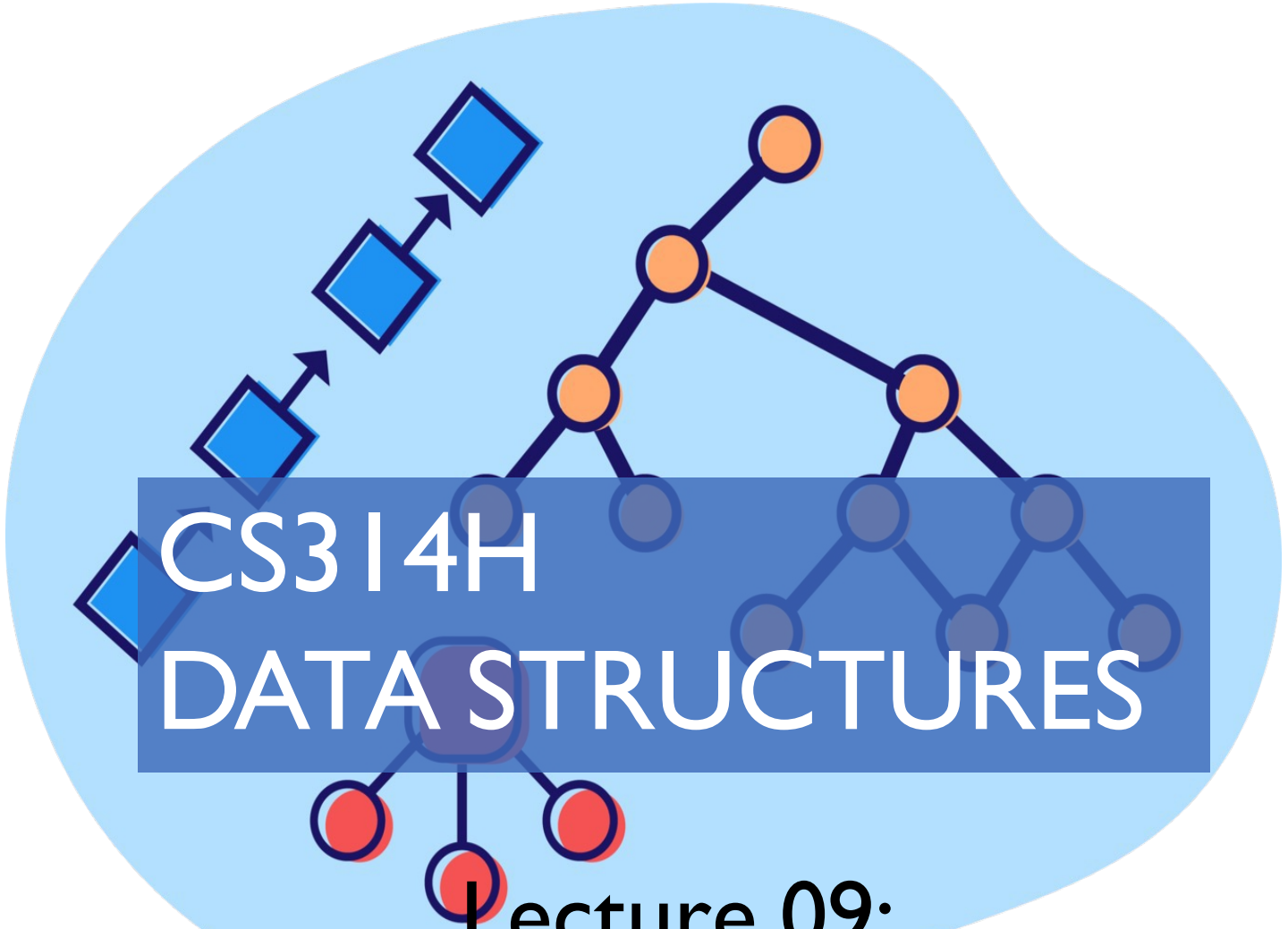


CS314H
DATA STRUCTURES

Lecture 09:
Algorithm Analysis

Mikyung Han

Please, interrupt and ask questions AT ANY TIME !

Reminders

- Read Ch 5, 6-6.8 if you haven't done so
- Prog 4 Part 1 submission due Friday 26 9 PM

Outline

1. Administrative
2. Comparing programs

How to compare two program??

- `hasDuplicate`
- Given a sorted int array determine if the array has a duplicate

Any idea?

- Algorithm 1: For each pair of element, check if they are the same
- Algorithm 2: For each element, check if it's equal to the one after it

First approach:
Can we measure how long each program takes to run?

Can we time the program?

- **Timing programs is prone to error (not reliable or portable):**
  - Hardware: processor(s), memory, etc.
  - OS, Java version, libraries, drivers
  - Other programs running
  - Implementation dependent

- **Can we even time an algorithm? What if it is intractable?**
  - Algorithm that cannot be solved in polynomial time

How about counting number of steps?

```java
public int stepsHasDuplicate1(int[] array) {
    int steps = 0;
    for (int i=0; i < array.length; i++) {
        for (int j=0; j < array.length; j++) {
            steps++; // The if statement is a step
            if (i != j && array[i] == array[j]) {
                return steps;
            }
        }
    }
    return steps;
}
```

OUTPUT

```
>> hasDuplicate1 average number of steps is 9758172 steps.
>> hasDuplicate2 average number of steps is 170 steps.
```

- What is above code depending upon?

We must do this via testing

Comparing ~~programs~~ algorithms

Why?

- Abstract out implementation detail
- Independent of CPU speed, programming languages, versions etc..
- Can do analysis before coding
- Enables us to study scalability

When analyzing algorithms,
we count number of operations

Operation? Didn't we just say we don't do # of steps?

An operation is any action whose cost does NOT depend on the input size

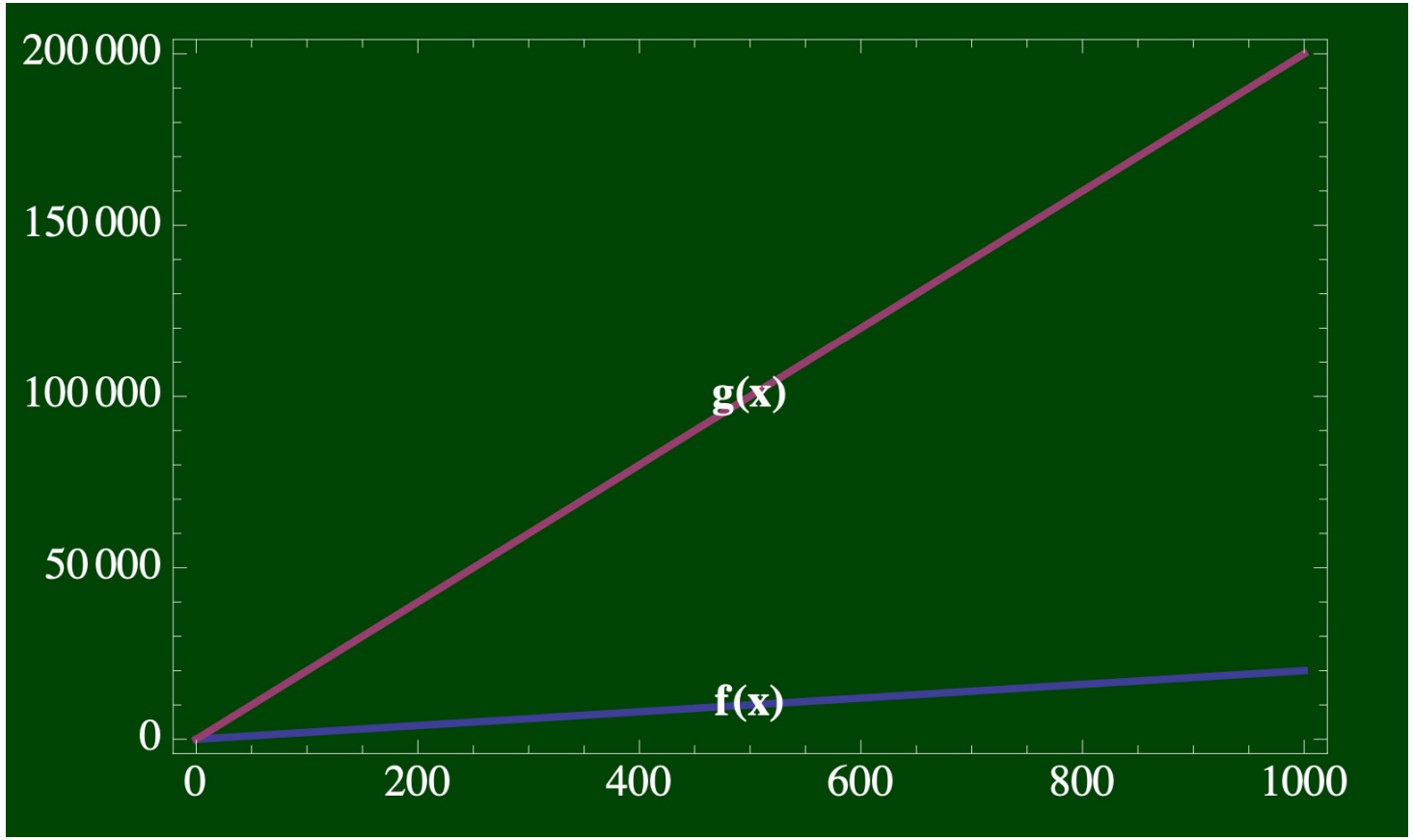
```
proc analyzeMe(list L) {
   L[0] = 3;
   int x = length(L); //read L.size
   sort(L);  //is this an operation?
   Return L[floor(x/2)];
}
```

We define time complexity T(n) as a function of input size n

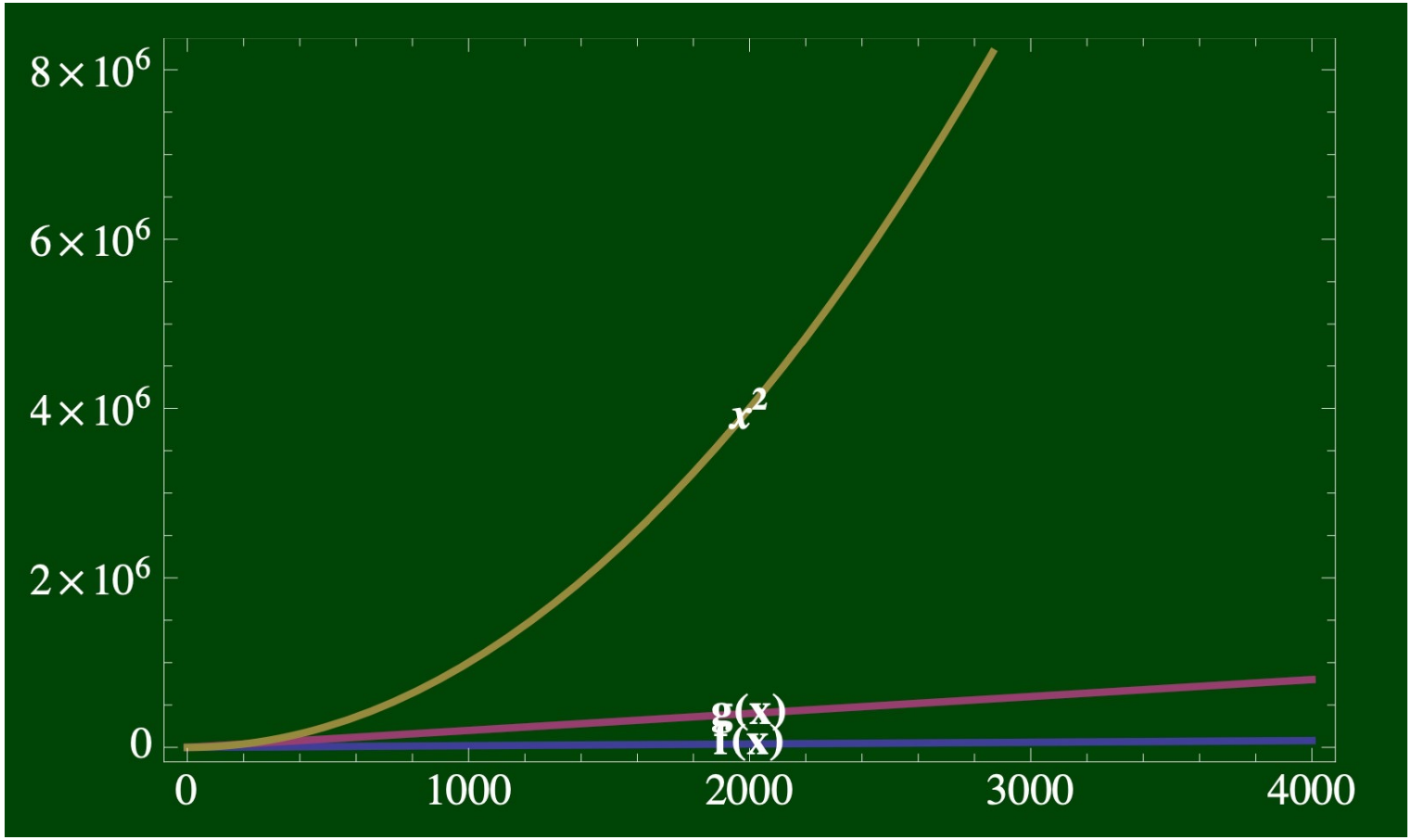
- We ignore constants!

Why is it ok to ignore constant?

Should be consider these “same”?

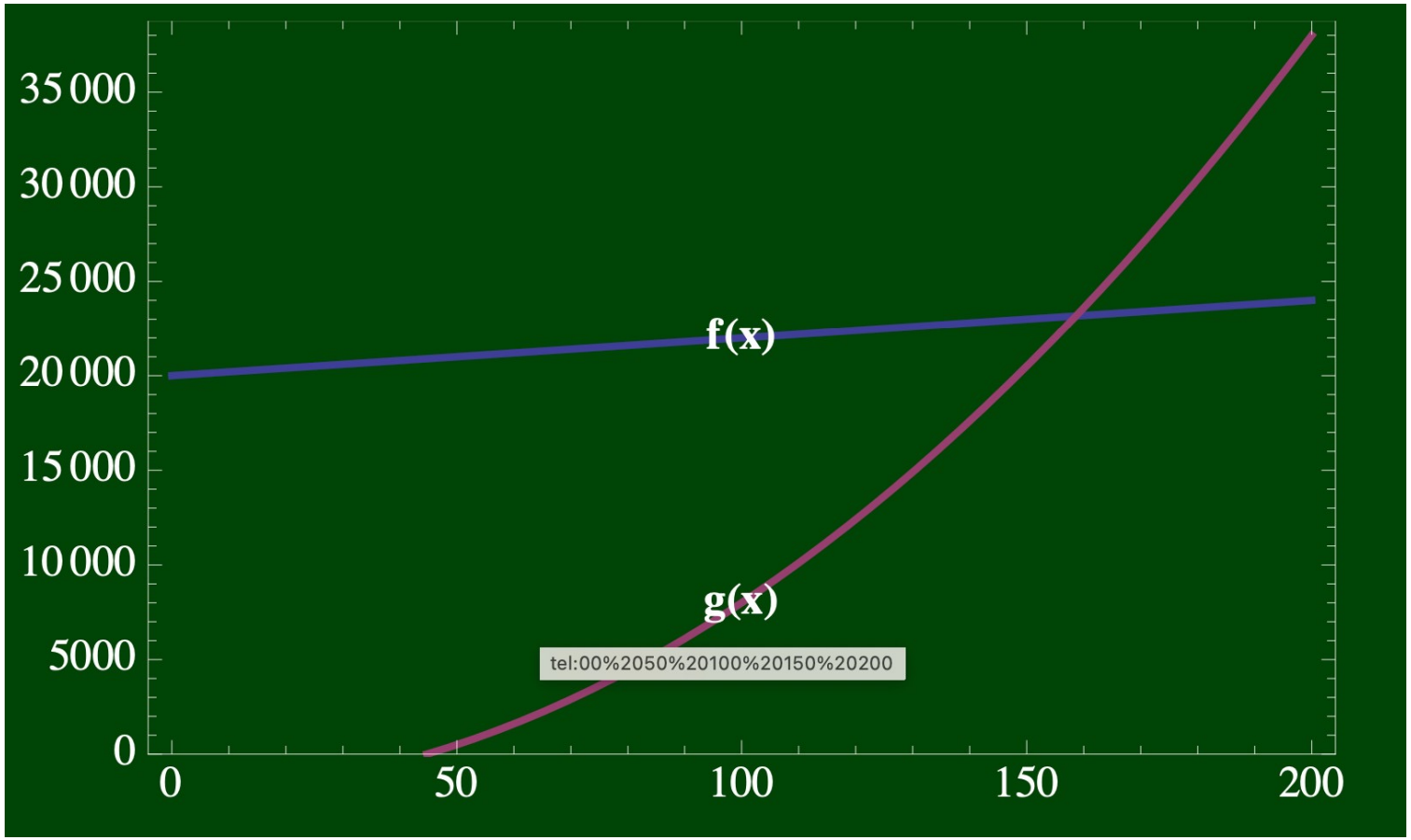


Probably yes
since they seem to grow at the same rate



We ignore constant because it doesn't affect the growth rate

Why care about large input?



With small input any algorithm could work well

Comparing algorithms

- Usually focus on time and space
- We usually consider the worst case (unless specified otherwise)
- We only consider large inputs
- We like to discuss them in bounds

Outline

1. Administrative
2. Generics
3. Comparing programs
4. Asymptotic Analysis

Asymptotics is an analysis of the behavior as the input size grows to infinity

Asymptotics

- We like to compare two functions
- If we have f and $4f$, consider them the same

$f \leq g$ when...

$f \leq cg$ where c is a constant and $c \neq 0$.

- We care about all values of the function that are big enough

$f \leq g$ when...

For all n "large enough", $f(n) \leq cg(n)$, where $c \neq 0$
For some $n_0 \geq 0$, for all $n \geq n_0$, $f(n) \leq cg(n)$, where $c \neq 0$
For some $c \neq 0$, for some $n_0 \geq 0$, for all $n \geq n_0$, $f(n) \leq cg(n)$

Big-O Definition

Definition (Big-Oh)

We say a function $f : A \to B$ **is dominated by** a function $g : A \to B$ when:

$$\exists (c, n_0 > 0).\ \forall (n \geq n_0).\ f(n) \leq cg(n)$$

Formally, we write this as $f \in \mathcal{O}(g)$.

Big-Oh Gotchas

- $\mathcal{O}(f)$ is a **set**! This means we should treat it as such.
- If we know $f(n) \in \mathcal{O}(n)$, then it is also the case that $f(n) \in \mathcal{O}(n^2)$, and $f(n) \in \mathcal{O}(n^3)$, etc.
- Remember that small cases, really don't matter. As long as it's **eventually** an upper bound, it fits the definition.

Big-O Examples

True or False?

(1) $4+3n \in \mathcal{O}(n)$

(2) $4+3n = \mathcal{O}(1)$

(3) $4+3n$ is $\mathcal{O}(n^2)$

(4) $n+2\log n \in \mathcal{O}(\log n)$

(5) $\log n \in \mathcal{O}(n+2\log n)$

Big-O Examples

prove $4 + 3n + 4n^2 \in \mathcal{O}(n^3)$

Big Omega for lower bound

Definition (Big-Omega)

We say a function $f : A \to B$ **dominates** a function $g : A \to B$ when:

$$\exists(c, n_0 > 0).\ \forall(n \geq n_0).\ f(n) \geq cg(n)$$

Formally we write this as $f \in \Omega(g)$.

Why/when is it useful to have lower bound?

Big Theta for tight bound

Definition (Big-Theta)

We say a function $f : A \to B$ **grows at the same rate as** a function $g : A \to B$ when: $f \in \mathcal{O}(g)$ and $f \in \Omega(g)$
Formally we write this as $f \in \Theta(g)$.

No need to use the same c, n_o values for O and Ω to prove

If you want to say "f is a tight bound for g" use Θ not O

Note we are analyzing the worst case time

What else can we analyze?

- Space
- Average case
- Best case
- Time over multiple operations
  - Amortization analysis

Worst case vs Average case vs Best case

- Best case: T(n) is the run time for the best-case input of size n
- Worst case: T(n) is the run time for the worst-case input of size n
- Average case: T(n) is the average run time, over all inputs of size n
  - Note in reality it's possible that not all inputs are equally likely.

Is best case the lower bound and the worst case the upper bound?

- No!
  - There can be the lower/upper/tight bound for the best case
  - There can be the lower/upper/tight bound for the worst case
  - There can be the lower/upper/tight bound for the average case

- **Worst case lower bound**
  - Comparison-based sorting is $\Omega(n \log(n))$ in the worst case.
- **Best case upper bound**
  - Insertion sort on already sorted list $\Omega(n)$

BTW, is $\log_2$ and $\log_{10}$ the "same"?

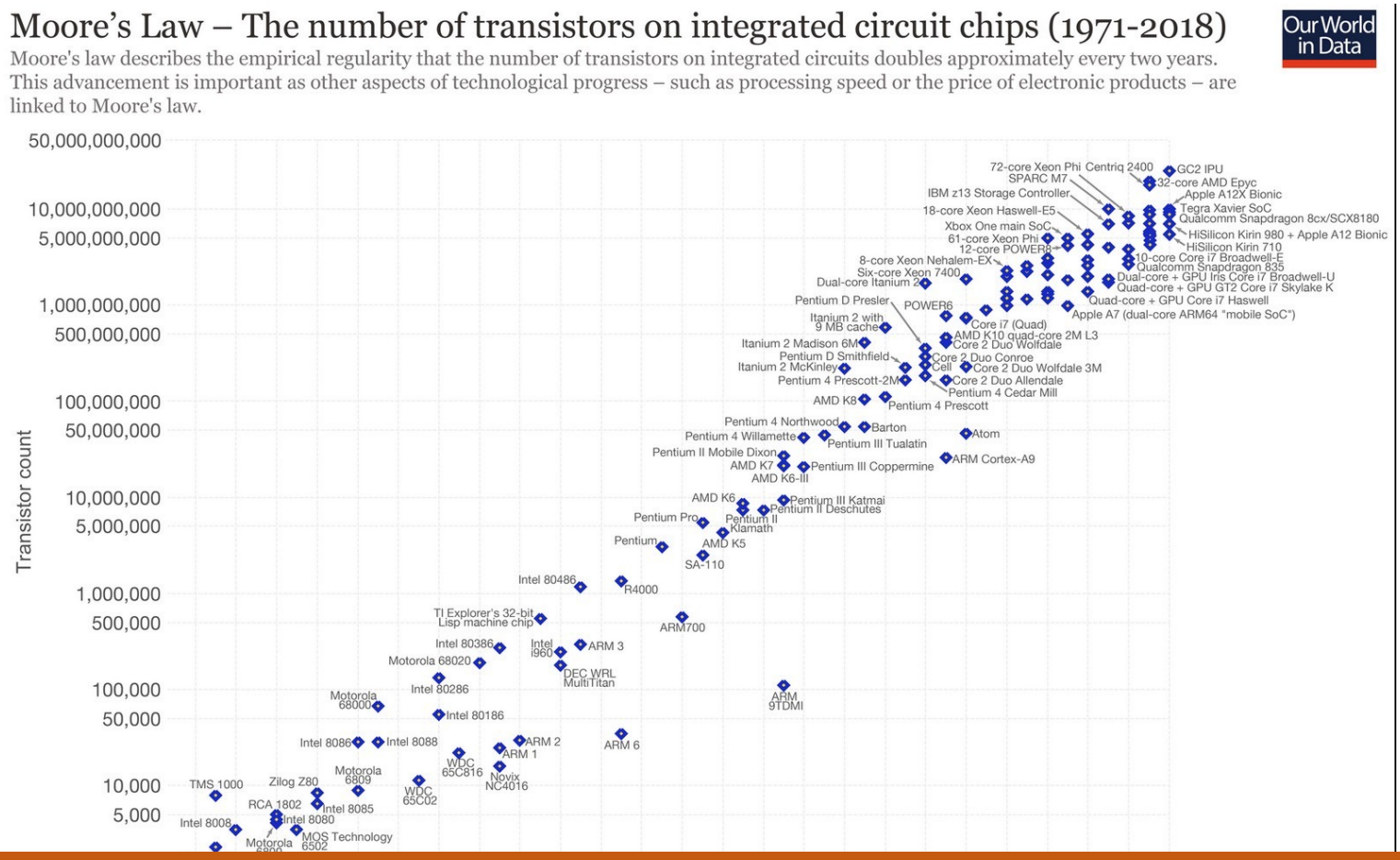
$$\log_b(x) = \frac{\log_d(x)}{\log_d(b)}$$

In asymptotic analysis, all log bases are the same

Outline

1. Administrative
2. Generics
3. Comparing programs
4. Asymptotic Analysis
5. Does it really matter?

Moore's law vs asymptotic analysis



Since computers are doubling in speed every year,
if I just wait then wouldn't inefficient algorithms run fast enough?

The Tyranny of Growth rate

T(n)	Max problem size for 1000 cycles	Max problem size for 10000 cycles	Increase in problem size
$100n$	10	100	10x
$5n^2$	14	45	3.2x
$0.5 \times n^3$	12	27	2.3x
2^n	10	13	1.3x

As computers become more powerful, asymptotics matter even more!
We need to solve larger and more complex problems