Assignment 1

Instructions

To run any of the solutions to the problems in this assignment, use the following command in the terminal. X refrences to the peroblems number:

```
go run pX.go
```

Replace the X with the number of the problem you want to run.

Problems

Problem 2

Defined two structs, one for the linked list Node that will help me create a snigle linked list, mening only pointing in one direction. And a struct Deque, containing pointers to the head and tail nodes (These two nodes that are the head and tail will be holding "garbage" values to assist in making the conccurency work properly). Both of these structs also have a mutex lock each to ensure safety of the data.

Implementation of the `Find()` function:

```go
func (dq *Deque) Find(k string, val any) (pred *Node, curr *Node) {
 pred = dq.head
 pred.Lock()
 curr = dq.head.nxt
 curr.Lock()
 if k == "pushBack" {
  for curr.val != math.MaxInt {
   pred.Unlock()
   pred = curr
   curr = curr.nxt
   curr.Lock()
  }
 } else if k == "popBack" {
  for curr.nxt.val != math.MaxInt {
   pred.Unlock()
   pred = curr
   curr = curr.nxt
   curr.Lock()
  }
 } else if k == "contains" && val != nil {
  for curr.val != val {
   if curr.val == math.MaxInt {
    break
   }
   pred.Unlock()
```

```
      pred = curr
      curr = curr.nxt
      curr.Lock()
     }
    }

    return pred, curr
  }
```

All other functions that edit the deque in any way use the find function to do so except for the pushfront and popfront functions which it was easier to implement what `Find()` would do for them directly due to it always being easy access from the head.

```
func (dq *Deque) PushFront(k int) {
 pred := dq.head
 pred.Lock()
 curr := dq.head.nxt
 curr.Lock()
 defer pred.Unlock()
 defer curr.Unlock()

 n := Node{val: k, nxt: curr}
 pred.nxt = &n
}
```

```
func (dq *Deque) PopBack() {
 pred, curr := dq.Find("popBack", nil)
 defer pred.Unlock()
 defer curr.Unlock()

 if curr.val != math.MaxInt {
  pred.nxt = curr.nxt
 }
}
```

I would argue that my implementation handles concurrent operations efficiently using mutex locks without introducing unnecessary complexity. The use of mutex locks helps in ensuring thread safety, preventing race conditions and ensuring the correctness of concurrent operations.

Problem 3

I implemented a function `generatePass()` to generate all possible passwords of a given length using recursion. Each goroutine generates passwords of only for their given index this does though limit the amount of goroutines who are able to for at the same time to the length of the charset. After a password is generated, they are sent to a channel. The function `hashAndCompare()` reads the passwords from that channel, hashes them and compares them to the target hash. When it finds a match it sends it to the `result` channel which leads to closing of the channel where all the passwords are sent to aswell as only letting the already started goroutines finish.

Benchmarking:

Number of goroutines: starts at length of the charset and is devied by 2 for every new ling in the block below.

```
# gorutiones -> time
36 -> 1m 30s
18 -> 9m 0s
9 -> 12m 46s
```

This time increase seems reasonable to me due too how many prefixes each goroutines have to handle depeding on the number of workers. This time trend countionus for other numbers of goroutines.

Problem 4

I define a struct `CountingSemaphore` with a mutex, a condition variable as well as a count. The use of the condition variable helps in ensuring that only a limited number of goroutines can enter the critical region simultaneously. The count is used to limit the number of goroutines that can enter the critical section at one time.

Example of how to use the `CountingSemaphore` struct. In my code example I initalize the `CountingSemaphore` with a count of two which will allow 2 goroutines to enter the critical section at one time, and make other gorutines wait until the `CountingSemaphore` count variable is more then 0 to be allowed to enter the critical section.

Here is how my Aquire and Release functions look like:

```
func (cs *CountingSemaphore) Acquire() {
 cs.mut.Lock()
 defer cs.mut.Unlock()

 for cs.count <= 0 {
  cs.cond.Wait()
 }

 cs.count--
}

func (cs *CountingSemaphore) Release() {
 cs.mut.Lock()
 defer cs.mut.Unlock()

 cs.count++
 cs.cond.Signal()
}
```