Go FAQ

Q: Why does 6.824 use Go for the labs?

A: Until a few years ago 6.824 used C++, which worked well. Go works a little better for 6.824 labs for a couple reasons. Go is garbage collected and type-safe, which eliminates some common classes of bugs. Go has good support for threads (goroutines), and a nice RPC package, which are directly useful in 6.824. Threads and garbage collection work particularly well together, since garbage collection can eliminate programmer effort to decide when the last thread using an object has stopped using it. There are other languages with these features that would probably work fine for 6.824 labs, such as Java.

Q: Do goroutines run in parallel? Can you use them to increase performance?

A: Go's goroutines are the same as threads in other languages. The Go runtime executes goroutines on all available cores, in parallel. If there are fewer cores than runnable goroutines, the runtime will pre-emptively time-share the cores among goroutines.

Q: How do Go channels work? How does Go make sure they are synchronized between the many possible goroutines?

A: You can see the source at https://golang.org/src/runtime/chan.go, though it is not easy to follow.

At a high level, a chan is a struct holding a buffer and a lock. Sending on a channel involves acquiring the lock, waiting (perhaps releasing the CPU) until some thread is receiving, and handing off the message. Receiving involves acquiring the lock and waiting for a sender. You could implement your own channels with Go sync.Mutex and sync.Cond.

Q: I'm using a channel to wake up another goroutine, by sending a dummy bool on the channel. But if that other goroutine is already running (and thus not receiving on the channel), the sending goroutine blocks. What should I do?

A: Try condition variables (Go's sync.Cond) rather than channels. Condition variables work well to alert goroutines that may (or may not) be waiting for something. Channels, because they are synchronous, are awkward if you're not sure if there will be a goroutine waiting at the other end of the channel.

Q: How can I have a goroutine wait for input from any one of a number of different channels? Trying to receive on any one channel blocks if there's nothing to read, preventing the goroutine from checking other channels.

A: Try creating a separate goroutine for each channel, and have each goroutine block on its channel. That's not always possible, but when it works it's often the simplest approach.

Otherwise try Go's select.

Q: When should we use sync.WaitGroup instead of channels? and vice versa?

A: WaitGroup is fairly special-purpose; it's only useful when waiting for a bunch of activities to complete. Channels are more general-purpose; for example, you can communicate values over channels. You can wait for multiple goroutines using channels, though it takes a few more lines of code than with WaitGroup.

```
Q: I need my code to perform a task once per second. What's the
easiest way to do that?

A: Create a goroutine dedicated to that periodic task. It should have
a loop that uses time.Sleep() to pause for a second, and then do the
task, and then loop around to the time.Sleep().

Q: How do we know when the overhead of spawning goroutines exceeds
the concurrency we gain from them?

A: It depends! If your machine has 16 cores, and you are looking for
CPU parallelism, you should have roughly 16 executable goroutines. If
it takes 0.1 second of real time to fetch a web page, and your network
is capable of transmitting 100 web pages per second, you probably need
about 10 goroutines concurrently fetching in order to use all of the
network capacity. Experimentally, as you increase the number of
goroutines, for a while you'll see increased throughput, and then
you'll stop getting more throughput; at that point you have enough
goroutines from the point of view of performance.

Q: How would one create a Go channel that connects over the Internet?
How would one specify the protocol to use to send messages?

A: A Go channel only works within a single program; channels cannot be
used to talk to other programs or other computers.

Have a look at Go's RPC package, which lets you talk to other Go
programs over the Internet:

  https://golang.org/pkg/net/rpc/

Q: What are some important/useful Go-specific concurrency patterns to know?

A: Here's a slide deck on this topic, from a Go expert:

https://talks.golang.org/2012/concurrency.slide

Q: How are slices implemented?

A: A slice is an object that contains a pointer to an array and a start and
end index into that array. This arrangement allows multiple slices to
share an underlying array, with each slice perhaps exposing a different
range of array elements.

Here's a more extended discussion:

  https://blog.golang.org/go-slices-usage-and-internals

I use slices often, and arrays never. A Go slice is more flexible than
a Go array since an array's size is part of its type, whereas a
function that takes a slice as argument can take a slice of any
length.

Q: What are common debugging tools people use for Go?

Q: fmt.Printf()

As far as I know there's not a great debugger for Go, though gdb can be
made to work:

https://golang.org/doc/gdb

In any case, for most bugs I've found fmt.Printf() to be an extremely
effective debugging tool.
```

Q: When is it right to use a synchronous RPC call and when is it right to use an asynchronous RPC call?

A: Most code needs the RPC reply before it can proceed; in that case it makes sense to use synchronous RPC.

But sometimes a client wants to launch many concurrent RPCs; in that case async may be better. Or the client wants to do other work while it waits for the RPC to complete, perhaps because the server is far away (so speed-of-light time is high) or because the server might not be reachable so that the RPC suffers a long timeout period.

I have never used async RPC in Go. When I want to send an RPC but not have to wait for the result, I create a goroutine, and have the goroutine make a synchronous Call().

Q: Is Go used in industry?

A: You can see an estimate of how much different programming languages are used here:

https://www.tiobe.com/tiobe-index/