10 things

I wish I knew about

Code efficiency

as a junior dev

This is a talk about code efficiency

> I am not an expert

> But I got this far without needing to be one

> Improving efficiency is 50% technical knowledge, and 50% problem solving approaches

What is code efficiency?

> Low latency

> Low resource usage

> High throughput

> absolutely nothing to do with how many lines of code you write

1 >>

Don’t worry about it

unless you have to

1 >>
Don’t worry about it

> You have my full permission not to use anything you learn in this talk, for, like, months

> You’ve written tonnes of code, and almost all of it has been good enough

> Make it work. Make it right. Make it fast

> If it’s a problem, you’ll know

> If you think it might be a problem in the future:

>> have a way to measure lag

>> load test your system

>> follow good, OOP practices

2 >>

Find the bottleneck

20% of the code takes 80% of the processing power

2 >>

Find your bottleneck

> It doesn’t matter how wide the rest of the bottle is, the neck is what limits the flow

> Computer are really fast

> If they aren’t, it’s probably a specific problem.

> Don’t waste time trying to make the rest of the bottle wider, fix the neck

2 >>
Find your bottleneck

> How to find a bottleneck

>> be an expert

>> metrics, logs, and jstack

>> code profilers

>> it might not be your code. Networks and shared resources can be the slowest part

3 >>

Learn how to estimate efficiency

It’s not hard

3 >>
Learn how to estimate efficiency

(Number of times you need to do X) * (Amount of work it takes to do X)

> Heavy processes that run occasionally are fine

> Cheap processes you run a lot are fine

> Heavy processes you run all the time are not fine

> How to calculate how much work it takes to do X?

3 >>

Learn how to estimate efficiency

Efficiency of X = number of steps it takes to do X

> Steps, not seconds

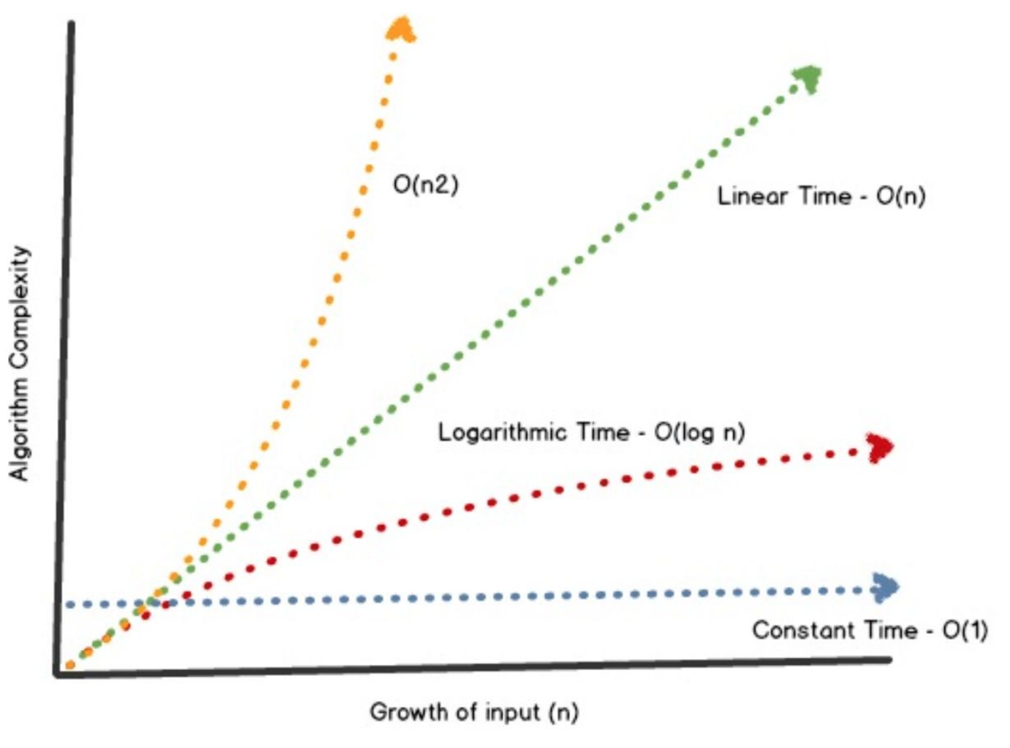
> The number of steps isn’t always obvious

> The number of steps isn’t always constant

> Understand what influences the number of steps, and how

3 >>

Learn how to estimate efficiency



3 >>

Learn how to estimate efficiency

> Pop quiz! How efficient are these?

Given that `ArrayList x = [5, 11, 9243, 6...]`

```
X.get(50)

sumList(x) //add all the elements in x together

X.contains(22)

x.containsAll(y)

x.sort()

What if x was sorted?
```

3 >>
Learn how to estimate efficiency

> What about space?

> Easy!

> get a small sample

> get a volume estimate

> multiply them together

4 >>

Learn the little hacks for your primary coding language

Because it’s easy to do

4 >>

Learn the little hacks for your primary coding language

```
list.contains(x)

Vs

set.contains(x)
```

4 >>

Learn the little hacks for your primary coding language

```
logger.debug(“Processing resource: {}”, resource}

vs

logger.debug(“processing resource: “ + resource.toString()}
```

4 >>

Learn the little hacks for your primary coding language

```
logger.debug(“Processing resource: {}”, resource}

vs

logger.debug(“processing resource: “ + resource.toString()}
```

4 >>

Learn the little hacks for your primary coding language

```
if (i == 0 || expensiveFunction())
```

vs

```
if (expensiveFunction() || i == 0)
```

4 >>

Learn the little hacks for your primary coding language

```
Integer i = 1
```

vs

```
int i = 1
```

4 >>

Learn the little hacks for your primary coding language

> use caches were appropriate

> write lazy code

> avoid nested lists

> avoid regexes

> Use string builders for concatenation

> Define collection size at creation

> check hubot comment explanations

5 >>

Learn the deep stuff, too

Because there’s no substitute for actual understanding

5 >>

Learn the deep stuff, too

> This isn't a lecture on Java memory management

> This is me telling you to go to lecture on Java memory management

> It takes a lot of time and effort to learn, but it's worth it

> So much of efficiency is down to implementation details

> If you don't have a basic understanding of how things work under the hood, it can be hard to even follow conversations about efficiency

5 >>
Learn the deep stuff, too

>> What is the JVM and how does it work?

>> How is memory allocated? How much memory does your app have, and how is being assigned? How can you tune this?

>> What is garbage collection? When does it happen, and what does it do? Why is too much of it bad?

>> what are memory leaks and how to find them? What about GC thrashing? Fragmented memory?

>> what are threads? How are they managed? What states can they be in, and what do these states mean? How many threads is best?

>> what are the java primitive types, and what are their sizes? How are they different to objects? How are common collections implemented?

6 >>

Multi-threading and multiple instances provide infinite* scaling

But it comes with a cost

6 >>

Multi-threading and multiple instances provide infinite* scaling

> Multi-threading allows your app to execute many pieces of code at the same time

> It’s limited by the number of processors on you machine. Our live servers can ~80 processors

> Multiple instances allow horizontal scaling. Want more power? Just deploy another instance on a new machine

> Combine these two, and you can process truly huge amounts of data

> But there’s a cost

6 >>
Multi-threading and multiple instances provide infinite* scaling

> concurrency is hard

>> Non-linear code is harder to understand

>> Non-deterministic bugs are harder to detect and diagnose

>> It’s not always obvious when you’ve made a serious mistake

> Multiple instances of an app are hard (if they need to coordinate with each other)

> And multiple instances of an app have another cost...

6 >>

Multi-threading and multiple instances provide infinite* scaling

> Hardware

> It isn’t free

> You aren’t increasing efficiency, you’re increasing throughput by throwing money at the problem

> Horizontal scaling is amazing and necessary, but it can eat all your profit if you let it

> example: twitter firehose

7 >>

Consider redesigning the algorithm

It’s easier than micro-optimising code

7 >>
Consider redesigning the algorithm

> Rewriting is sometimes better than improving what you have

> Potential for massive performance gains

> Examples

>> prematched query ids on tweets

>> mnemosyne

>> timestamping database entries when polling for differences

> Requires you to really understand your problem space AND have a good idea

8 >>

If all else fails, find an expert

It’s going to get technical

8 >>
If all else fails, find an expert

> Micro-optimising code is a lot of work for not much gain, and it makes code less readable

> But if you really understand what your code is doing, and really, really need it to be faster, there will be a way

> Sometimes it’s as simple as changing an int to a short

> Sometimes it’s as crazy as changing WHERE IN(...) to WHERE NOT NOT IN(...)

> In time, you can become the expert you need

> but you’ll never be an expert in everything

9 >>

Measure your “efficiency enhancement”

Don’t just assume it worked

9 >>

Measure your "efficiency enhancement"

> Efficiency is really complicated. Things that "should work" might not

>> multithreading can slow down your processes

>> exponential complexity can be faster than constant

>> improvements outside your bottleneck might not make any difference

> Run timed comparison between your branch and the master branch

10 >>

Be mindful of external systems

Your apps efficiency isn’t the only one that matters

10 >>
Be mindful of external systems

> Just because your app can keep up with the load doesn’t mean our whole system can

> External systems might be servicing 100s of requests just like yours, and the impact adds up

> It can be easy to overlook or not notice your impact on other systems

>> Databases

>> Networks

>> Services

>> Downstream processes

Recap

> If you don’t have performance problems, don’t worry about it

> If you do have performance problems, focus on finding and fixing the bottle neck

> Picking up some small efficiency hacks is easy, but has limited impact

> Learning how stuff works under the hood is pretty important

> Redesigning your algorithm can be better than trying to improve what you have

> Multithreading and horizontal scaling allow massive scaling

> hardcore black magic is also an option, but I won’t recommend it