CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

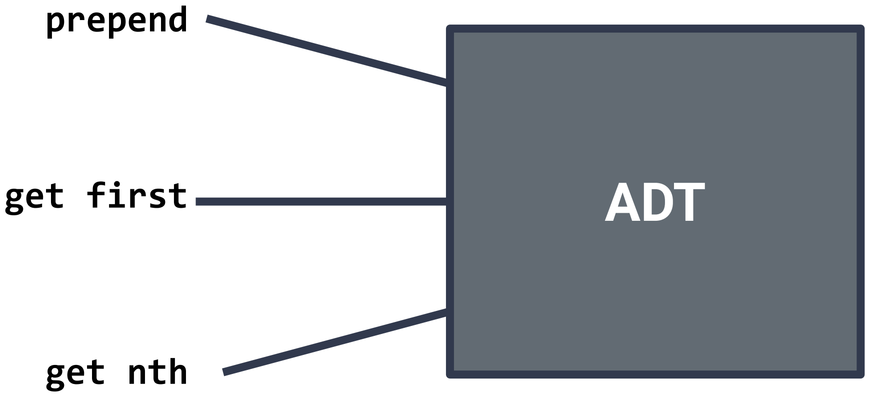
Lec 04: Intro to Complexity

Announcements and Feedback

- Office hours start this week
- Normal recitations begin next week
- Academic Integrity Quiz due Tonight @ 11:59PM
- PA0 due Friday @ 11:59PM
- WA1 due Friday @ 11:59PM

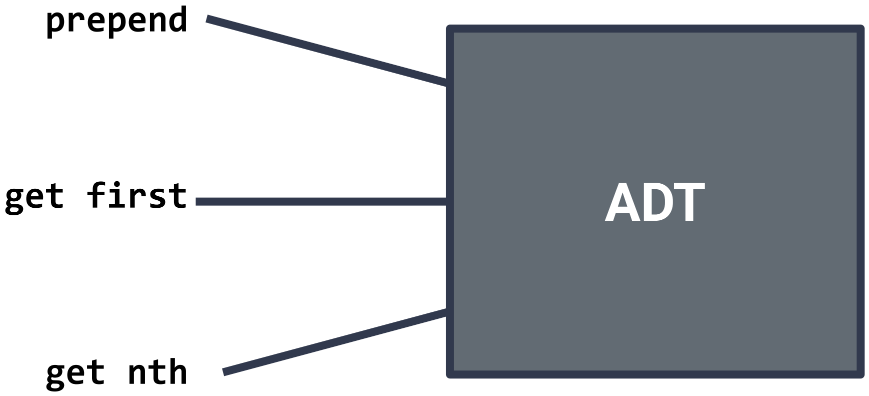
Thought Experiment

An Abstract Data Type is a specification of **what** a data structure can do



Thought Experiment

Often, many data structures can satisfy a given ADT…how do you choose?



Thought Experiment

Data Structure 1
- Very fast `prepend`, `get first`
- Very slow `get nth`

Data Structure 2
- Very fast `get nth`, `get first`
- Very slow `prepend`

Data Structure 3
- Very fast `get nth`, `get first`
- Occasionally slow `prepend`

Which is better?

Thought Experiment

Data Structure 1 (Linked List)

- Very fast `prepend`, `get first`
- Very slow `get nth`

Data Structure 2 (Array)

- Very fast `get nth`, `get first`
- Very slow `prepend`

Data Structure 3 (Array Buffer…in reverse)

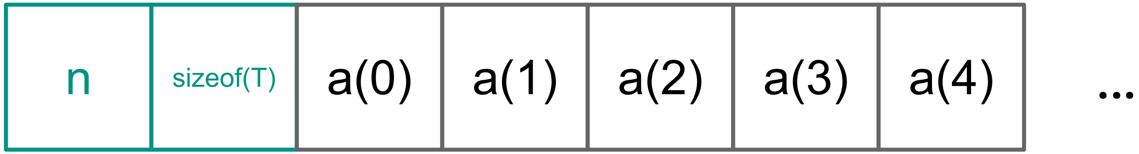
- Very fast `get nth`, `get first`
- Occasionally slow `prepend`

Which is better?

IT DEPENDS!

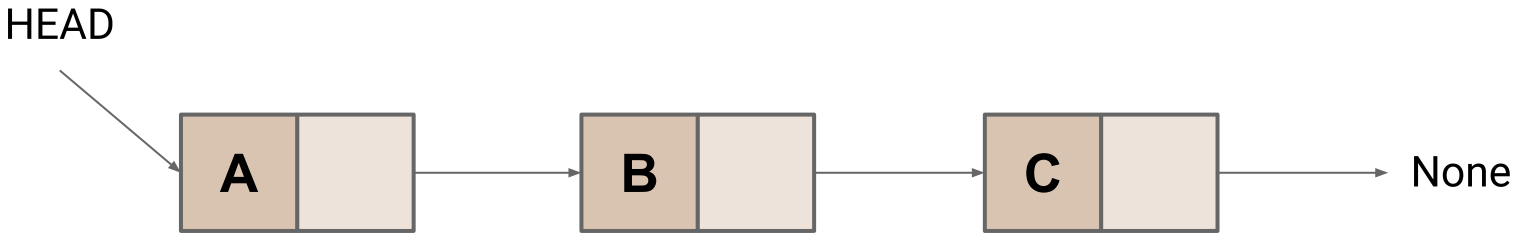
A (very) Brief Refresher: Array

- An array is an ordered container (elements stored one after another)
- Array elements are all stored in a contiguous block of memory

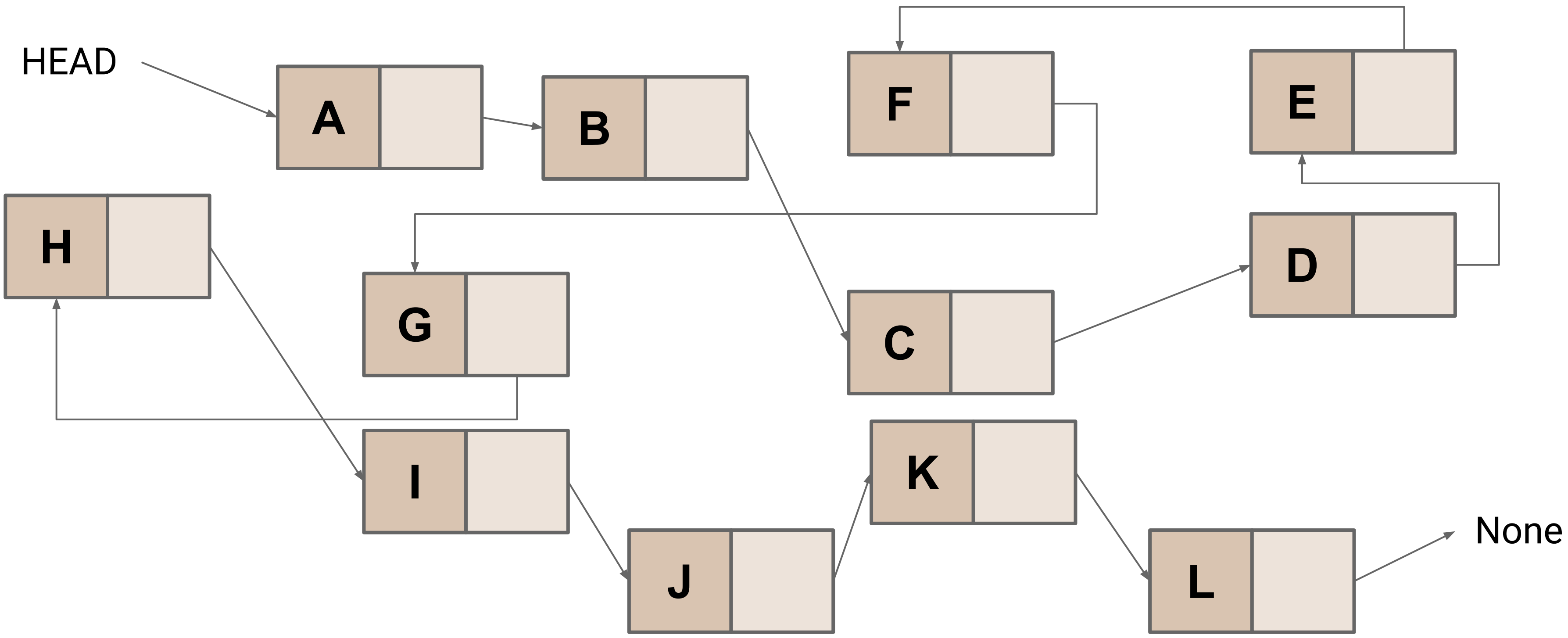


A (very) Brief Refresher: Linked Lists

- Also an ordered container
- Each element stores a pointer to the next element
    - …not necessarily in a contiguous block of memory

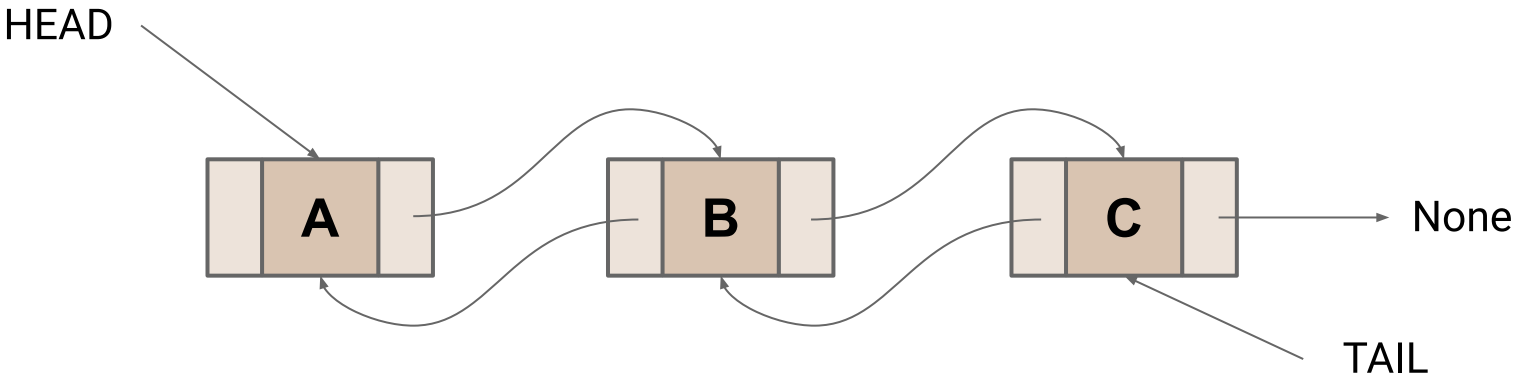


A (very) Brief Refresher: Linked Lists



A (very) Brief Refresher: Linked Lists

- Can also be doubly linked (a next AND a prev pointer per node)
- PA1 will have you implementing a **Sorted Doubly Linked List** with some minor twists



Thought Experiment

Data Structure 1 (Linked List)

- Very fast `prepend`, `get first`
- Very slow `get nth`

Data Structure 2 (Array)

- Very fast `get nth`, `get first`
- Very slow `prepend`

Data Structure 3 (Array Buffer…in reverse)

- Very fast `get nth`, `get first`
- Occasionally slow `prepend`

What is "fast"? "slow"?

Attempt #1: Wall-clock time?

- What is fast?
  - 10s? 100ms? 10ns?
  - …it depends on the task
- Algorithm vs Implementation
  - Compare Grace Hopper’s implementation to yours
- What machine are you running on?
  - Your old laptop? A lab machine? The newest, shiniest processor on the market?
- What bottlenecks exist? CPU vs IO vs Memory vs Network…

Attempt #1: Wall-clock time?

- What is fast?
  - 10s? 100ms? 10ns?
  - …it depends on the task
- Algorithm vs Implementation
  - Compare Grace Hopper's implementation to yours
- What machine are you running on?
  - Your old laptop? A lab machine? The newest, shiniest processor?
- What bottlenecks exist? CPU vs IO vs Memory vs Network…

Wall-clock time is not terribly useful…

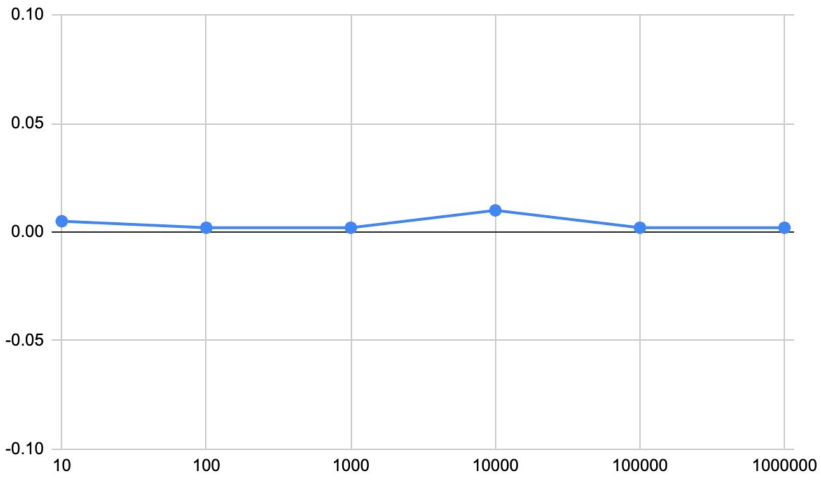
Analysis Checklist

1. Don’t think in terms of wall-time, think in terms of “number of steps”

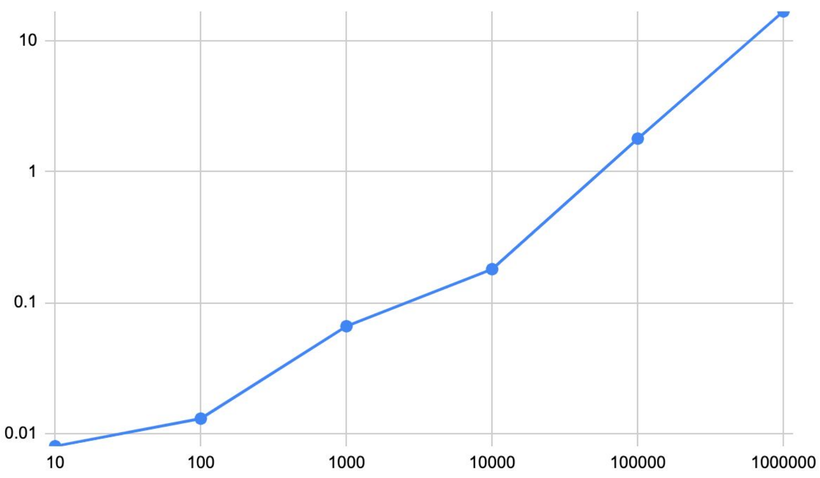
Let’s do a quick demo…

Comparing Random Access for Array vs List

Array

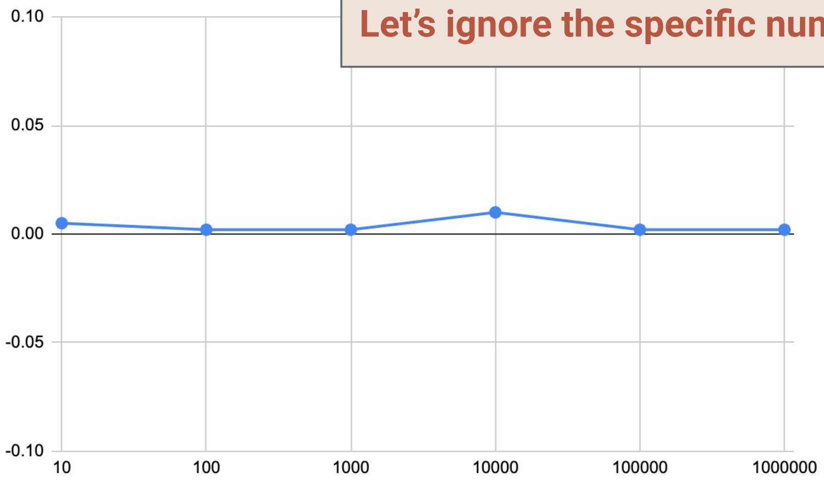


List

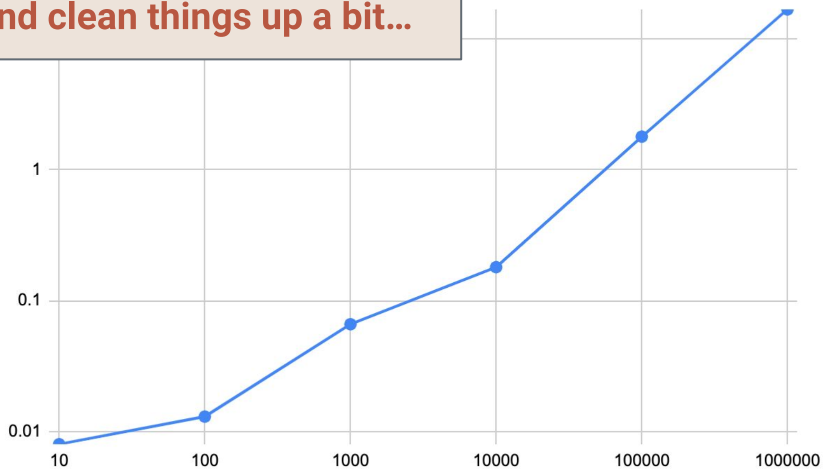


Comparing Random Access for Array vs List

Array



List

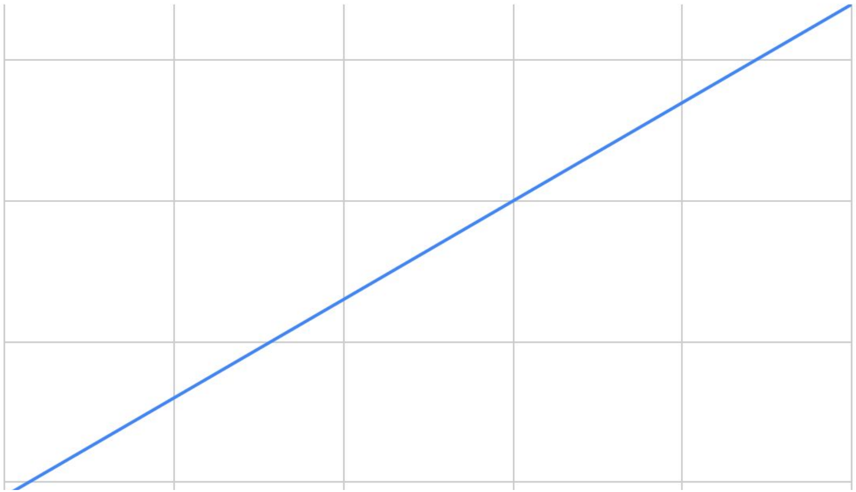


Let’s ignore the specific numbers and clean things up a bit…

Comparing Random Access for Array vs List

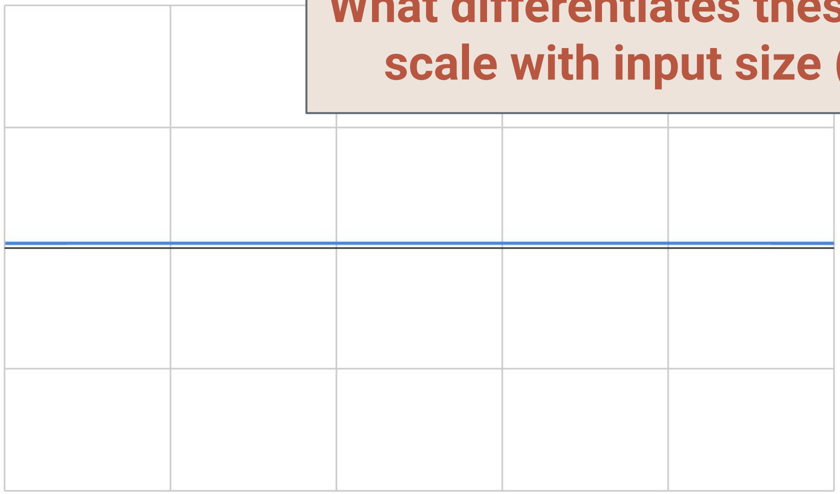
Array

List

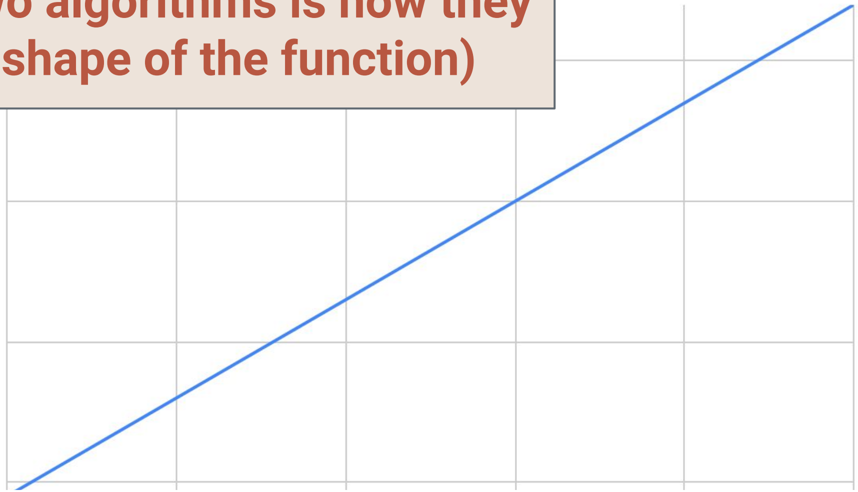


Comparing Random Access for Array vs List

Array



List



What differentiates these two algorithms is how they scale with input size (the shape of the function)

Analysis Checklist

1. Don’t think in terms of wall-time, think in terms of “number of steps”
2. **To give a useful solution, we should take “scale” into account**
    - **How does the runtime change as we change the size of the input?**

Counting Steps

```
public void updateUsers(User[] users) {
  x = 1;
  for(user : users) {
    user.id = x;
    x = x + 1;
  }
}
```

Counting Steps

```
public void updateUsers(User[] users) {
  x = 1;
  for(user : users) {
    user.id = x;
    x = x + 1;
  }
}
```

1

Counting Steps

```java
public void updateUsers(User[] users) {
  x = 1;
  for(user : users) {
    user.id = x;
    x = x + 1;
  }
}
```

$$1 + \sum_{user \in users}$$

Counting Steps

```java
public void updateUsers(User[] users) {
  x = 1;
  for(user : users) {
    user.id = x;
    x = x + 1;
  }
}
```

$$1 + \sum_{user \in users} 4$$

Counting Steps

```java
public void updateUsers(User[] users) {
  x = 1;
  for(user : users) {
    user.id = x;
    x = x + 1;
  }
}
```

$$1 + \sum_{user \in users} 4 = 1 + 4 \cdot |\mathtt{users}|$$

Counting Steps

```java
public void userFullName(User[] users, int id) {
  User user = users[id];
  String fullName = user.firstName + user.lastName;
  return fullName;
}
```

Counting Steps

```java
public void userFullName(User[] users, int id) {
  User user = users[id];
  String fullName = user.firstName + user.lastName;
  return fullName;
}
```

3 steps…(sort of, more details later)

Counting Steps

```java
public void totalReads(User[] users, Post[] posts) {
  int totalReads = 0;
  for(post : posts) {
    int userReads = 0;
    for(user : users) {
      if(user.readPost(post)){ userReads += 1; }
    }
    totalReads += userReads;
  }
}
```

Counting Steps

```java
public void totalReads(User[] users, Post[] posts) {
  int totalReads = 0;
  for(post : posts) {
    int userReads = 0;
    for(user : users) {
      if(user.readPost(post)){ userReads += 1; }
    }
    totalReads += userReads;
  }
}
```

1 .

Counting Steps

```
public void totalReads(User[] users, Post[] posts) {
  int totalReads = 0;
  for(post : posts) {
    int userReads = 0;
    for(user : users) {
      if(user.readPost(post)){ userReads += 1; }
    }
    totalReads += userReads;
  }
}
```

$$1 + \sum_{post \in posts}$$

Counting Steps

```java
public void totalReads(User[] users, Post[] posts) {
  int totalReads = 0;
  for(post : posts) {
    int userReads = 0;
    for(user : users) {
      if(user.readPost(post)){ userReads += 1; }
    }
    totalReads += userReads;
  }
}
```

$$1 + \sum_{post \in posts} \Bigg(3$$

Counting Steps

```java
public void totalReads(User[] users, Post[] posts) {
  int totalReads = 0;
  for(post : posts) {
    int userReads = 0;
    for(user : users) {
      if(user.readPost(post)){ userReads += 1; }
    }
    totalReads += userReads;
  }
}
```

$$1 + \sum_{post \in posts} \left(3 + \sum_{user \in users} \right.$$

Counting Steps

```java
public void totalReads(User[] users, Post[] posts) {
  int totalReads = 0;
  for(post : posts) {
    int userReads = 0;
    for(user : users) {
      if(user.readPost(post)){ userReads += 1; }
    }
    totalReads += userReads;
  }
}
```

$$1 + \sum_{post \in posts} \left(3 + \sum_{user \in users} 2 \right)$$

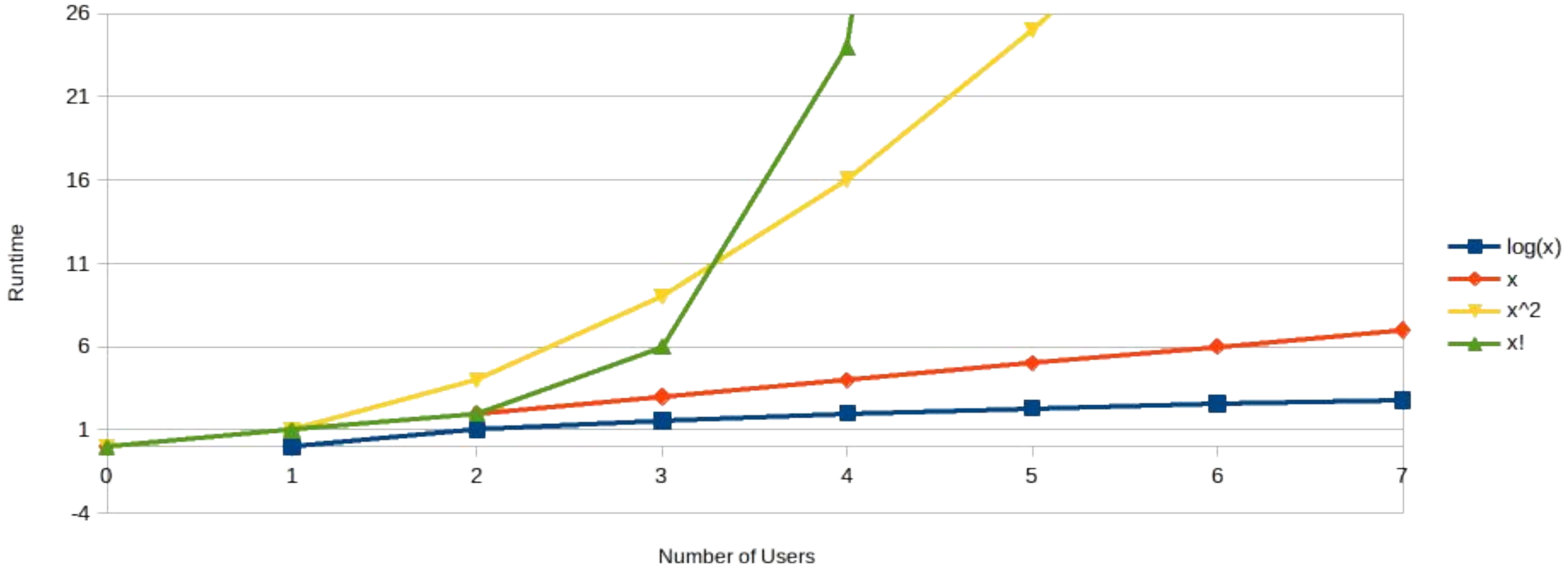
Steps to "Functions"

Now that we have number of steps in terms of summations…

…which we can simplify (like in WA1) into mathematical functions…

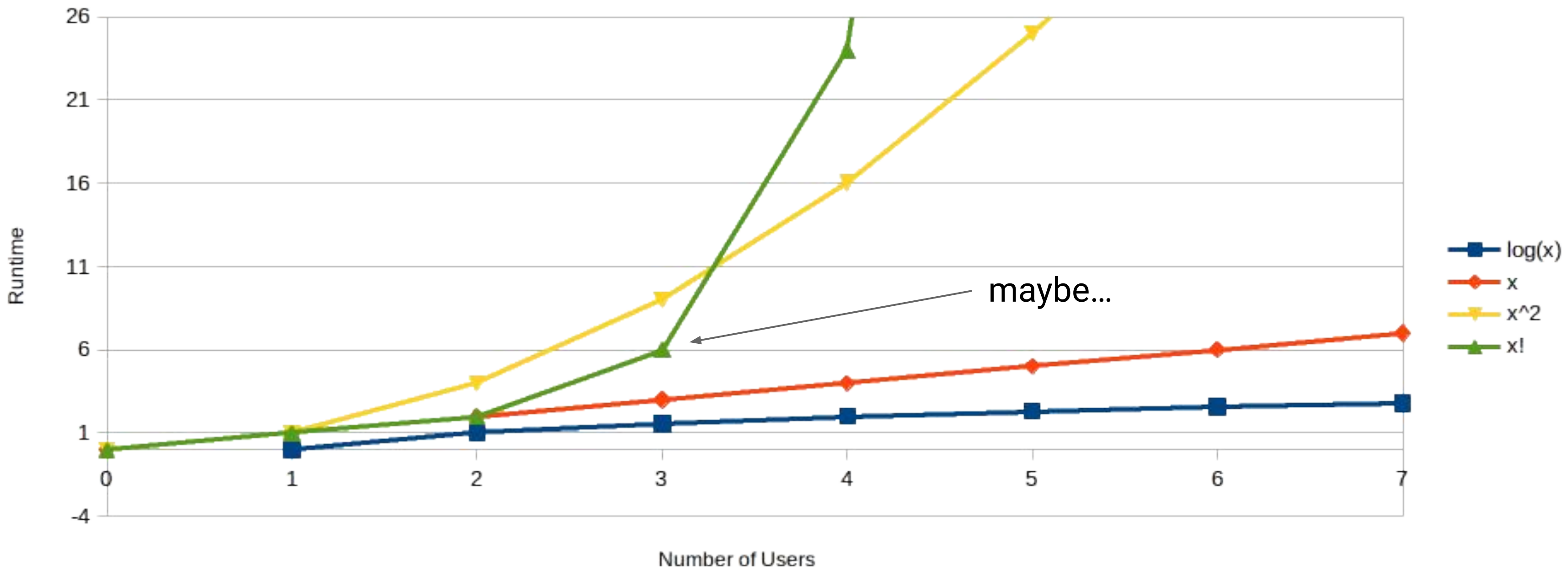
We can start analyzing runtime as a function

Runtime as a Function



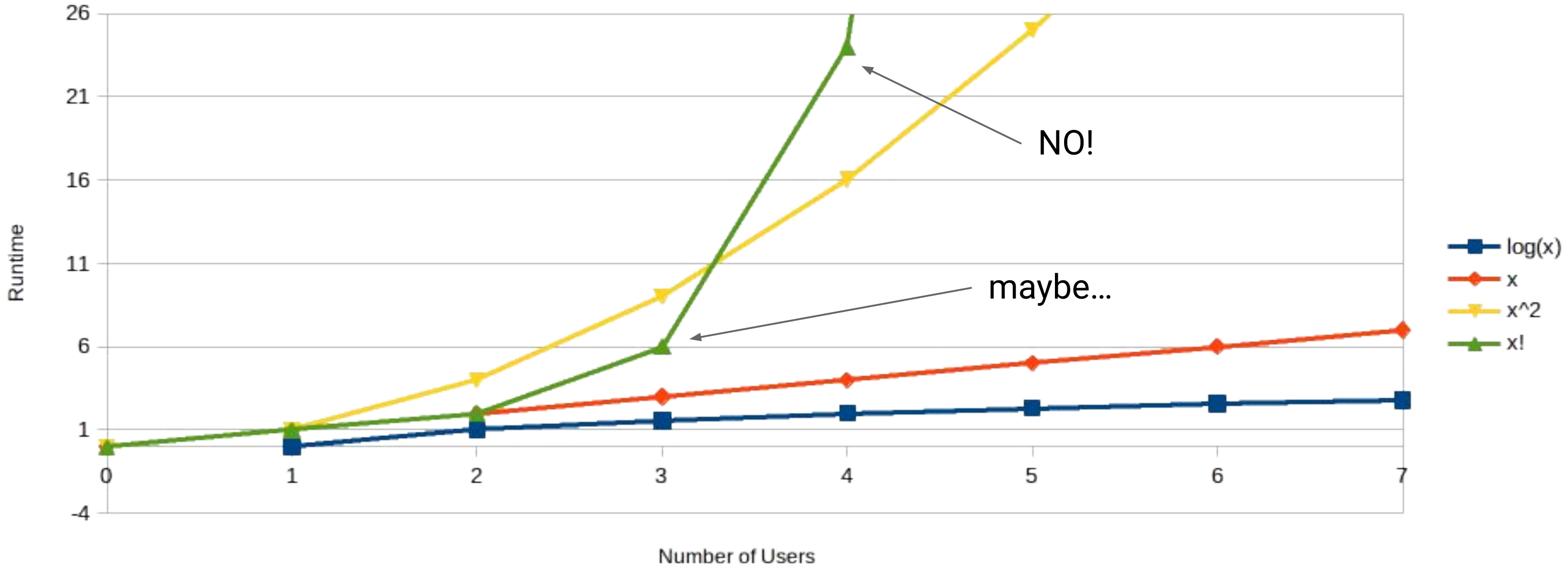
Would you consider an algorithm that takes |Users|! number of steps?

Runtime as a Function



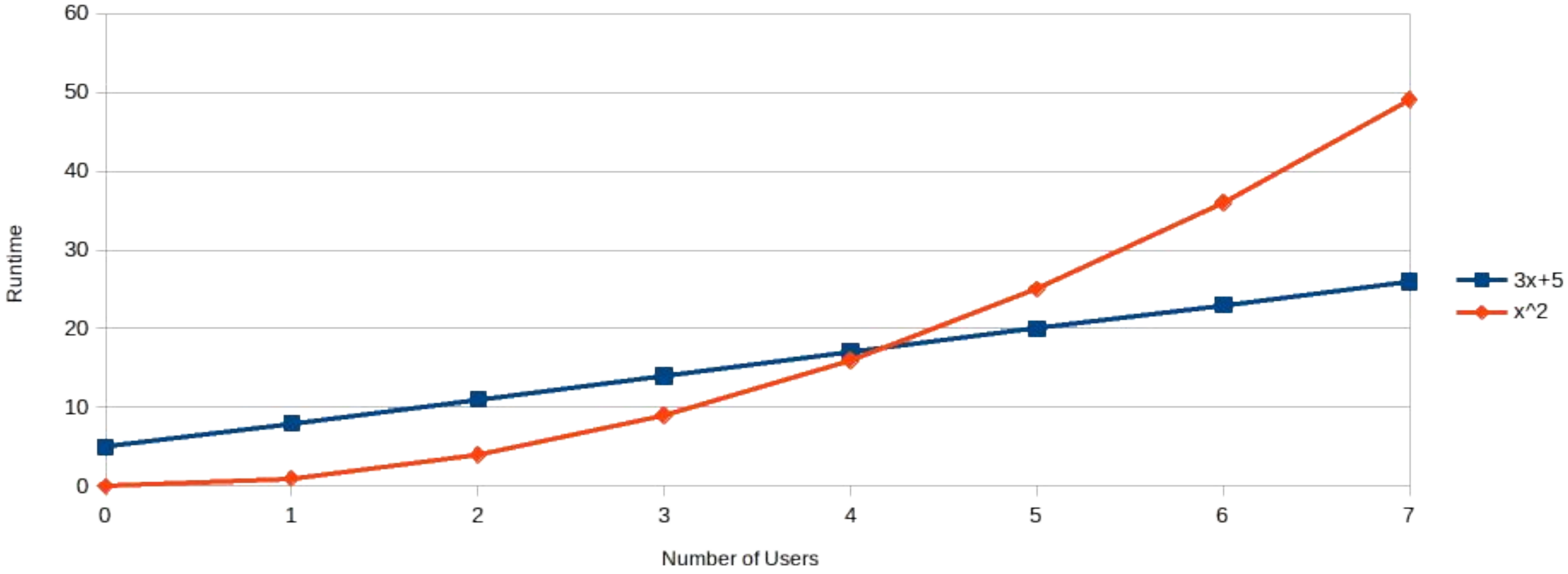
Would you consider an algorithm that takes |Users|! number of steps?

Runtime as a Function



Would you consider an algorithm that takes |Users|! number of steps?

Runtime as a Function



Which is better? 3x|Users|+5 or $|Users|^2$

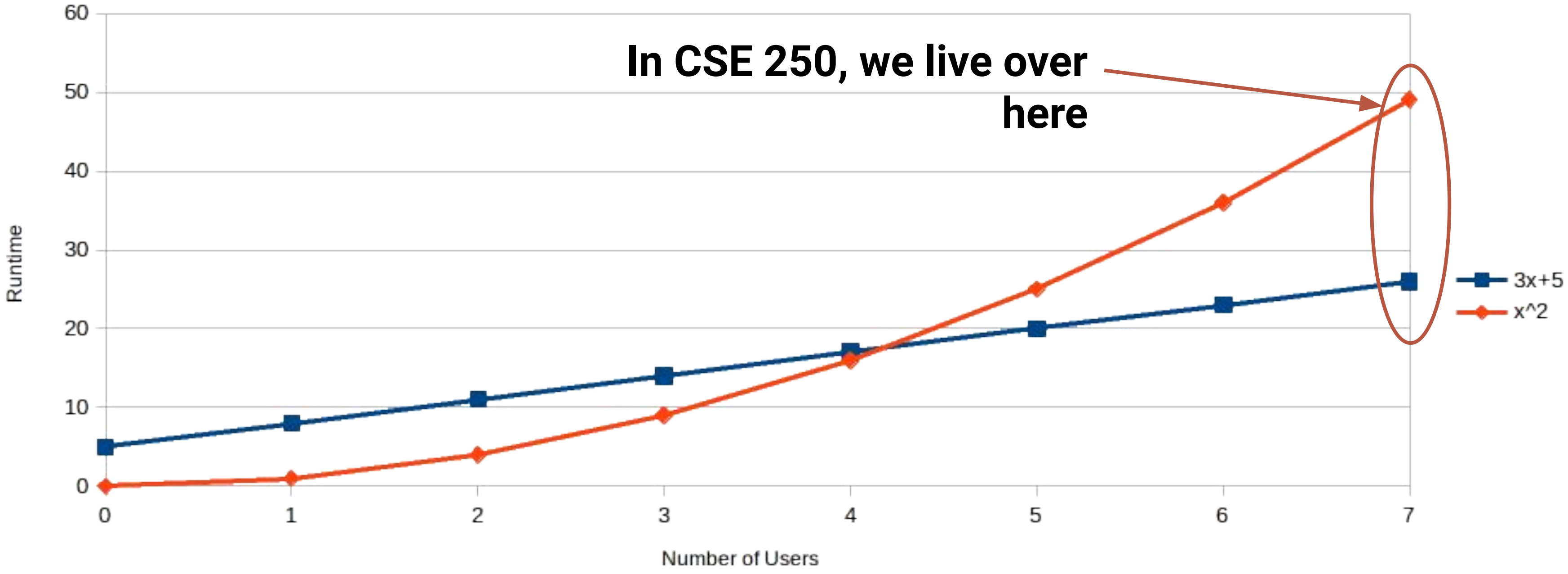
Analysis Checklist

1. Don't think in terms of wall-time, think in terms of "number of steps"
2. To give a useful solution, we should take "scale" into account
    - How does the runtime change as we change the size of the input?

Analysis Checklist

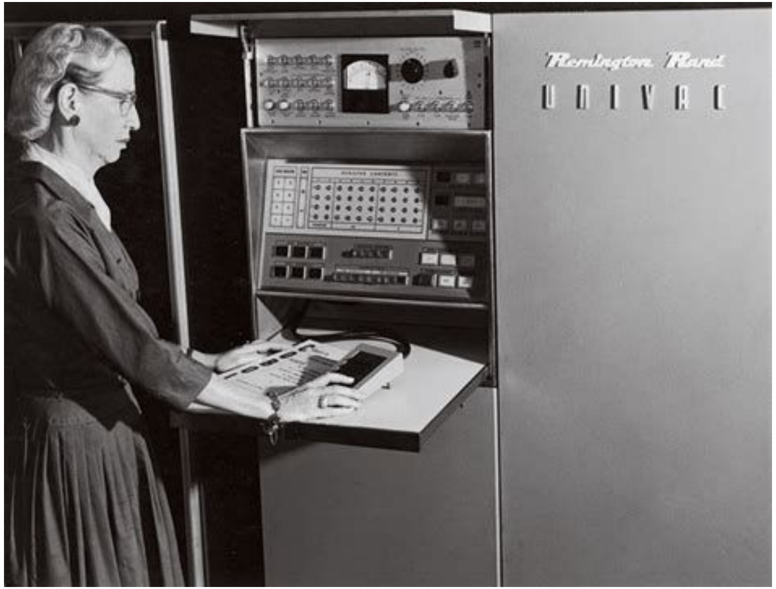
1. Don’t think in terms of wall-time, think in terms of “number of steps”
2. To give a useful solution, we should take “scale” into account
    - How does the runtime change as we change the size of the input?
3. **Focus on “large” inputs**
    - **Rank functions based on how they behave at large scales**

Runtime as a Function

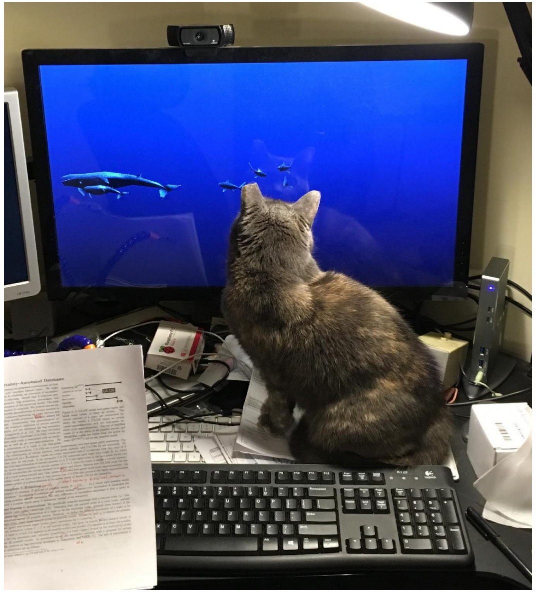


Which is better? 3x|Users|+5 or $|Users|^2$

Goal: Ignore implementation details



vs

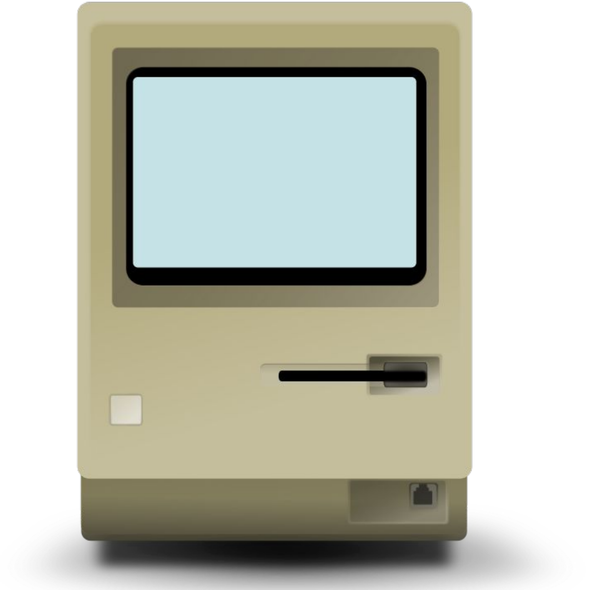


Seasoned Pro Implementation

Error 23: Cat on Keyboard

Goal: Ignore execution environment

vs



Intel i9

Motorola 68000

Images from openclipart.org, used with permission

Goal: Judge the Algorithm Itself

- How fast is a step? Don’t care
  - Only count number of steps
- Can this be done in two steps instead of one?
  - “3 steps per user” vs “some number of steps per user”
  - Sometimes we don’t care…sometimes we do

Analysis Checklist

1. Don't think in terms of wall-time, think in terms of "number of steps"
2. To give a useful solution, we should take "scale" into account
   - How does the runtime change as we change the size of the input?
3. Focus on "large" inputs
   - Rank functions based on how they behave at large scales

Analysis Checklist

1. Don't think in terms of wall-time, think in terms of "number of steps"
2. To give a useful solution, we should take "scale" into account
    - How does the runtime change as we change the size of the input?
3. Focus on "large" inputs
    - Rank functions based on how they behave at large scales
4. **Decouple algorithm from infrastructure/implementation**
    - **Asymptotic notation…?**

Attempt #2: Growth Functions

Not a function in code…but a mathematical function:

$$T(n)$$

n: The "size" of the input

ie: number of users,rows, pixels, etc

***T*(n): The number of "steps" taken for input of size n**

ie: 20 steps per user, where n = |Users|, is 20 x n

Some Basic Assumptions:

Problem sizes are non-negative integers

$$n \in \{0, 1, 2, 3, \ldots\} = \{0\} \cup \mathbb{Z}^+$$

We can’t reverse time…(obviously)

$$T(n) > 0$$

Smaller problems aren’t harder than bigger problems

$$n_1 < n_2 \Rightarrow T(n_1) \leq T(n_2)$$

Some Basic Assumptions:

Problem sizes are non-negative integers

$$n \in \{0, 1, 2, 3, \ldots\} = \{0\} \cup \mathbb{Z}^+$$

We can't reverse time…(obviously)

$$T(n) > 0$$

Smaller problems aren't harder than bigger problems

$$n_1 < n_2 \Rightarrow T(n_1) \leq T(n_2)$$

$T: \{0\} \cup \mathbb{Z}^+ \to \mathbb{R}^+$

T is non-decreasing

First Problem...

We are still implementation dependent...

$$T_1(n) = 19n$$

$$T_2(n) = 20n$$

First Problem...

We are still implementation dependent...

$$T_1(n) = 19n$$

$$T_2(n) = 20n$$

Does 1 extra step per element really matter...?

Is this just an implementation detail?

First Problem...

We are still implementation dependent...

$$T_1(n) = 19n$$

$$T_2(n) = 20n$$

$$T_3(n) = 2n^2$$

T_1 and T_2 are much more "similar" to each other than they are to T_3

First Problem...

We are still implementation dependent...

$$T_1(n) = 19n$$

$$T_2(n) = 20n$$

$$T_3(n) = 2n^2$$

T_1 and T_2 are much more "similar" to each other than they are to T_3

How do we capture this idea formally?

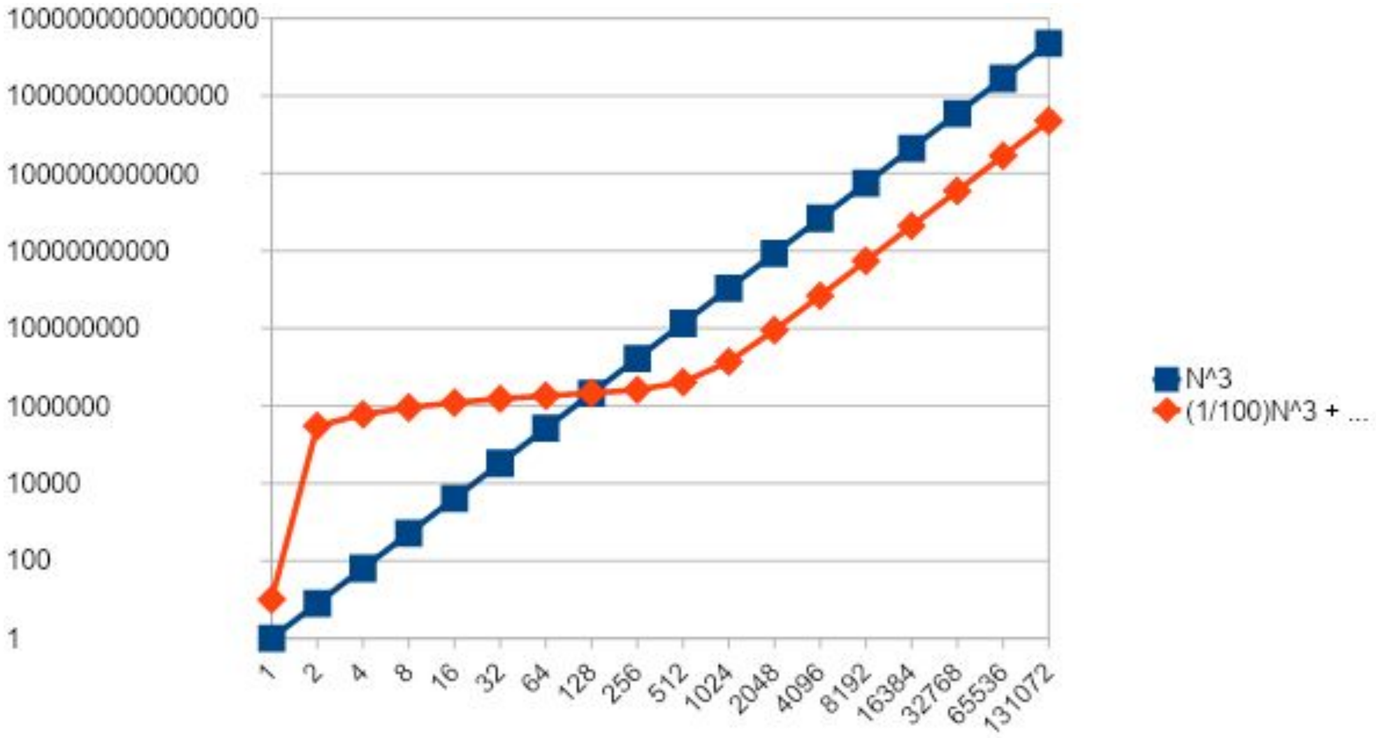
How Do We Capture Behavior at Scale?

Consider the following two functions:

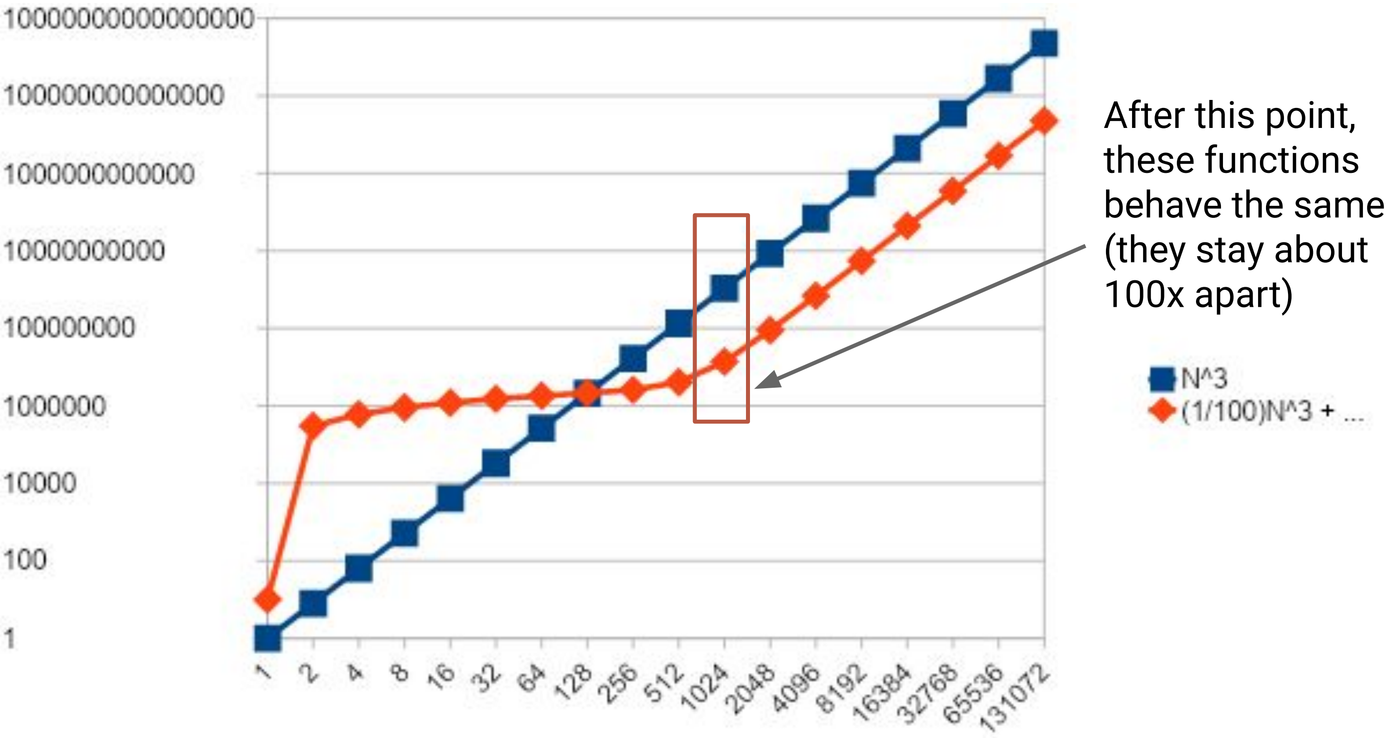
$$\frac{1}{100}n^3 + 10n + 1000000\log(n)$$

$$n^3$$

How Do We Capture Behavior at Scale?



How Do We Capture Behavior at Scale?



How Do We Capture Behavior at Scale?

$$\lim_{n \to \infty} \frac{\frac{1}{100} n^3 + 10n + 1000000 \log(n)}{n^3}$$

How Do We Capture Behavior at Scale?

$$\lim_{n\to\infty} \frac{\frac{1}{100}n^3 + 10n + 1000000\log(n)}{n^3}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{100}n^3}{n^3} + \frac{10n}{n^3} + \frac{1000000\log(n)}{n^3}$$

How Do We Capture Behavior at Scale?

$$\lim_{n\to\infty} \frac{\frac{1}{100}n^3 + 10n + 1000000\log(n)}{n^3}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{100}n^3}{n^3} + \frac{10n}{n^3} + \frac{1000000\log(n)}{n^3}$$

$$= \lim_{n\to\infty} \frac{1}{100} + \boxed{\lim_{n\to\infty} \frac{10}{n^2}} + \boxed{\lim_{n\to\infty} \frac{1000000\log(n)}{n^3}}$$

These terms go to 0

How Do We Capture Behavior at Scale?

$$\lim_{n\to\infty} \frac{\frac{1}{100}n^3 + 10n + 1000000\log(n)}{n^3}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{100}n^3}{n^3} + \frac{10n}{n^3} + \frac{1000000\log(n)}{n^3}$$

$$= \lim_{n\to\infty} \frac{1}{100} + \lim_{n\to\infty} \frac{10}{n^2} + \lim_{n\to\infty} \frac{1000000\log(n)}{n^3}$$

$$= \frac{1}{100}$$

Attempt #3: Asymptotic Analysis

Consider two functions, f(n) and g(n)

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

In this particular case, f grows w.r.t. n faster than g

So…if f(n) and g(n) are the number of steps two different algorithms take on a problem of size n, which is better?

Attempt #3: Asymptotic Analysis

Case 1: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$ *(f grows faster; g is better)*

Case 2: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$ *(g grows faster; f is better)*

Case 3: $\lim_{n \to \infty} \frac{f(n)}{g(n)} = some\ \ constant$ *(f and g "behave" the same)*

Goal of “Asymptotic Analysis”

We want to organize runtimes (growth functions) into different ***Complexity Classes***

Within the same complexity class, runtimes “behave the same”

Goal of “Asymptotic Analysis”

“Strategic Optimization” focuses on improving the complexity class of your code!

Back to Our Previous Example...

$$\frac{1}{100}n^3 + 10n + 1000000\log(n)$$

The 10n and 1000000 log(n) “don’t matter”

The 1/100 “does not matter”

Back to Our Previous Example...

$$\frac{1}{100}n^3 + 10n + 1000000\log(n)$$

The 10n and 1000000 log(n) “don’t matter”

The 1/100 “does not matter”

n^3 is the dominant term, and that determines the “behavior”

Why Focus on Dominating Terms?

$f(n)$	**10**	**20**	**50**	**100**	**1000**
$log(log(n))$	0.43 ns	0.52 ns	0.62 ns	0.68 ns	0.82 ns
$log(n)$	0.83 ns	1.01 ns	1.41 ns	1.66 ns	2.49 ns
n	2.5 ns	5 ns	12.5 ns	25 ns	0.25 μs
$nlog(n)$	8.3 ns	22 ns	71 ns	0.17 μs	2.49 μs
n^2	25 ns	0.1 μs	0.63 μs	2.5 μs	0.25 ms
n^5	25 μs	0.8 ms	78 ms	2.5 s	**2.9 days**
2^n	0.25 μs	0.26 ms	**3.26 days**	**10^{13} years**	**10^{284} years**
$n!$	0.91 ms	**19 years**	**10^{47} years**	**10^{141} years**	🤯

Why Focus on Dominating Terms?

$$2^n \gg n^c \gg n \gg log(n) \gg c$$