Go Performance Tutorial

Josh Bleecher Snyder
Braintree/PayPal

Plan

- Introduction and philosophy
- Tools: Benchmarks, profiles
- Habits and techniques: string/[]byte, memory, concurrency
- Advanced tools and techniques
- Other kinds of optimization
- Wrap-up and stump the chump

10.10.32.101 (http://10.10.32.101)

Introduction and philosophy

Write simple, clear code

- Usually the fastest anyway

codereview.appspot.com/131840043 (https://codereview.appspot.com/131840043)

- Easy to see optimization opportunities
- Compiler and runtime optimized for normal code
- Take it easy on abstraction (reflection, interfaces)

"All problems in computer science can be solved by another level of ir
of course for the problem of too many indirections." - David Wheeler

Write good tests and use version control

Enables experimentation.

"If you're not going to get the right answer, I don't see the point. I can
fast if they don't have to be correct." - Russ Cox

Develop good habits

"Programmers waste enormous amounts of time thinking about, or w
the speed of noncritical parts of their programs, and these attempts a
actually have a strong negative impact when debugging and maintena
considered. We should forget about small efficiencies, say about 97%
premature optimization is the root of all evil. Yet we should not pass u
opportunities in that critical 3%." - Donald Knuth

Know thy tools, at all levels

- Can you cheat? Does it matter?
- Algorithms
- Language
- Benchmarking and profiling
- Machine and OS: Disk vs network vs memory

"People who are more than casually interested in computers should h
some idea of what the underlying hardware is like. Otherwise the pro
will be pretty weird." - Donald Knuth

The easiest wins around

- Use the most recent release of Go!
- Use the standard library.
- Use more hardware.

Benchmarking

Hello, benchmarks

Demo: package fib

Tour of testing.B and 'go test' flags

Demo: word length count

Comparing benchmarks

- benchcmp

```
go get -u golang.org/x/tools/cmd/benchcmp
```

- benchviz

```
go get -u github.com/ajstarks/svgo/benchviz
```

- benchstat

```
go get -u rsc.io/benchstat
```

Benchmarking concurrent code

Demo: ngram

Profiling

Hello, profiling

- Where have all the cycles gone?
- Support built into the runtime
- `go tool pprof`, graphviz
- OS X sadness

CPU profiling

Demo: fib

Memory profiling

Demo: ascii

Profiling gotchas

- Don't run multiple profilers at once.
- Don't run tests when profiling.
- If the output doesn't make sense, poke around or ask for help.

Block profiling

Demo: ngram

Other kinds of profiling

In package `runtime/pprof`:

- `goroutine`: helpful for finding sources of leaking goroutines
- `threadcreate`: helpful for debugging runaway thread creation (us
  cgo)

Basic memory stats available in package `runtime`: `ReadMemStats`

golang.org/pkg/runtime/#MemStats (https://golang.org/pkg/runtime/#MemStats)

Whole program profiling

Set up first thing in `func main`.

Use runtime and runtime/pprof packages...but it is a pain.

Dave Cheney made a nice helper package:

```
go get -u github.com/pkg/profile
```

godoc.org/github.com/pkg/profile (https://godoc.org/github.com/pkg/profile)

Monitoring live servers

Cheap enough to do in production!

And easy, using `net/http/pprof`.

```
import _ "net/http/pprof"
```

Use pprof to view CPU:

```
go tool pprof -pdf http://localhost:3999/debug/pprof/profile > o.pdf && open o
```

Heap:

```
go tool pprof http://localhost:3999/debug/pprof/heap
```

Goroutines:

```
go tool pprof http://localhost:3999/debug/pprof/goroutine
```

See `net/http/pprof` docs.

Monitoring live servers

Demo: present

localhost:3999/debug/pprof (http://localhost:3999/debug/pprof)

```
go tool pprof -pdf http://localhost:3999/debug/pprof/goroutine > o.pdf && oper
```

Oh goodness!

github.com/golang/go/issues/11507 (https://github.com/golang/go/issues/11507)

Protecting the net/http/pprof endpoints

`net/http/pprof` registers endpoints with `http.DefaultServeMux`.

So don't use `http.DefaultServeMux`.

```
serveMux := http.NewServeMux()
// use serveMux to serve your regular website

pprofMux := http.NewServeMux()
pprofMux.HandleFunc("/debug/pprof/", pprof.Index)
pprofMux.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
pprofMux.HandleFunc("/debug/pprof/profile", pprof.Profile)
pprofMux.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
// use pprofMux to serve the pprof handles
```

Or use a single non-default `ServeMux` but insert http handler middlew

Execution tracing

- New as of Go 1.5! Google for "Go execution tracer" to see the desig
- A few rough edges still.
- Incredibly detailed and powerful, with all the good and bad that th

Execution tracing

Demo: ngram

Techniques and habits

string and []byte

string and []byte

Common source of performance problems.
Easy to learn good habits.
Helps to know what's happening under the hood.

Under the hood

string

- basic type
- interpreted as UTF-8
- *immutable*

[]byte

- just another slice type
- no particular interpretation
- *mutable*

```
func set() {
    var b []byte
    b[0] = 0
    _ = b
}
```

Correct conversions are expensive

Above all, the compiler and runtime must be correct.
Speed is a bonus.

In the general case, converting between string and []byte requires an

```
func BenchmarkConvert(b *testing.B) {
    var s string
    buf := bytes.Repeat([]byte("abcdef"), 50)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        s = string(buf)
    }
    _ = s
}
```

Shrink and grow

string

- slicing is very cheap and safe
- concatenation is expensive (alloc + copy x 2)

[]byte

- slicing is very cheap but not obviously safe
- append is sometimes expensive (sometimes alloc, always copy x 1
  copy x 2)

Good habits

- Live in just one world (modulo code clarity and correctness).
- Convert as late as possible.
- Pay attention to concatenation, particularly in loops.

bytes.Buffer

Use a bytes.Buffer to build strings.

```
func BenchmarkConcat(b *testing.B) {
    for i := 0; i < b.N; i++ {
        var s string
        for j := 0; j < 100; j++ {
            s += "a"
        }
        _ = s
    }
}
func BenchmarkBuffer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        var buf bytes.Buffer
        for j := 0; j < 100; j++ {
            buf.WriteByte('a')
        }
        _ = buf.String()
    }
}
```

APIs

Use dedicated APIs:

- bytes and strings packages
- "io.Writer".Write vs io.WriteString
- "bufio.Scanner".Bytes vs "bufio.Scanner".Text
- "bytes.Buffer".Bytes vs "bytes.Buffer".String

Related: Implement WriteString for your io.Writers:

```
func WriteString(s string) (n int, err error)
```

Avoid building strings

If the set of choices is small, pick a string rather than building it. (Or use stringer: golang.org/x/tools/cmd/stringer.)

```
func BenchmarkConstruct(b *testing.B) {
    for i := 0; i < b.N; i++ {
        s = fmt.Sprintf("foo-%d", i%3)
    }
}
func BenchmarkPick(b *testing.B) {
    for i := 0; i < b.N; i++ {
        switch i % 3 {
        case 0:
            s = "foo-0"
        case 1:
            s = "foo-1"
        case 2:
            s = "foo-2"
        }
    }
}
```

Order of operations

Convert last (usually).

For example, slice after converting.

```
var s = strings.Repeat("abc", 100)
var p []byte

func BenchmarkSliceConvert(b *testing.B) {
    for i := 0; i < b.N; i++ {
        p = []byte(s[3:6])
    }
}
func BenchmarkConvertSlice(b *testing.B) {
    for i := 0; i < b.N; i++ {
        p = []byte(s)[3:6]
    }
}
```

If you're slicing multiple times, there are trade-offs: Multiple small allo
large monolithic chunk of memory.

Easy on the Sprintf

Use concatenation and strconv instead of fmt.Sprintf for simple things

```
func BenchmarkStrconv(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = strconv.Itoa(500)
    }
}
func BenchmarkSprintf(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fmt.Sprintf("%d", 500)
    }
}
```

API design

Design your APIs to allow reduced garbage.

- Provide []byte and string variants.
- Use io.Reader and io.Writer instead of buffers.
- BYO buffer.

Good:

```
Read(p []byte) (n int, err error)
```

Bad:

```
Read(n int) (p []byte, err error)
```

Techniques

- Reuse buffers.
- Take advantage of compiler optimizations.
- Intern strings.

Reuse buffers

```
var pool = sync.Pool{New: func() interface{} { return new(bytes.Buffer) }}
var p = bytes.Repeat([]byte{'a'}, 100)

func BenchmarkReuse(b *testing.B) {
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            buf := pool.Get().(*bytes.Buffer)
            buf.Write(p)
            _ = buf.String()
            buf.Reset()
            pool.Put(buf)
        }
    })
}
func BenchmarkNoReuse(b *testing.B) {
    b.RunParallel(func(pb *testing.PB) {
        for pb.Next() {
            var buf bytes.Buffer
            buf.Write(p)
            _ = buf.String()
        }
    })
}
```

Convert last

Pop quiz: How many allocs/op in this benchmark?

```
func BenchmarkConvert(b *testing.B) {
    p := bytes.Repeat([]byte{'a'}, 10)
    var n int
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        s := string(p)
        n += len(s)
    }
    _ = n
}
```

Compiler magic

Conversion optimizations in Go 1.5 include:

- map keys
- range expressions
- concatenation
- comparisons

Convert as late as possible to enable them to work.
(Future work may change that.)

More are possible. Those that work well on normal code may eventua
implemented.

Map keys

The map key optimization is particularly interesting.

```
var p = bytes.Repeat([]byte{'a'}, 100)
var m = make(map[string]bool)

func BenchmarkMapKey1(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = m[string(p)]
    }
}
func BenchmarkMapKey2(b *testing.B) {
    for i := 0; i < b.N; i++ {
        s := string(p)
        _ = m[s]
    }
}
```

Interning strings

```
var interned = make(map[string]string)

func intern(b []byte) string {
    s, ok := interned[string(b)] // does not allocate!
    if ok {
        return s
    }
    s = string(b)
    interned[s] = s
    return s
}

var p = bytes.Repeat([]byte{'a'}, 100)

func BenchmarkConvert(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = string(p)
    }
}
func BenchmarkIntern(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = intern(p)
    }
}
```

Caution

Be careful with interning!

- Advanced technique. Use with caution and only when necessary.
- Depends on compiler version.
- **Manual memory management!** Ewwwwww.
- Not thread safe. (But see github.com/josharian/intern for a hack.)

Optimizing memory usage

What is allocation?

Making a place to put stuff.

- You can't avoid all allocation. That's ok!
- Why it matters: allocation, zeroing/copying, GC, limited resource, ir
- Number of allocations vs size of allocation.

What allocates?

Lots of things, but it varies by compiler.
In practice, there are no strict rules.

Common sources of allocations are:

- Data growth (append, concatenation, map assignment, stacks)
- new, make, and &
- string/[]byte conversion
- Interface conversions
- Closures

Develop good habits, profile, and benchmark.

Escape analysis

- Heap vs stack
- Subtle, interacts with growable stacks and GC
- Stack pressure vs heap
- `-gcflags=-m`

Mostly, just know that it exists and what it is.

Good habits

- Avoid unnecessary data growth.
- Avoid unnecessary string/[]byte conversions.
- Design APIs that allow re-use.
- Use values where you can.
- Avoid gratuitous boxing, reflection, and indirection.

Unnecessary data growth

Buffer:

```
buf, err := ioutil.ReadAll(r)
// check err
var x T
err = json.Unmarshal(buf, &x)
// check err
```

Stream:

```
dec := json.NewDecoder(r)
var x T
err := dec.Decode(&x)
// check err
```

Unnecessary/deep recursion

Stacks take memory too. Stack growth is an alloc+copy+process.

(Most data growth is alloc+copy.)

API design

Use io.Reader and io.Writer.

Also, io.Reader is a fine example itself!

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

It is hard to anticipate your users' needs. Give them the tools to be eff
it.

Use values

Value:

```
type OptBool uint8

const (
    Unset = OptBool(iota)
    SetFalse
    SetTrue
)
```

Pointer:

```
type OptBool *bool
```

But take care with large values.

```
var a [10000]int{}
for _, i := range a {
}
fmt.Println(a)
```

Go easy on the abstraction

- Reflect allocates heavily.
- Most interface conversions allocate.
- Creating closures usually allocates.

Techniques

- Provide initial capacity estimates for data structures.
- Trade off allocation size and number of allocations.
- Reuse objects. Maintain a free list or use sync.Pool.
- Steal ideas from the standard library.

Initial capacity estimates

Delayed/multiple allocs vs exactly one alloc

```
const size = 100

func BenchmarkDelayedAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        var s []int
        for i := 0; i < size; i++ {
            s = append(s, i)
        }
        sink = s
    }
}
func BenchmarkOneAlloc(b *testing.B) {
    for i := 0; i < b.N; i++ {
        s := make([]int, 0, 100)
        for i := 0; i < size; i++ {
            s = append(s, i)
        }
        sink = s
    }
}
```

Pre-allocate backing array

```
type Buffer struct {
    buf       []byte
    off       int
    runeBytes [utf8.UTFMax]byte
    bootstrap [64]byte
    lastRead  readOp
}
```

runeBytes avoids allocation during WriteRune:

```
utf8.EncodeRune(b.runeBytes[0:], r)
```

bootstrap avoids allocation for small buffers:

```
b.buf = b.bootstrap[0:]
```

Reuse objects

Local re-use is better:

```
buf := make([]byte, 1024)
for {
    n, err := r.Read(buf)
    // use err, n, buf
    // look out: buf's contents will be overwritten in the next Read call
}
```

Sometimes there's no context (type or scope) to allow re-use. Enter sy

```
for {
    buf := pool.Get().([]byte)
    // use buf
    // optional: clear buf for safety
    for i := range buf {
        buf[i] = 0
    }
    pool.Put(buf)
}
```

Struct layout

Go guarantees struct field alignment.

```
type Efficient struct {
    a interface{}
    b *int
    c []int
    d uint16
    e bool
    f uint8
}

type Inefficient struct {
    e bool
    a interface{}
    f uint8
    b *int
    d uint16
    c []int
}

var e Efficient
var i Inefficient
fmt.Println(unsafe.Sizeof(e), unsafe.Sizeof(i)) // 28 36
```

Struct layout

cmd/wasted:

golang.org/cl/2179 (https://golang.org/cl/2179)

Unlikely to happen automatically:

golang.org/issue/10014 (https://golang.org/issue/10014)

"No Go compiler should probably ever reorder struct fields. That seem
to solve a 1970s problem, namely packing structs to use as little space
2010s problem is to put related fields near each other to reduce cache
(unlike the 1970s problem) there is no obvious way for the compiler to
solution. A compiler that takes that control away from the programme
that much less useful, and people will find better compilers." - Russ Co

Optimizing concurrent programs

Optimizing concurrent programs

Concurrency correctness is hard, even with Go.

Habits

- Use mutexes instead of channels for simple shared state.
- Minimize critical sections.
- Don't leak goroutines.
- Gate access to shared resources, particularly the file system.

Mutexes and channels

Mutexes are good for mutual exclusion, like simple shared state. They
simple in such cases.

Channels are for everything else: Flow control, communication, coord

Minimize critical sections

Separate work that requires shared state from work that does not. Only hold the lock when you really need it. Refactor as needed.

Before:

```
func (t *T) Update() {
    t.Lock()
    defer t.Unlock()
    // expensive work that can be done independently
    // update shared state
}
```

After

```
func (t *T) Update() {
    // expensive work that can be done independently
    t.Lock()
    defer t.Unlock()
    // update shared state
}
```

Don't leak goroutines

When you start a new goroutine, pause to ask when it will/how it will c

```
func doh(c chan int) {
    go func() {
        for i := range c {
            // use i
        }
    }()
    // who closes c? who calls doh?
}
```

Goroutines are so cheap you might not notice leaks quickly.

Profile or manually inspect the result of a SIGQUIT.

Gate access to shared resources

It's easy to thrash the filesystem, make lots of threads, and create chu
scheduler. It's also easy to prevent.

```
type gate chan bool

func (g gate) enter() { g <- true }
func (g gate) leave() { <-g }

type gatefs struct {
    fs vfs.FileSystem
    gate
}

func (fs gatefs) Open(p string) (vfs.ReadSeekCloser, error) {
    fs.enter()
    defer fs.leave()
    // ...
    return gatef{file, fs.gate}, nil
}

var fsgate = make(gate, 8)
```

Use buffered I/O

Every read or write to a file corresponds to a system call. These are re
expensive, particularly in high numbers.

The `bufio` package makes buffered I/O easy. Use it.

```
f, err := os.Open("abc.txt")
// handle err
r := bufio.NewReader(f)
```

Techniques

- sync.RWMutex is only sometimes better than sync.Mutex.
- Use buffered channels.
- Provide backpressure or dropping.
- Partition shared data structures.
- Batch work to amortize cost of lock acquisition.
- Use sync/atomic.
- Cooperate with the scheduler
- Avoid false sharing by padding data structures.

sync.RWMutex vs sync.Mutex

sync.RWMutex does strictly more work than sync.Mutex and has mor
semantics.

sync.RWMutext can help a lot, but it can also hurt. Profile and/or benc

Use buffered channels

Buffered and unbuffered channels have different semantics and sync
guarantees.

Buffered channels are much cheaper, if both semantics work for you.

Provide backpressure or dropping

Critical for operational stability of distributed services, but also useful

Partition shared data structures

Before:

```
type Counter struct {
    mu sync.Mutex
    m  map[string]int
}
```

After:

```
const shards = 16

type Counter struct {
    mu [shards]sync.Mutex
    m  [shards]map[string]int
}
```

Can reduce contention.

Adds cost of hashing, increases data structure size, and depends on d
data. Measure with real world data. `rand.Zipf` can be helpful for ber

Batch work

```
// consumer
var sum int
for i := range c {
    sum += i
}

// producer before
for !done {
    sum := count(stuff)
    c <- sum
}

// producer after
for !done {
    sum := 0
    for i := 0; i < 16; i++ {
        sum += count(stuff)
    }
    c <- sum
}
```

Can dramatically reduce contention, but not always applicable. Can in
due to batching.

atomic.Value

```
var (
    configmu    sync.Mutex    // protects configvalue
    configvalue *atomic.Value // value of map[string]string
)

func config() map[string]string {
    return configvalue.Load().(map[string]string)
}

func set(key, val string) {
    configmu.Lock()
    defer configmu.Unlock()
    old := config()
    m := make(map[string]string, len(old)+1)
    for k, v := old {
        m[oldk] = oldv
    }
    m[k] = v
    configvalue.Store(m)
}
```

For frequently read but infrequently written data structures. Requires
and writer synchronization (or a single writer). Danger of logical races.

atomic int and pointer operations

```
var count uint32

func inc() {
    atomic.AddUint32(&count, 1)
}

func get() uint32 {
    return atomic.LoadUint32(&count)
}
```

Cheapest form of concurrency-safety available in Go. Great caution re
to misuse in subtle ways!

Mostly helpful for cheap, scalable counters.

If you use atomic.* with a value anywhere, you must use it everywher

Extra special care required when using 64 bit integer sizes on 32 bit pl
alignment requirements.

Cooperative scheduling

Go scheduling is currently cooperative. It mostly just works, except for
no function calls.

```
var x int
for i := 0; i < 1<<30; i++ {
    x = x ^ i
}
```

Solution: Use runtime.Gosched()

```
var x int
for i := 0; i < 1<<30; i++ {
    x = x ^ i
    if i & 0xFFFF == 0xFFFF {
        runtime.Gosched()
    }
}
```

Avoid false sharing

Usually solvable by rearrangement or padding.

```
type T [1024]Padded

type Padded struct {
    mu sync.Mutex
    x  *X
    _  [128]byte
}
```

Diagnose first; the medicine is bitter.

Advanced tools and techniques

Advanced tools and techniques

- Compiler flags
- Runtime flags and calls
- Assembly and cgo
- Code generation
- Micro-optimizations

Compiler flags

Use:

```
go build -gcflags=-S pkg
```

Or:

```
go tool compile -S a.go b.go c.go
```

Important flags:

```
-h    help
-S    print assembly listing
-m    print optimization decisions such as escape analysis
-l    turn off inlining, repeat to make inlining more aggressive
-N    disable optimizations
-B    disable bounds checking
```

Demo

GODEBUG and GOGC

Sample GODEBUG use:

```
GODEBUG=scheddetail=1,schedtrace=1000 go run x.go
```

Useful GODEBUG variables for performance investigation:

```
allocfreetrace=1: print all allocs and frees (it's a lot!)
gctrace=1, gctrace=2: print GC activity
schedtrace=X: print scheduler state every X ms
scheddetail=1: print detailed scheduler state
```

Sample GOGC use:

```
GOGC=off go run x.go
```

Or:

```
runtime.SetGCPercent(-1) // -1 for off, 50 for aggressive GC, 100 for default
```

cgo

Plenty of rope.

- overhead: stack switch and calling convention change
- takes up a thread
- medium chunks of work
- cross-compilation is not trivial

Assembly

Upgrade from rope to gun.

- overhead: function call
- dangerous
- basically undocumented
- small chunks of work (but not individual instructions)
- not subject to Go 1 guarantee
- go vet is helpful

Useful when there is no other way.

Code generation

Helpful for:

- Avoiding the need for an abstraction layer (yes yes, generics)
- Unrolling loops or calculations
- Generating pre-calculated tables
- Generating efficient code that is hard to read or maintain

```
const _Num_name = "OneTwo"

var _Num_index = [...]uint8{0, 3, 6}

func (i Num) String() string {
    if i < 0 || i+1 >= Num(len(_Num_index)) {
        return fmt.Sprintf("Num(%d)", i)
    }
    return _Num_name[_Num_index[i]:_Num_index[i+1]]
}
```

Micro-optimizations

Knuth alert!

- arrays instead of maps for lookups with small integer keys

```
var a = [10]string{2: "even prime", 9: "maybe prime"}
var m = map[int]string{2: "even prime", 9: "maybe prime"}
```

- slice instead of map for very small quantities of data
- manually unwind instead of using defer
- index into slice instead of pointer in giant data structures

Compiler dependent:

- optimized memclear
- rotate instruction: `i<<13 | i>>(64-13)`

Other kinds of optimization

- Binary size
- Build time

Binary size

go tool nm

Helpful for finding large static data:

```
package main

var a [100000]int

func main() {
    _ = a[0] // prevent the linker from dropping a
}
```

Result:

```
$ go build bigarray.go && go tool nm -size -sort=size bigarray | head -n 4
   d47c0     800000 B main.a
   86140     152299 R runtime.pclntab
   4d220     124312 T runtime.etext
   4d220     124312 R type.*
```

Millions of strings

The only way to get millions of strings is to generate them. If you're ge
many, generate them as a single string and slice as needed.

```
var nums = [...]string{"0", "1", "2", "3", "4", ..., "99999"}
```

Binary size: 3471840 bytes

```
var nums = "0123...99999"
```

Binary size: 1558960 bytes

ldflags

Normal:

```
$ go build helloworld.go && stat -f "%N: %z bytes" helloworld
helloworld: 2344944 bytes
```

Without DWARF:

```
$ go build -ldflags=-w helloworld.go && stat -f "%N: %z bytes" helloworld
helloworld: 1746928 bytes
```

But you lose debug information.

Build time

go install

`go build` builds and discards.
`go install` build and keeps.

Use `go install`.

Enable caching of stable code

The unit of compilation is the package.

If you have a large, stable chunk of code (frequently a generated file c
put it in a different package than high churn code. Use internal packag
worried about API visibility.

Split up giant functions

This oughtn't matter. It does.

Giant static data and tables can generate giant functions. (See previou

One hack: Use multiple init functions.

Wrap-up

Want more?

- Read the standard library
- Blog posts by Dmitry Vyukov and Russ Cox
- Lurk on golang-codereviews@googlegroups.com (or even better, c
- Ask questions on golang-nuts
- Experiment!

Experiment where?

- Profile your own code. (But know when to stop.)
- Pick an open source project. Find and fix a significant performance
  but the largest projects, there's usually at least one.)
- Futz around with the standard library. (But remember that clarity a
  maintainability trumps speed.)

Reminders

- Write simple, clear code
- Write tests and use version control
- Cheat (solve an easier problem instead)
- Develop good habits
- Know your tools and use them

Stump the chump

Thank you

Josh Bleecher Snyder
Braintree/PayPal
josharian@gmail.com (mailto:josharian@gmail.com)
@offbymany (http://twitter.com/offbymany)