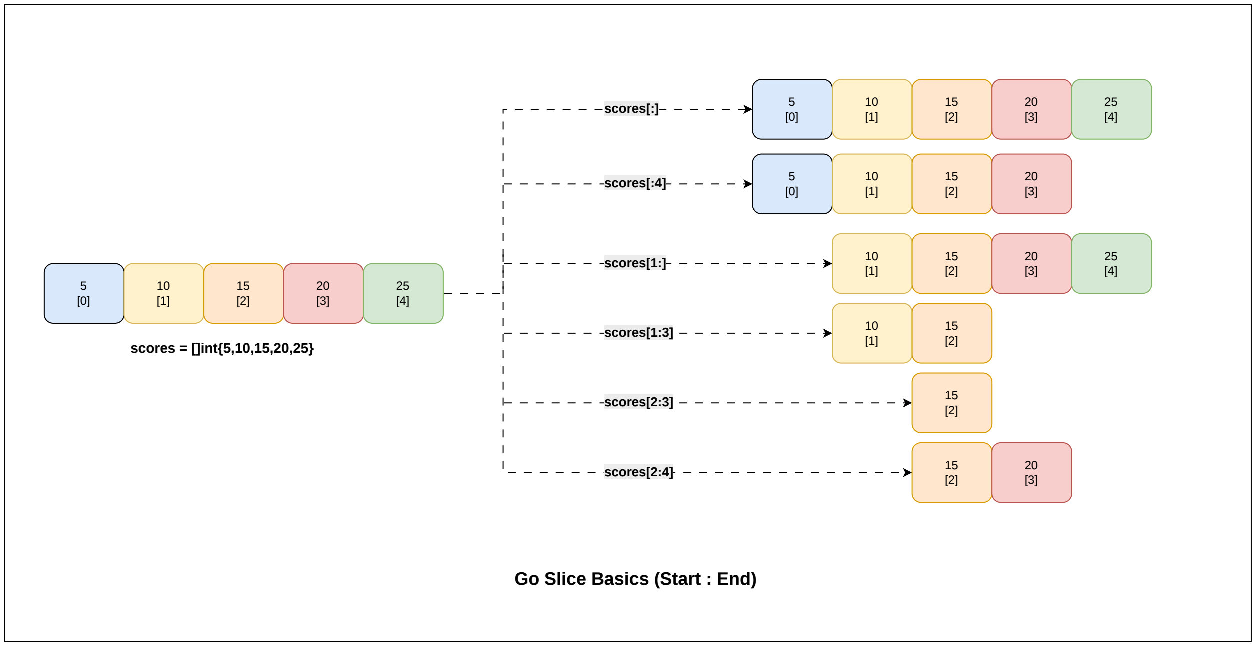
- `scores[1:]` creates a new slice starting from index 1 (skipping the first element)
- We check edge cases first (empty or single element)

- Then we find the maximum value in the remaining scores

Slice slicing syntax:

```
scores:=[]int{5,10,15,20}

scores[1:]// Everything from index 1 onwards: [10, 15, 20]
scores[:3]// Everything up to (but not including) index 3: [5, 10, 15]
scores[1:3]// From index 1 up to (not including) 3: [10, 15]
```



Step 5.4: Run the test

```
gotest
```

```
PASS
ok      tournament    0.002s
```

Step 5.5: Add edge case tests

```
t.Run("edge case: player with only warm-up",func(t*testing.T){
    player1:=[]int{10}// Only warm-up, no actual rounds

    got:=BestRounds(player1)
    want:=[]int{0}

if!equal(got, want){
        t.Errorf("got %v but wanted %v", got, want)
}
})

t.Run("edge case: empty slice",func(t*testing.T){
    player1:=[]int{}

    got:=BestRounds(player1)
    want:=[]int{0}

if!equal(got, want){
        t.Errorf("got %v but wanted %v", got, want)
}
})
```

Step 5.6: Run all tests

```
gotest
```

```
PASS
ok      tournament    0.002s
```

All tests pass, including edge cases!

Learning Outcomes:

- How to use slice slicing with `[start:end]` syntax
- Breaking complex logic into helper functions

Stage 6: Benchmarking - Making Code Faster

Remember our `Repeat` function from Stage 1? Let's see how fast it really is, and learn how to make it faster!

Step 6.1: Write a benchmark

Benchmark functions in Go are special test functions used to measure how quickly code runs. They always start with `Benchmark` (not `Test`) and take a `*testing.B` parameter instead of `*testing.T`. The `b.N` value in the loop tells Go how many times to run the code to get a good measurement. For each iteration, the code inside the loop is benchmarked:

Add to `repeat_test.go`:

```
funcBenchmarkRepeat(b*testing.B){
for i:=0; i< b.N; i++{
Repeat("Go! ",5)
}
}
```

- `b *testing.B`: Passed into every Benchmark function; use it to control the test and to get benchmarking tools.
- `b.N`: The number of times the benchmark code should be run. Go picks this automatically to get reliable results.
- Place the code you want to measure inside the loop: `for i := 0; i < b.N; i++ { ... }`.

This pattern lets Go figure out how fast your code is!

Step 6.2: Run the benchmark

```
gotest -bench=.
```

You'll see something like:

```
goos: linux
goarch: amd64
pkg: tournament
cpu: Intel(R) Xeon(R) Processor @ 2.20GHz
BenchmarkRepeat-2        3365713              340.8 ns/op
PASS
ok      tournament      1.522s
```

- `BenchmarkRepeat-2`: The name of your benchmark function (`BenchmarkRepeat`), and the number (`2`) shows how many threads or CPUs Go used for the test.
- `3365713`: The benchmark loop ran about 3.3 million times to measure performance.
- `340.8 ns/op`: On average, each call to `Repeat` took 340.8 nanoseconds (ns) per operation (`op`).

This output tells you both how many times your function executed and how long each execution took on average.

Step 6.3: Check memory allocations

```
gotest -bench=. -benchmem
```

```
BenchmarkRepeat-2        3872854              283.9 ns/op            64 B/op          4
allocs/op
```

```
PASS
ok      tournament      2.430s
```

- `3872854`: Go repeated our benchmark about 3.8 million times to measure speed.
- `283.9 ns/op`: Each run of the Repeat function takes about 284 nanoseconds.
- `64 B/op`: Every time we call Repeat, it uses 64 bytes of memory.
- `4 allocs/op`: For every repetition, Go makes 4 separate memory allocations.

Memory allocations happen because Go strings are *immutable*. Each time you execute `result = result + text`, Go creates a brand new string in memory rather than extending the old one. This results in multiple allocations.

Let’s see what happens step by step (and count the allocations):

```


For count = 5, result += text

Step 0: ""                (empty string)
Step 1: + "Go! "  ⟶  "Go! "                (allocation #1)
Step 2: + "Go! "  ⟶  "Go! Go! "            (allocation #2)
Step 3: + "Go! "  ⟶  "Go! Go! Go! "        (allocation #3)
Step 4: + "Go! "  ⟶  "Go! Go! Go! Go! "    (allocation #4)
Step 5: + "Go! "  ⟶  "Go! Go! Go! Go! Go! "
```

Above, 4 allocations occur at steps 1–4, matching the `4 allocs/op` measurement from the benchmark output. At each step (when you add "`+ text`" to the result), a new string is allocated for the growing result string.Go never reuses or expands the old one.This is why each repeat operation uses extra memory and more allocations! It's much slower and wastes memory.

Step 6.4: Optimize with `strings.Builder`

Add to `repeat.go`:

```
import"strings"

funcRepeatBuilder(textstring, countint)string{
var builder strings.Builder
for i:=0; i< count; i++{
	builder.WriteString(text)
}
return builder.String()
}
```

- `strings.Builder` pre-allocates a memory buffer
- Writes to the buffer without creating new strings each time
- Only creates the final string once with `.String()`

Step 6.5: Write a test for the new function

Add to `repeat_test.go`:

```
funcTestRepeatBuilder(t*testing.T){
    got:=RepeatBuilder("Go! ",3)
    want:="Go! Go! Go! "

if got!= want{
	t.Errorf("got %q but wanted %q", got, want)
}
}
```

Step 6.6: Run the test

```
gotest
```

```
PASS
ok      tournament    0.001s
```

Step 6.7: Benchmark the optimized version

Add to `repeat_test.go`:

```
funcBenchmarkRepeatBuilder(b*testing.B){
for i:=0; i< b.N; i++{
RepeatBuilder("Go! ",5)
}
}
```

Step 6.8: Run both benchmarks

You can try the benchmarks with different repeat counts (`n`) to see how memory allocations and performance change.

Below are the outputs from running `go test -bench=. -benchmem` with repeat counts of **5**, **10**, and **15** — both for the standard `Repeat` and optimized `RepeatBuilder` functions.

Case 1: n = 5

```
BenchmarkRepeat-2            2803276          425.6 ns/op          64 B/op
4 allocs/op
BenchmarkRepeatBuilder-2       4617885          269.5 ns/op          56 B/op
3 allocs/op
PASS
ok      tournament    3.15s
```

- Regular version makes 4 allocations. `RepeatBuilder` only 3.

- `RepeatBuilder` uses a bit less memory (56B vs 64B) and faster (425 ns vs 269 ns).

The `RepeatBuilder` function typically makes three memory allocations: the first happens when the internal buffer is created by the first call to `WriteString`, the second occurs if the buffer needs to grow to fit all the repeated data, and the third takes place when converting the builder's buffer into the final returned string using `builder.String()`.

Case 2: n = 10

```
BenchmarkRepeat-2               1265368           894.8 ns/op          248 B/op
9 allocs/op
BenchmarkRepeatBuilder-2         2736033            420.4 ns/op         120 B/o
p          4 allocs/op
PASS
ok      tournament    3.70s
```

Case 3: n = 15

```
BenchmarkRepeat-2                865504           1857 ns/op            536 B/op
14 allocs/op
BenchmarkRepeatBuilder-2         2633767            470.7 ns/op         120 B/op
4 allocs/op
PASS
ok      tournament    3.34s
```

Conclusion

In this lab, you practiced using for loops, arrays, and slices to efficiently manage and process collections of data in Go. You also worked with variadic functions and learned how to benchmark your Go code to measure performance. These foundational skills will help you write clear and efficient Go programs.

In the next lab, we'll dive into structs and interfaces—two core building blocks in Go for modeling and organizing real-world data and behaviors.