Slices & Strings

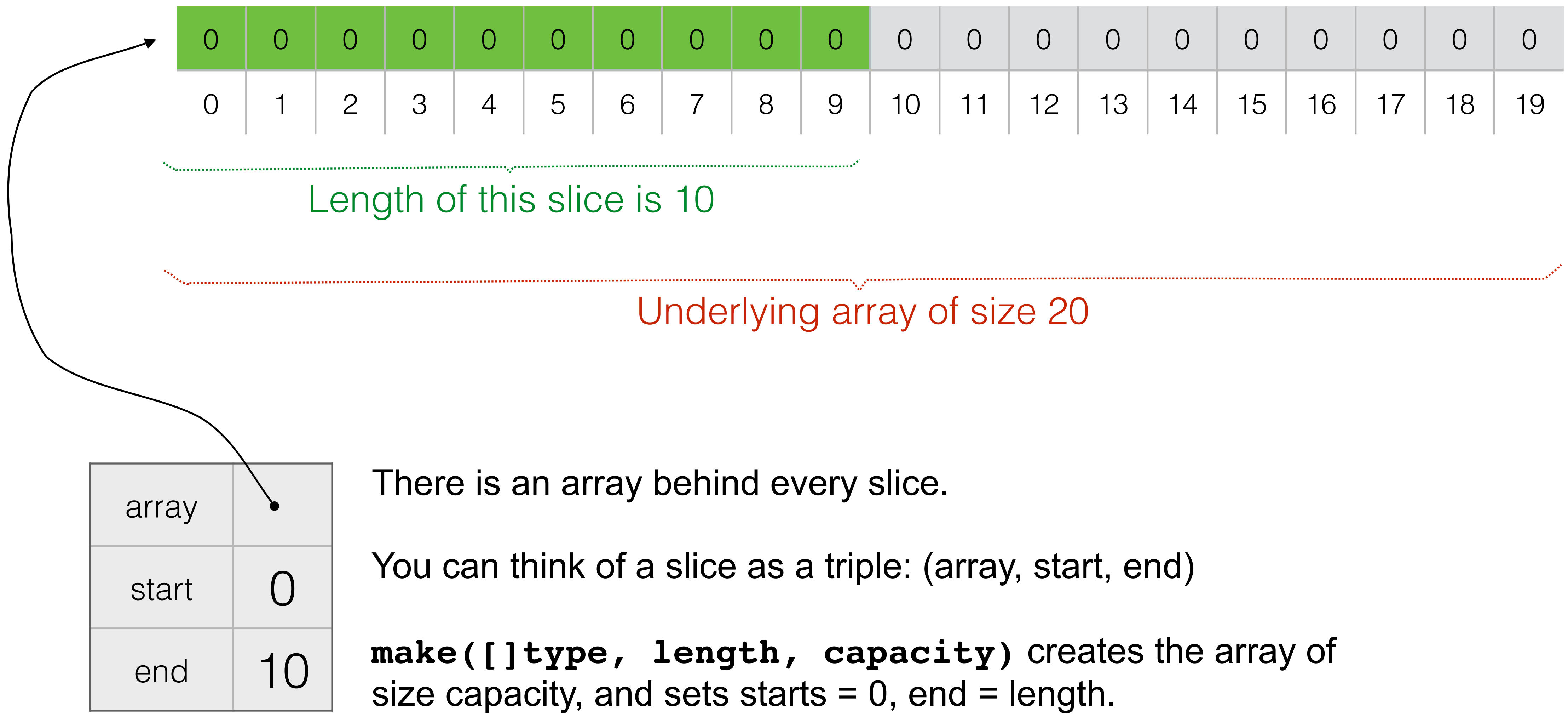
02-201 / 02-601

Slices

A slice variable is declared by not specifying a size in []

```
var s []int
// at this point s has the special value nil
// and can’t be used as an array
s = make([]int, 10, 20)
```

This creates an array of size 20 with a slice of size 10 inside it.



0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Length of this slice is 10

Underlying array of size 20

array	●
start	0
end	10

There is an array behind every slice.

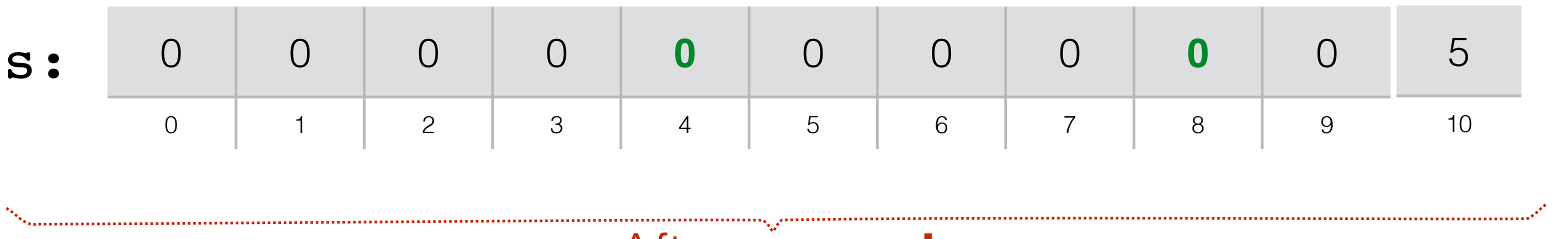
You can think of a slice as a triple: (array, start, end)

`make([]type, length, capacity)` creates the array of size capacity, and sets starts = 0, end = length.

Append

What if we want to make a slice bigger by adding something to the end of it?

```
s := make([]int, 10)
s = append(s, 5)
```



s:	0	0	0	0	0	0	0	0	0	0	5
	0	1	2	3	4	5	6	7	8	9	10

After **append**

Note: the syntax is the somewhat redundant:

```
s = append(s, 5)
```

An Updated primeSieve()

```
func primeSieve(isComposite []bool) {
    var biggestPrime = 2 // will hold the biggest prime found so far
    for biggestPrime < len(isComposite) {
        // knock out all multiples of biggestPrime
        for i := 2*biggestPrime; i < len(isComposite); i += biggestPrime {
            isComposite[i] = true
        }
        // find the next biggest non-composite number
        biggestPrime++
        for biggestPrime < len(isComposite) && isComposite[biggestPrime] {
            biggestPrime++
        }
    }
}

func main() {
    var composites []bool = make([]bool, 100000000)
    primeSieve(composites)
    var primeCount int = 0
    var primesList []int = make([]int, 0)
    for i, isComp := range composites {
        if !isComp && i >= 2 {
            primeCount++
            fmt.Println(“Number of primes ≤”, i, “is”, primeCount)
            primesList = append(primesList, i)
        }
    }
}
```

primeSieve takes a slice (which can be of any size)

len(isComposite) is the length of the slice (i.e. end - start + 1)

Create a new slice (with underlying array) (capacity == length by default)

primeSieve() can change the values of composites

Can **for…range** through a slice just like an array.

Another Append Example

```go
// take a box and list of 2D points and return the 2D points that lie in the box
func pointsInBox(
    x1,y1,x2,y2 float64,
    xs, ys []float64
) ([]float64, []float64) {

    var xout = make([]float64, 0)
    var yout = make([]float64, 0)

    for i := range xs {
        if x1 <= xs[i] && xs[i] <= x2 && y1 <= ys[i] && ys[i] <= y2 {
            xout = append(xout, xs[i])
            yout = append(yout, ys[i])
        }
    }
    return xout, yout
}

func main() {
    var x = []float64{-1, 3.2, 7.8, -2.45}
    var y = []float64{-2, -4.0, 3.14, 2.7}

    xlist, ylist := pointsInBox(-5,-5,5,5, x, y)

    for i := range xlist {
       fmt.Println(xlist[i], ylist[i])
    }
}
```

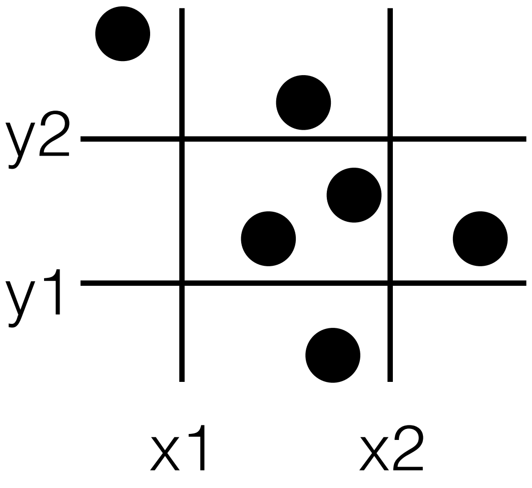
start with 0-length arrays

append adds element to end of array.

You must use the form:

```
    x = append(x, E)
```

to append E to slice x.



Array and Slice Literals

Recall: a *literal* is an explicit value in your program:
3 is a integer literal
“Pittsburgh” is a string literal

Can also write slice literals:
[]**float64**{3.2, -30, 84, 62}
[]**int**{1,2,3,6,7,8}

Slices: no explicit length
Arrays: explicit length
(same rule as when creating the variables)

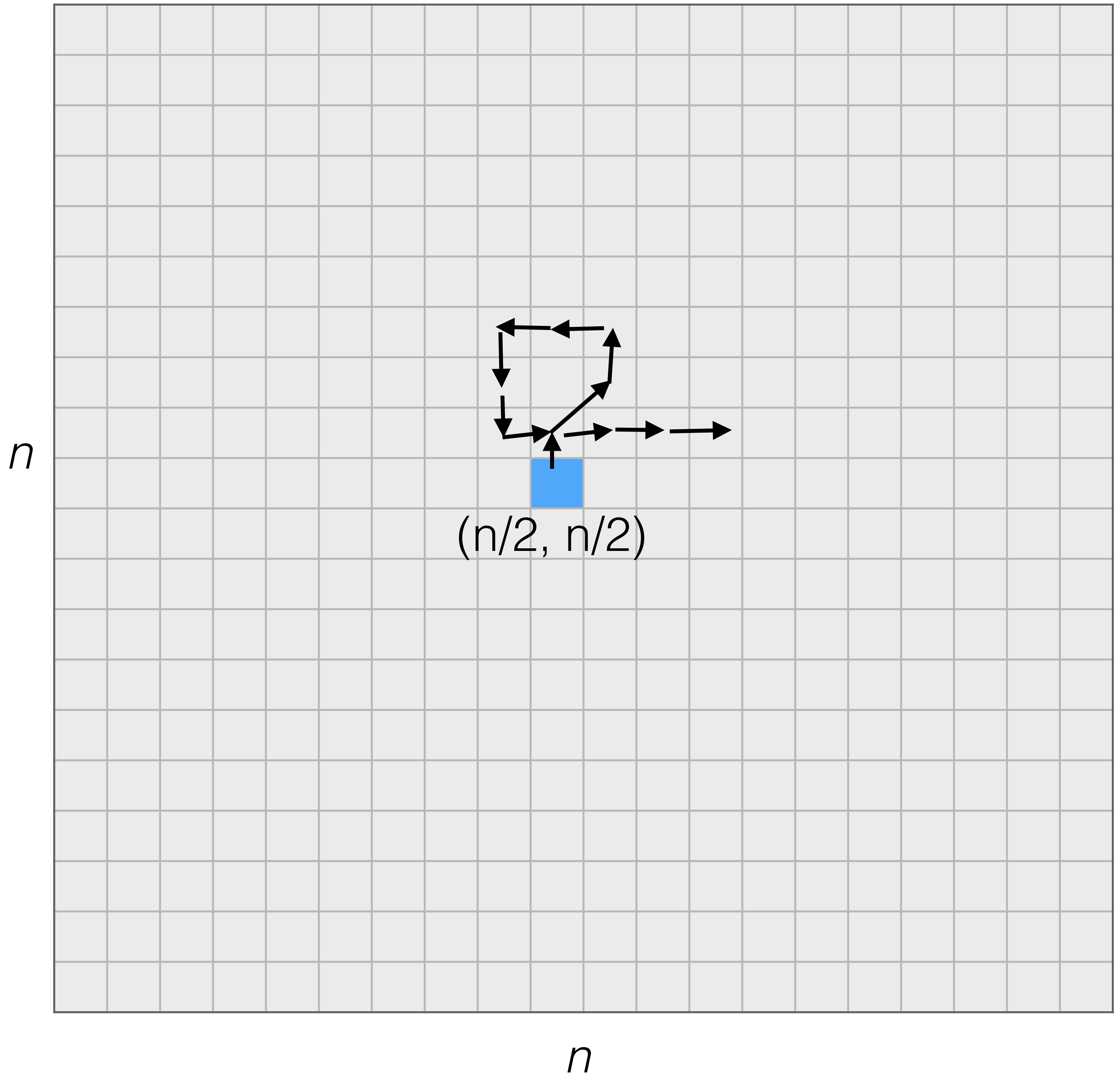
And array literals:
[4]**float64**{3.2, -30, 84, 62}
[6]**int**{1,2,3,6,7,8}

Useful if you have a fixed, short list of data.

Multi-dimensional Slices:
Self-Avoiding Walk Example

Example: Self-Avoiding Random Walks

Simulate a random walk on an n-by-n chessboard
but don't allow the walk to visit the same square twice



Need to keep track of where the walk has been → 2D slice

Creating a 2-D Slice

2-D slices are “slices of slices”. This creates a slice of *n* slices, each of which is not yet initialized:

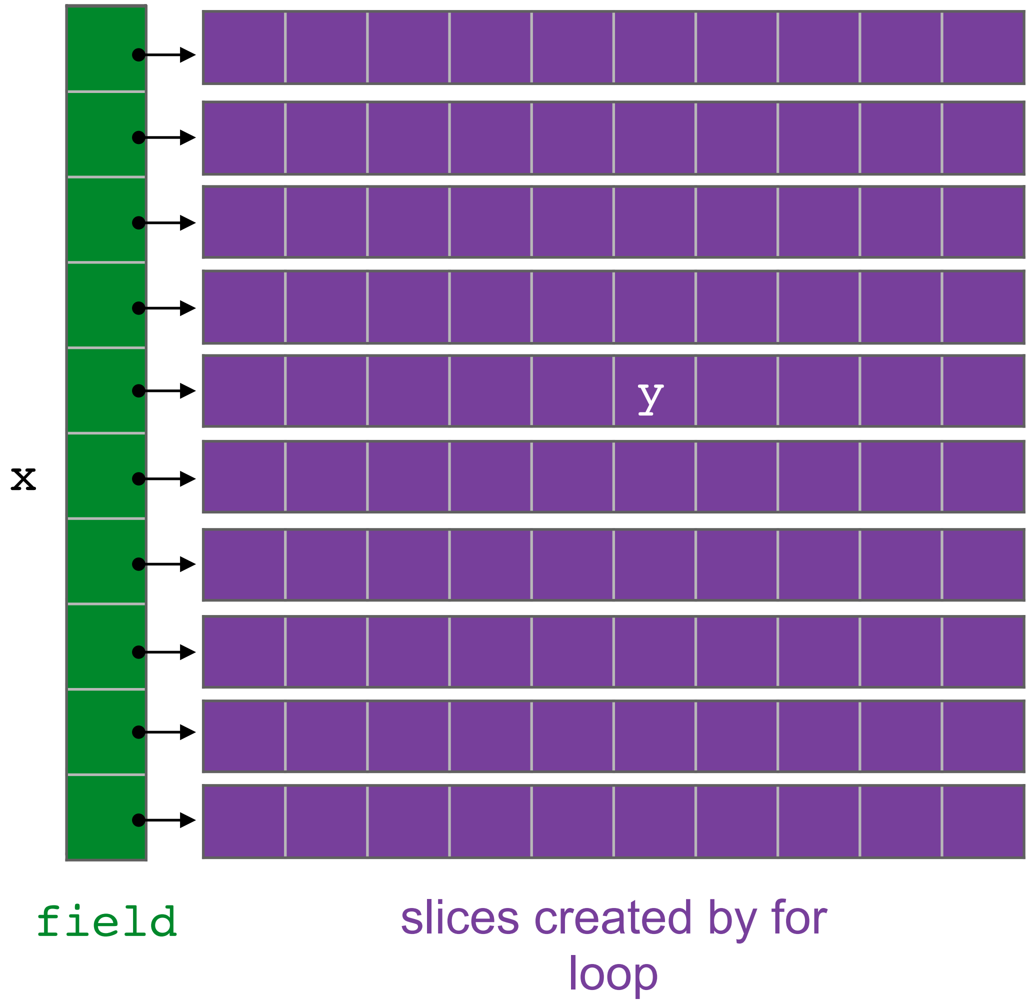
```
var field [][]bool = make([][]bool, n)
```

To initialize all the slices in field, you must write an explicit loop:

```
for row := range field {
   field[row] = make([]bool, n)
}
```

Can use field like a 2D array now:

```
var x, y = len(field)/2, len(field)/2
field[x][y] = true
```



Self-Avoiding Random Walk Code

```go
func selfAvoidingRandomWalk(n, steps int) {
    var field [][]bool = make([][]bool, n)
    for row := range field {
        field[row] = make([]bool, n)
    }
    var x, y = len(field)/2, len(field)/2

    field[x][y] = true
    fmt.Println(x,y)

    for i := 0; i < steps; i++ {
        // repeat until field is empty
        xnext, ynext := x, y
        for field[xnext][ynext] {
            xnext,ynext = randStep(x, y, len(field))
        }
        x, y = xnext, ynext
        field[x][y] = true
        fmt.Println(x,y)
    }
}
```

`make([][]bool, n)` is the same as `make([]([]bool), n)`

It creates a slice of slices, each of which hasn't yet been created

The green **for** loop creates slices for each of field[0], field[1], etc.

Bug: What if the walk gets stuck?

```
func selfAvoidingRandomWalk(n, steps int) {
    var field [][]bool = make([][]bool, n)
    for row := range field {
        field[row] = make([]bool, n)
    }
    var x, y = len(field)/2, len(field)/2

    field[x][y] = true
    fmt.Println(x,y)

    for i := 0; i < steps; i++ {
        if stuck(x,y,field)
           return
        }
        // repeat until field is empty
        xnext, ynext := x, y
        for field[xnext][ynext] {
            xnext,ynext = randStep(x, y, len(field))
        }
        x, y = xnext, ynext
        field[x][y] = true
        fmt.Println(x,y)
    }
}
```

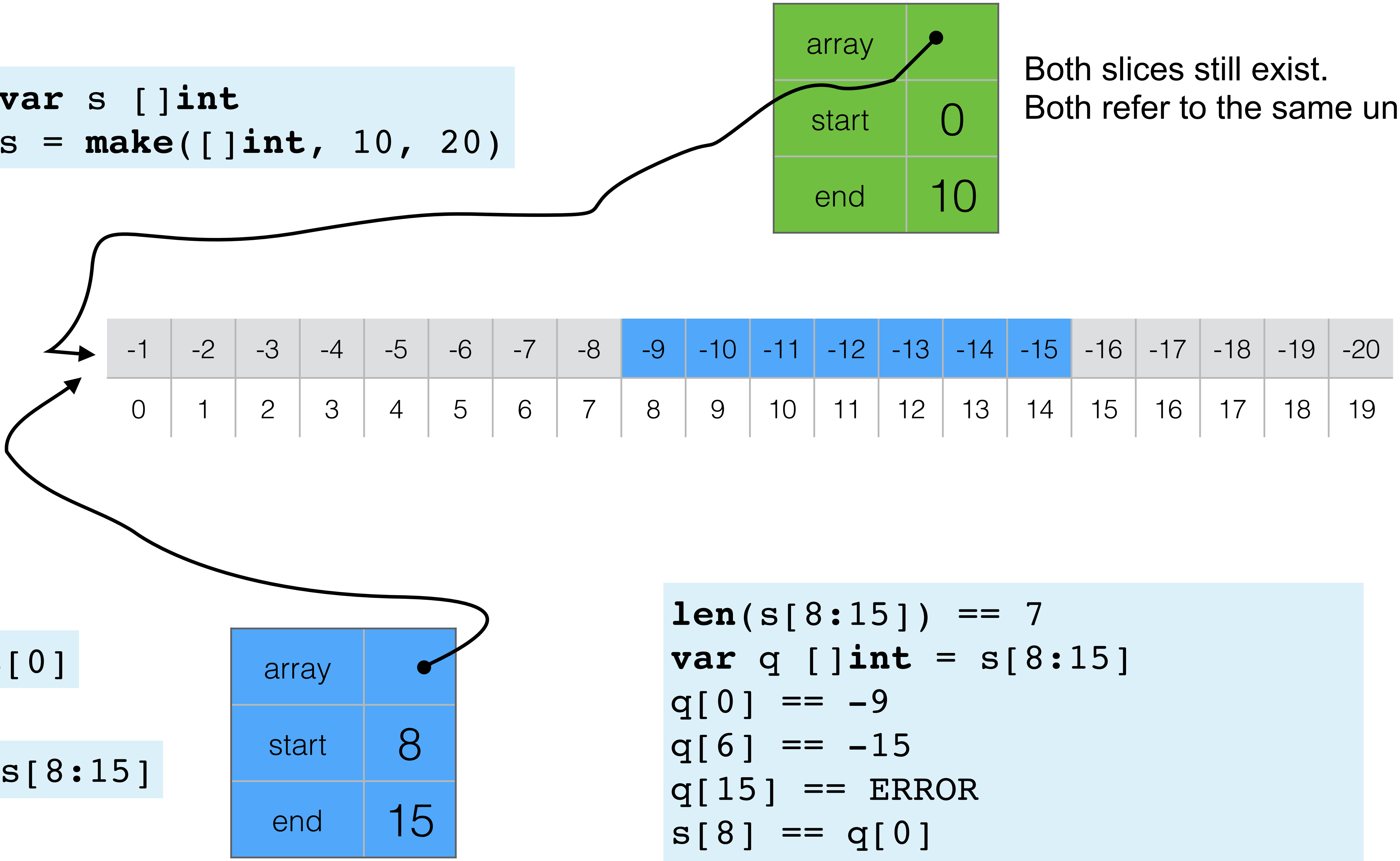
Add test to stop if stuck

Can initialize a slice using []type{value1, value2, …}

```
func stuck(x,y int, field [][]bool) bool {
   var deltas = []int{-1,0,1}
   for _, dx := range deltas {
      for _, dy := range deltas {
         nx, ny := x+dx, y+dy
         if inField(nx, n) && inField(ny, n) && !field[nx][ny] {
             return false
         }
      }
   }
   return true
}
```

Subslices: A picture

```
var s []int
s = make([]int, 10, 20)
```



array	
start	0
end	10

Both slices still exist.
Both refer to the same underlying array.

-1	-2	-3	-4	-5	-6	-7	-8	-9	-10	-11	-12	-13	-14	-15	-16	-17	-18	-19	-20
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

```
s[0]
```

```
s[8:15]
```

array	
start	8
end	15

```
len(s[8:15]) == 7
var q []int = s[8:15]
q[0] == -9
q[6] == -15
q[15] == ERROR
s[8] == q[0]
s[9] = 12 // now q[1] == 12 too
```

Subslices Example

```
// create a new slice of 0 length
var primes = []int
primes = make([]int, 0)

// add the first prime to our list
primes = append(primes, 2)

// add the next 999 primes to our list
for i := 1; i < 1000; i++ {
    next := getNextPrimeAfter(primes[len(primes)-1])
    primes = append(primes, next)
}

// print out the 27 through 50th prime
fmt.Println(primes[26:51])
```

Assume we have a function
getNextPrimeAfter(n **int**) **int**
that gives us the next prime after n

len(primes)-1 is the index of the last element in our primes slice.

Subslice: A[x:y] means the part of the slice from index x up to (but not including) y

Strings

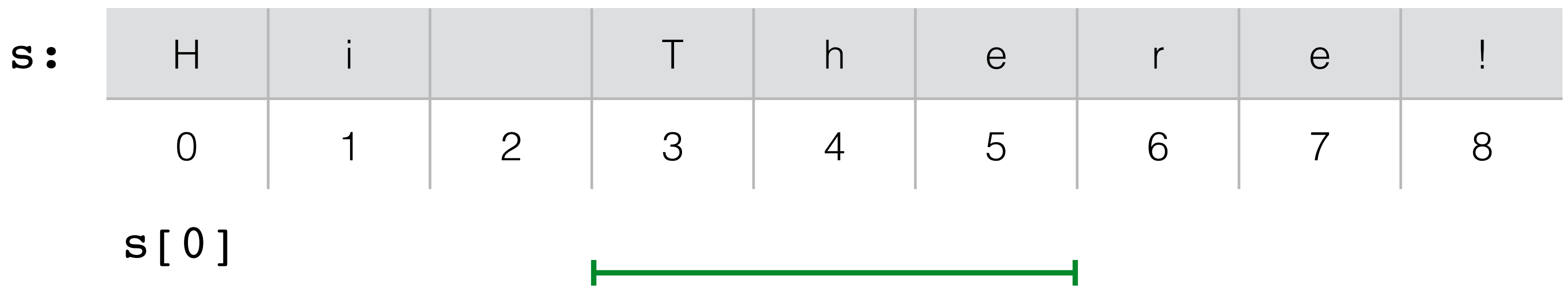
Indexing Strings

Strings work like arrays of **uint8**s in some ways:

You can access elements of string s with s[i].
You can iterate through their “letters” using **for…range**
You **cannot** modify a string once it has been created.

```
s := “Hi There!”
fmt.Println(s[0])             // prints H
fmt.Println(s[len(s)-1])      // prints !
fmt.Println(s[3:5])           // prints Th
fmt.Println(s[1:])            // prints i There!
fmt.Println(s[:4])            // prints Hi T
s[3] = “t”                    // ERROR! Can’t assign to strings

var str string = s[3:6]
fmt.Println(str)              // prints The
fmt.Println(str[0])           // prints T
```



s:	H	i		T	h	e	r	e	!
	0	1	2	3	4	5	6	7	8

`s[0]`

`s[3:6]`

`s[x:y]` creates a new string using characters [x,y) from `s`.
That is the string ends at character y-1.

`len(s[x:y]) == y - x`

Example: Reverse Complementing DNA

```
// Complement computes the reverse complement of a
// single given nucleotide. Ns become Ts as if they
// were As. Any other character induces a panic.
func Complement(c byte) byte {
     if c == ‘A’ { return ’T’ }
     if c == ‘C’ { return ‘G’ }
     if c == ‘G’ { return ‘C’ }
     if c == ’T’ { return ’A’ }

     panic(fmt.Errorf("Bad character: %s!", string(c)))
}
```

A letter is a single character inside single quotes ‘ ‘

The reverse complement of a string of DNA is the string reversed with C ↔ G and A ↔ T

DNA string `r`:	ACGGGATGA
complement of `r`:	TGCCCTACT
reverse complement of `r`:	TCATCCCGT

```
// reverseComplement() returns the reverse
// complement of the given string
func reverseComplement(r string) string {
     s := make([]byte, len(r))
     for i := 0; i < len(r); i++ {
          s[len(r)-i-1] = Complement(r[i])
     }
     return string(s)
}
```

Create a byte array s

Reverse and complement string r, storing the letters into s

Convert byte array s into **string**.

Slices Summary

- Slices work nearly the same as arrays except:
    - You have to explicitly initialize them with **make**(*type, length*)
    - Now *length* doesn’t need to be known when you write the program.
    - When you use a slice as a function parameter, it is not copied, and the function sees (and can modify) the original slice.

- You have to explicitly write code to create 2-D (or 3-D, etc.) slices.
- You should almost always use slices when you need to create a list of variables.