COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 2: Measuring Performance I

Chris Lucas (cflucas@ncat.edu)

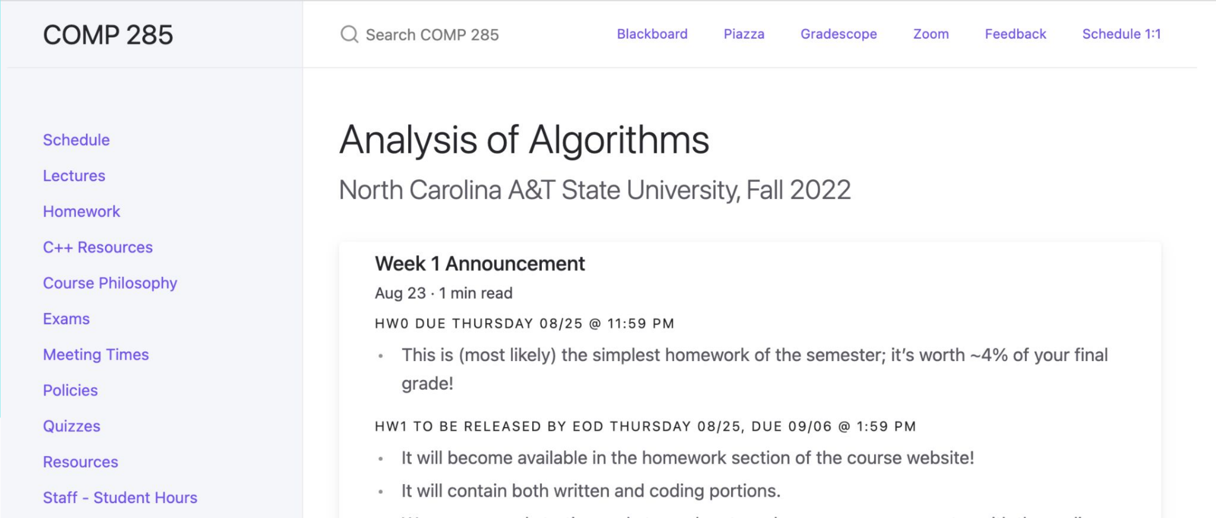
HW0 is Due!
@11:59PM

The easiest homework in this class, so hopefully everyone completes it.

Worth ~4% of your final grade.

HW1 Released by EOD!

comp285-fall22.ml



COMP 285

Search COMP 285

Blackboard Piazza Gradescope Zoom Feedback Schedule 1:1

- Schedule
- Lectures
- Homework
- C++ Resources
- Course Philosophy
- Exams
- Meeting Times
- Policies
- Quizzes
- Resources
- Staff - Student Hours

Analysis of Algorithms

North Carolina A&T State University, Fall 2022

Week 1 Announcement

Aug 23 · 1 min read

HW0 DUE THURSDAY 08/25 @ 11:59 PM

- This is (most likely) the simplest homework of the semester; it's worth ~4% of your final grade!

HW1 TO BE RELEASED BY EOD THURSDAY 08/25, DUE 09/06 @ 1:59 PM

- It will become available in the homework section of the course website!
- It will contain both written and coding portions.

Piazza!

(search then post)

link

Thank you Tolani!

Before that!

Teaching Assistants

Priya Rachakonda

lrachakonda@aggies.ncat.edu

Recall What We Accomplished!

- Course Philosophy
  - Algorithms are fun!
    - Does it work?
    - How fast?
    - Can I do better?
- Technical Stuff
  - Karatsuba for integer multiplication
  - Divide & Conquer
  - Hand-wavy “runs in big-Oh” sort of stuff

.

Today

- We’re getting more formal…
  - Does it work?
  - How fast?
- How?
  - Introducing examples
  - Building upon to get to full-fledged algorithm(s)

Big Questions!

- How to multiply faster?
- What is Big-O?
- Why do we Big-O?
- How do we Big-O?

Recall where we ended last lecture...

Can we do this for multiplication of integers?

$$1234 * 5678$$

How fast is this anyway?

About n^2 one-digit operations

(How many one-digit operations?)

How fast is this anyway?

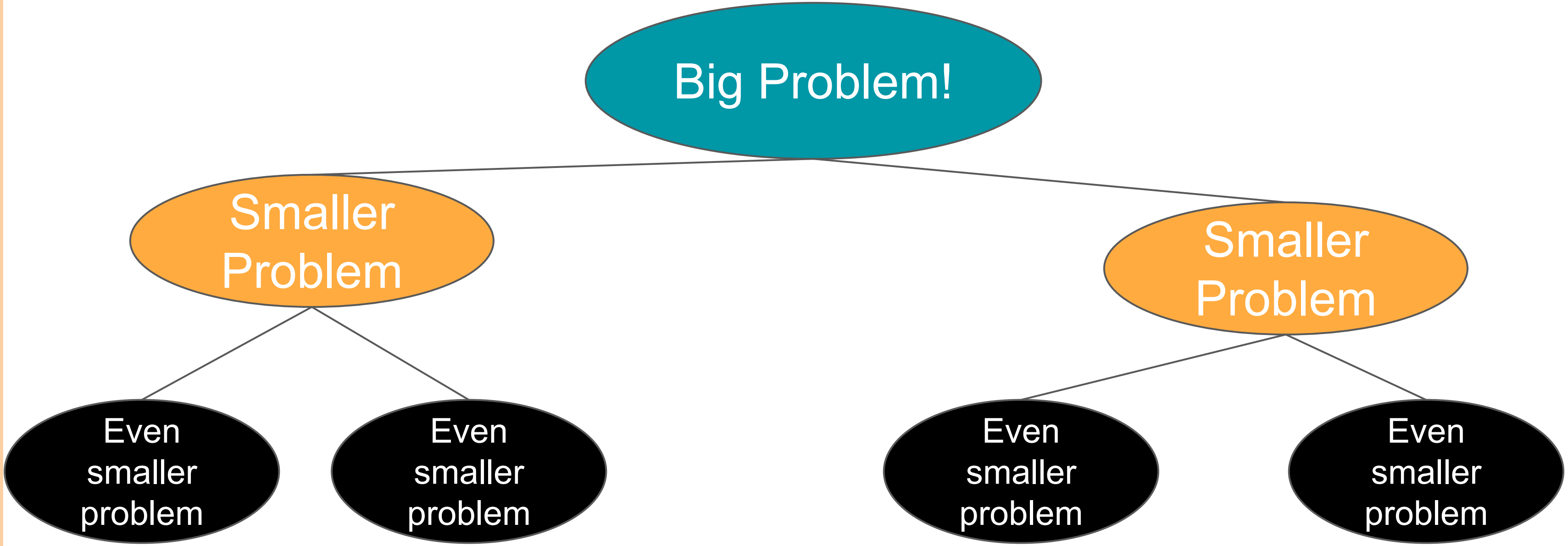
About n^2 one-digit operations

*Multiply each one of the n-digits in the first number with each one of the n-digits in the second number (n * n)*

(How many one-digit operations?)

A technique to know! - Divide & Conquer

Break problem up into smaller (easier) sub-problems



And in general!

Break up an n-digit integer x:

$[x_1, x_2, \ldots, x_n] = [x_1, x_2, \ldots x_{n/2}] * 10^{n/2} + [x_{n/2+1}, x_{n/2+2}, \ldots, x_n]$

And in general!

Break up an n-digit integer x:

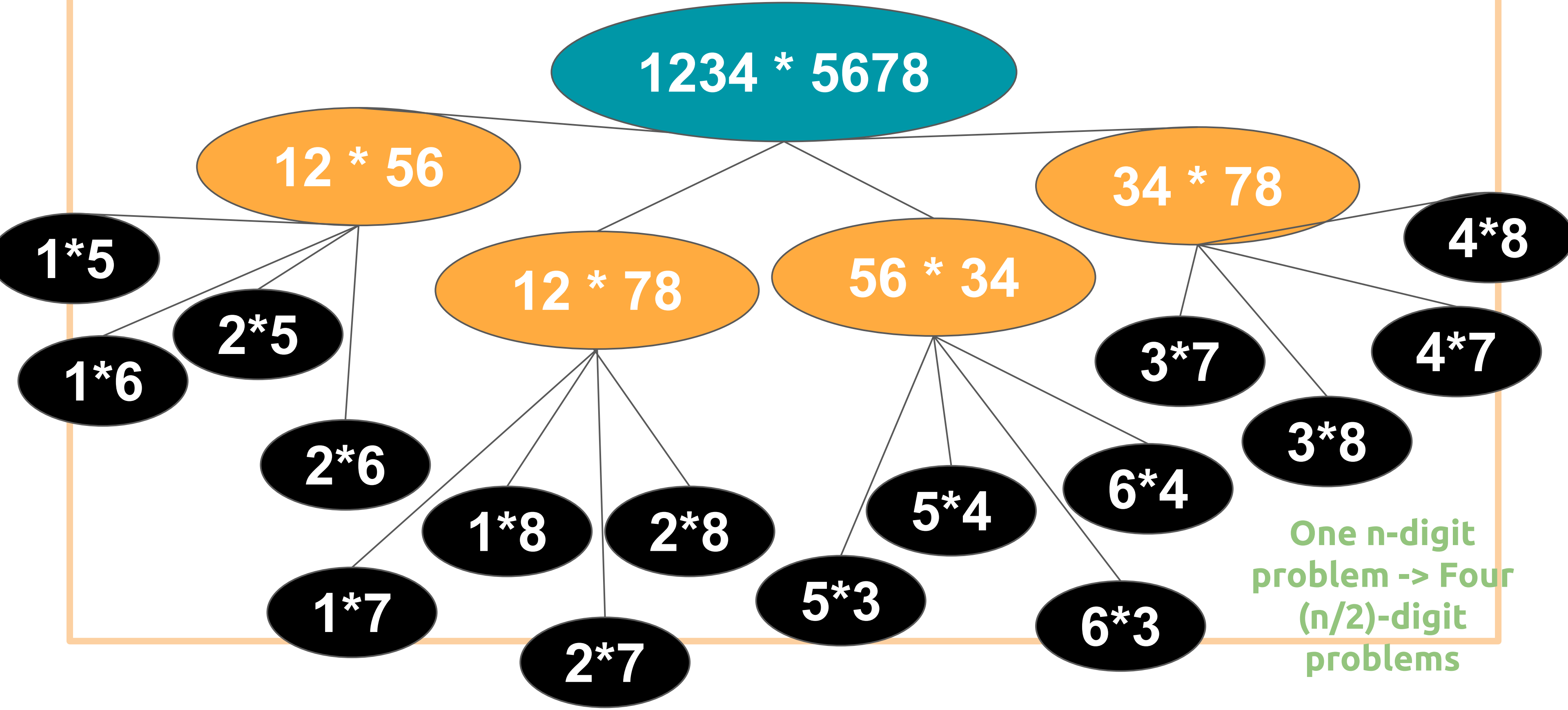
$[x_1, x_2, \ldots, x_n] = [x_1, x_2, \ldots x_{n/2}]*10^{n/2} + [x_{n/2+1}, x_{n/2+2}, \ldots, x_n]$

$$x \times y = (a \times 10^{n/2} + b)(c \times 10^{n/2} + d)$$

$$= \underbrace{(a \times c)}_{1} 10^n + (\underbrace{a \times d}_{2} + \underbrace{c \times b}_{3}) 10^{n/2} + \underbrace{(b \times d)}_{4}$$

One n-digit problem -> Four (n/2)-digit problems

Can we do this for multiplication of integers? (pt. 2)



One n-digit problem -> Four (n/2)-digit problems

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x*, *y*):

If n =1:

return x*y

Base case: we have 1-digit multiplication, cannot break into subproblems

return answer

So what is our algorithm in pseudocode?

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x*, *y*):

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x*, *y*):

return answer

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x*, *y*):

If n =1:

return x*y

Compute a, b, c, d from x, y

Base case: we have 1-digit multiplication, cannot break into subproblems

a, b, c, d are n/2-digit numbers

return answer

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x, y*):

If n =1:

return x*y

Compute a, b, c, d from x, y

Compute *ac, ad, bc, bd* from ???

return answer

Base case: we have 1-digit multiplication, cannot break into subproblems

a, b, c, d are n/2-digit numbers

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x*, *y*):

If n =1:

return x*y

Base case: we have 1-digit multiplication, cannot break into subproblems

Compute a, b, c, d from x, y

a, b, c, d are n/2-digit numbers

Compute *ac*, *ad*, *bc*, *bd* from recursion

ac =

ad =

bc=

bd =

Recursive cases

return answer

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x*, *y*):

If n =1:

return x*y

Compute a, b, c, d from x, y

Compute *ac*, *ad*, *bc*, *bd* from recursion

ac = **Multiply(**a,c**)**

ad = **Multiply(**a,d**)**

bc= **Multiply(**b,c**)**

bd = **Multiply(**b,d**)**

return answer

Base case: we have 1-digit multiplication, cannot break into subproblems

a, b, c, d are n/2-digit numbers

Recursive cases

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(x, y):

Base case: we have 1-digit multiplication, cannot break into subproblems

If n =1:

    return x*y

Compute a, b, c, d from x, y

a, b, c, d are n/2-digit numbers

Compute *ac, ad, bc, bd* from recursion

    ac = **Multiply(**a,c**)**

    ad = **Multiply(**a,d**)**

    bc= **Multiply(**b,c**)**

    bd = **Multiply(**b,d**)**

Recursive cases

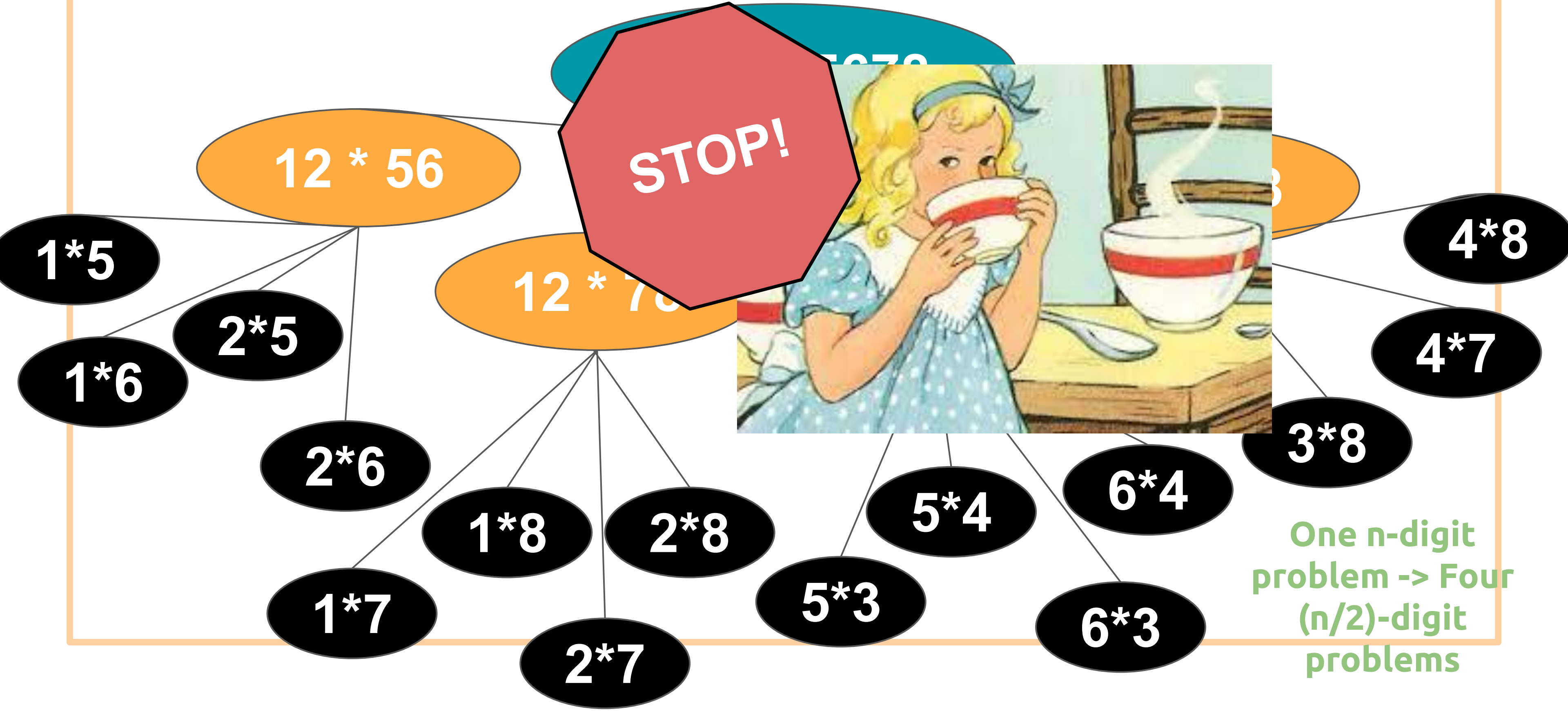
Calculate *xy* using results

    answer = $ac\,10^n + (ad + bc)\,10^{n/2} + bd$

return answer

Let’s code it!!!

Can we do this for multiplication of integers? (pt. 2)



One n-digit problem -> Four (n/2)-digit problems

How does this work?

$$x \times y = (a \times 10^{n/2} + b)(c \times 10^{n/2} + d)$$

$$= (a \times c)10^{n} + (a \times d + c \times b)10^{n/2} + (b \times d)$$

How does this work?

$$x \times y = (a \times 10^{n/2} + b)(c \times 10^{n/2} + d)$$
$$= (a \times c)10^n + (a \times d + c \times b)10^{n/2} + (b \times d)$$

(a+b)(c+d) = ac + bd + bc + ad

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Multiply(*x, y***):**

If n =1:

    return x*y

Base case: we have 1-digit multiplication, cannot break into subproblems

Compute a, b, c, d from x, y

a, b, c, d are n/2-digit numbers

Compute *ac, ad, bc, bd* from recursion

    ac = **Multiply(**a,c**)**

    ad = **Multiply(**a,d**)**

    bc= **Multiply(**b,c**)**

    bd = **Multiply(**b,d**)**

Recursive cases

Calculate *xy* using results

    answer = $ac\,10^{n} + (ad + bc)\,10^{n/2} + bd$

return answer

So what is our algorithm in pseudocode?

x,y are n-digit numbers

Karatsuba(*x, y***):**

If n =1:

return x*y

Compute a, b, c, d from x, y

Compute *ac, ad, bc, bd* from recursion

ac = **Karatsuba(**a,c**)**

bd = **Karatsuba(**b,d**)**

z = **Karatsuba(**a+b,c+d**)**

Calculate *xy* using results

answer = $ac\,10^n + (z - ac - bd)10^{n/2} + bd$

return answer

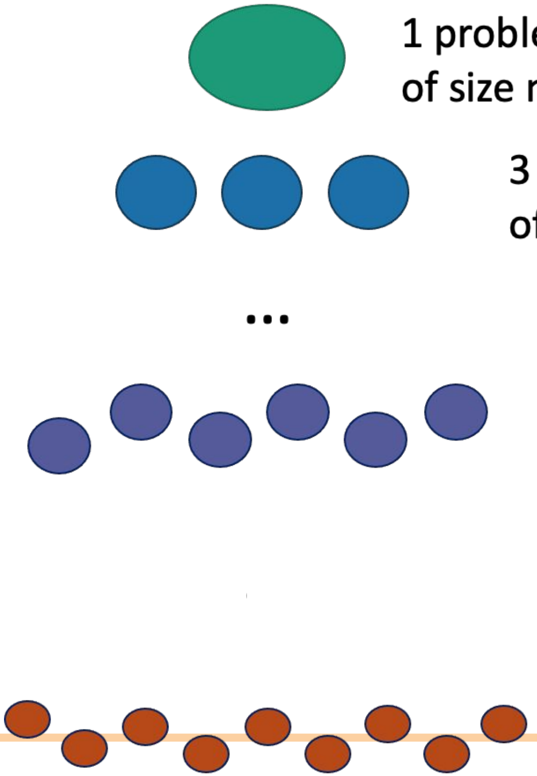
Base case: we have 1-digit multiplication, cannot break into subproblems

a, b, c, d are n/2-digit numbers

Recursive cases

Let’s code it!!!

What's the running time?



1 problem
of size n

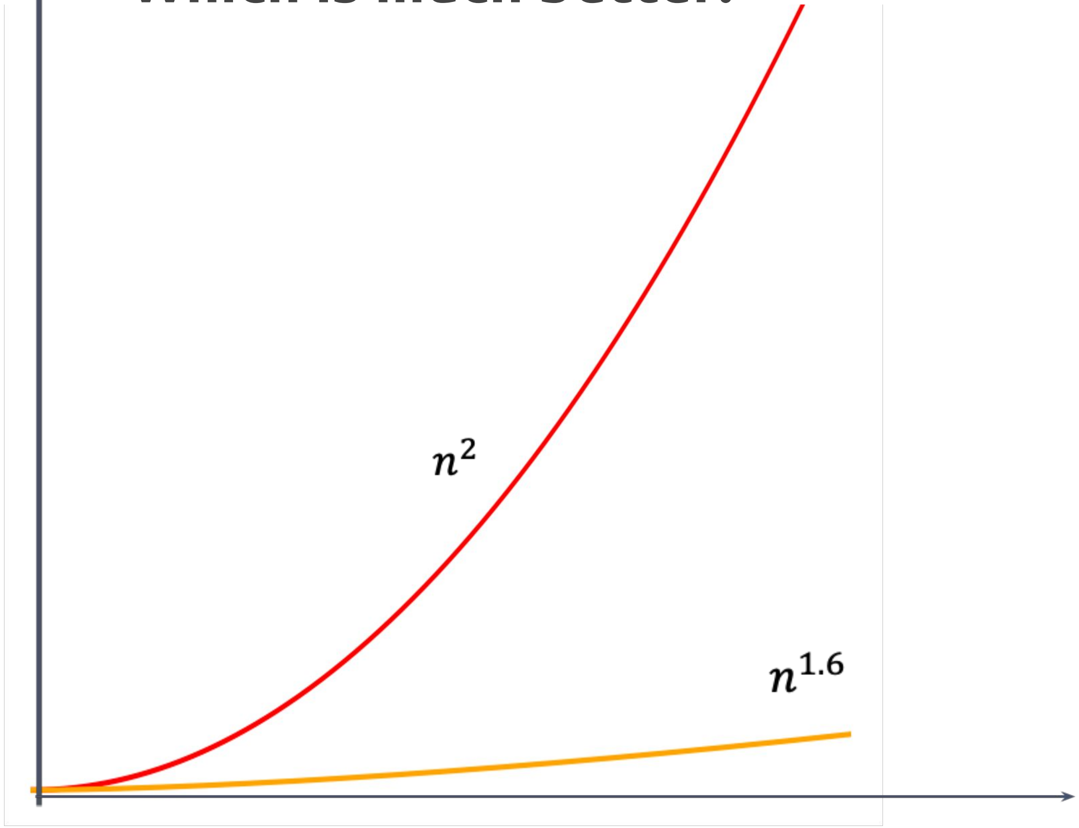
3 problems
of size n/2

...

3^t problems
of size $n/2^t$

$\underline{n^{1.6}}$ problems
of size 1

Which is much better!



Big Questions!

- How to multiply faster?
- What is Big-O? (informal)
- Why do we Big-O?
- How do we Big-O?

Putting it together!

Putting it together!

+

KaratsubaMultiply(*x*, *y*):

Putting it together!

KaratsubaMultiply(x, y):

Big-O!

Putting it together!

+ **KaratsubaMultiply(*x*, *y*):**

= **Big-O!**

Increase input size -> increase # operations ?

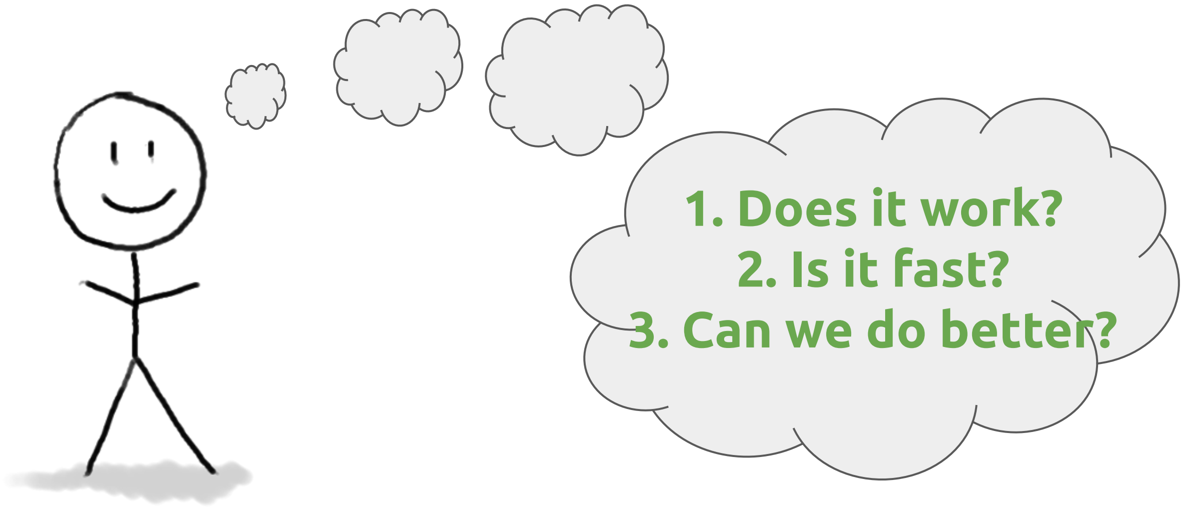
What is Big-O?

What is Big-O?

- Algorithms are judged by their **correctness** and **efficiency** (time efficiency and space efficiency).

What is Big-O?

- Algorithms are judged by their **correctness** and **efficiency** (time efficiency and space efficiency).



What is Big-O?

- Algorithms are judged by their **correctness** and **efficiency** (time efficiency and space efficiency).

- Big-O is **how we quantify efficiency**; it gives us a way to compare different algorithms to say which are better than others.

What is Big-O?

- Algorithms are judged by their **correctness** and **efficiency** (time efficiency and space efficiency).

- Big-O is **how we quantify efficiency**; it gives us a way to compare different algorithms to say which are better than others.

(This is also something asked in every whiteboard coding interview!)

What is Big-O? (pt. 2)

- Big-O is a way to express the algorithm’s efficiency in terms of the size of its input (which we often call “N”).

What is Big-O? (pt. 2)

- Big-O is a way to express the algorithm’s efficiency in terms of the size of its input (which we often call “N”).

- Big-O communicates an upper-bound on how many “operations” an algorithm will take.

KaratsubaMultiply

What is an “operation”?

What is an “operation”?

- Examples of a single operation (or “step”) of a program:
  - x + y
  - a == b
  - int x = 4
  - std::cout << 4
  - vec.size()

Big Questions!

- How to multiply faster?
- What is Big-O? (informal)
- Why do we Big-O?
- How do we Big-O?

Why does efficiency matter?

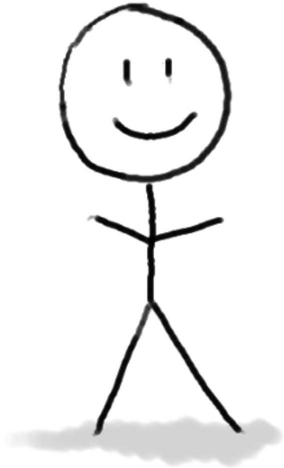
Why does efficiency matter?

- Time is precious
  - so much of why computers are useful is because **they compute quickly**

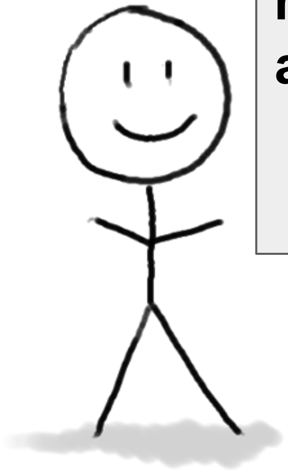
- Resources are limited
  - deployed software is often running at a massive scale where efficiencies there can have huge implications cost- and environment-wise (e.g. energy/electricity usage).

Why not measure runtime using time? (ms, s)?

Why not measure runtime using time? (ms, s)?

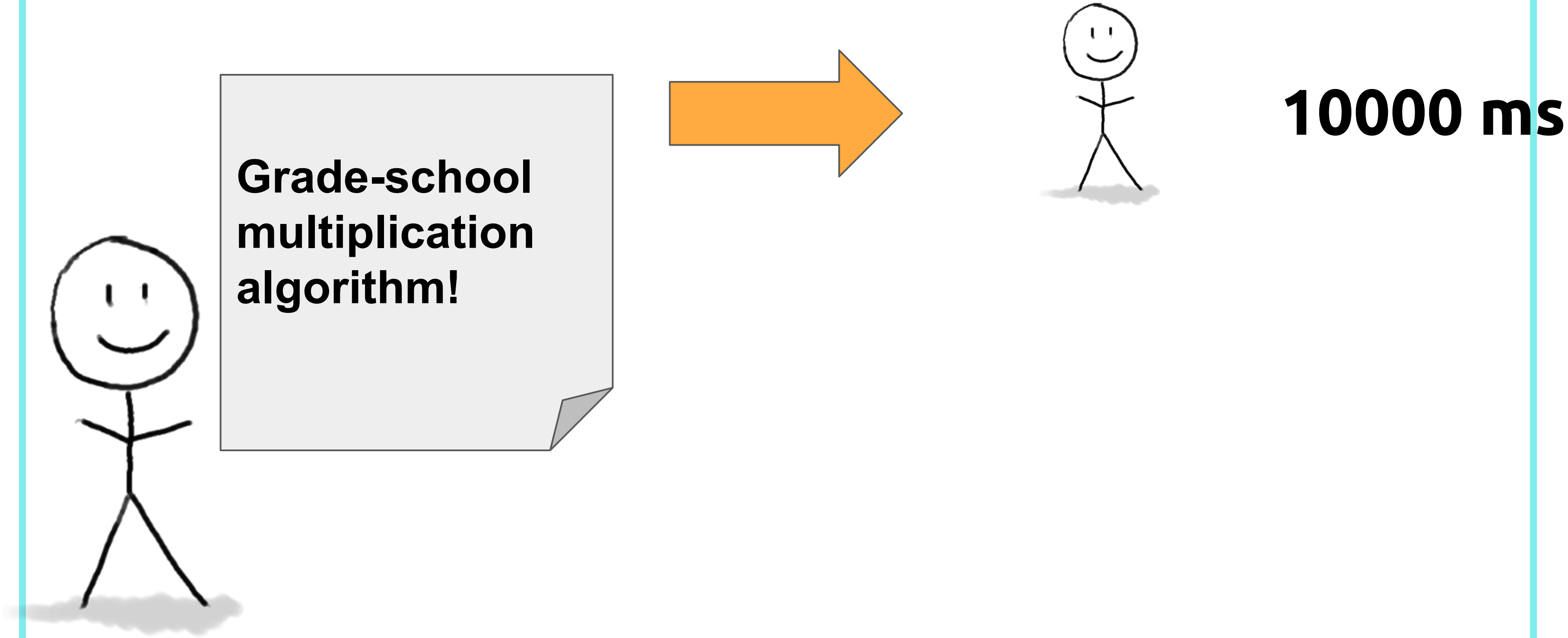


Why not measure runtime using time? (ms, s)?

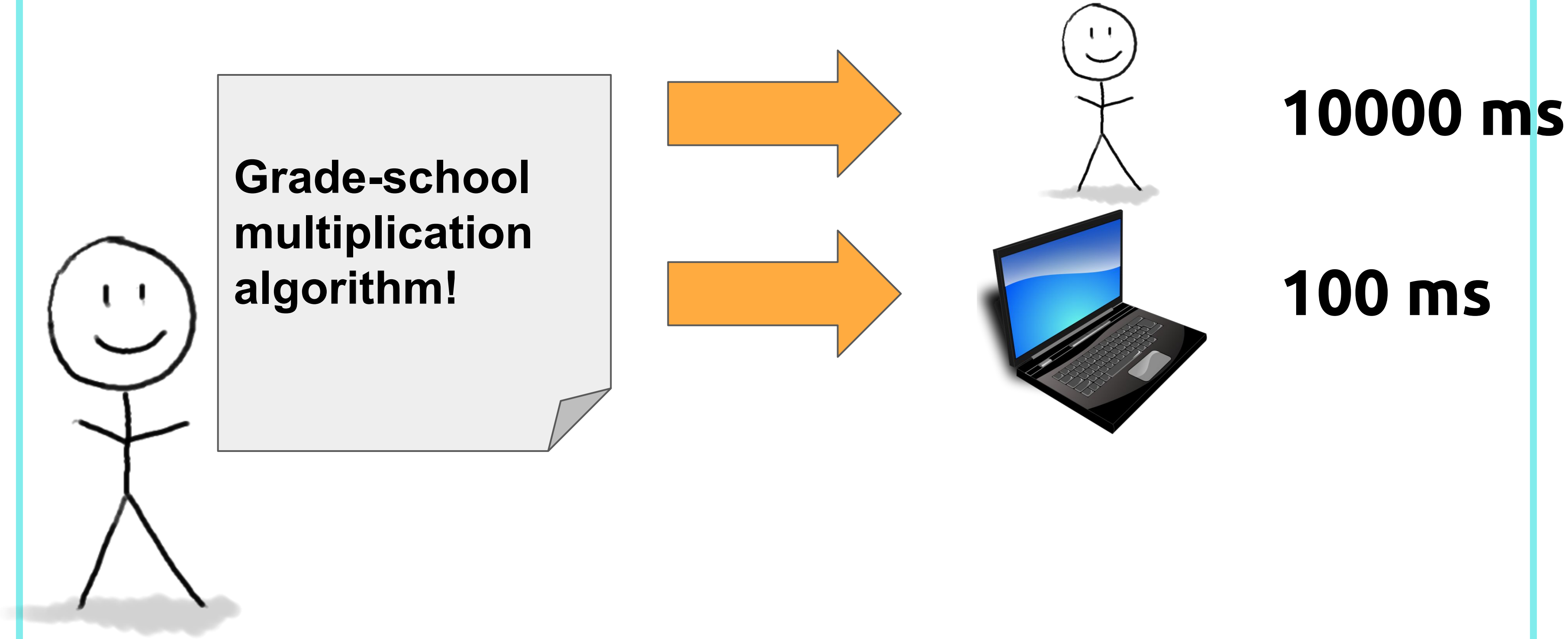


Grade-school multiplication algorithm!

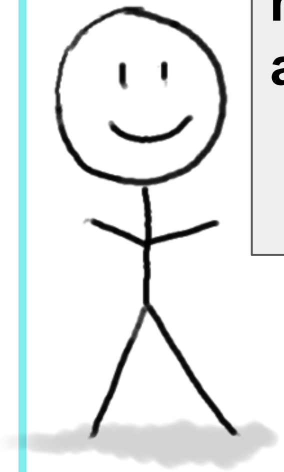
Why not measure runtime using time? (ms, s)?



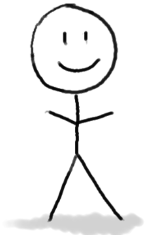
Why not measure runtime using time? (ms, s)?



Why not measure runtime using time? (ms, s)?



Grade-school multiplication algorithm!



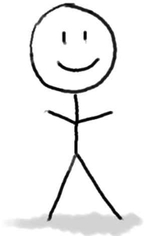
10000 ms

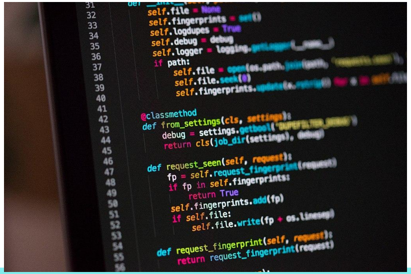
100 ms

1 ms

Why not measure runtime using time? (ms, s)?

Algorithm!





Pros and Cons of Doing This

Pros

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.
- Allows us to meaningfully compare how algorithms will perform on large inputs.

Cons

- Only makes sense if **n** is large (compared to the constant factors).
- Initially less intuitive? Weird notation? (It gets easier!)

Big Questions!

- How to multiply faster?
- What is Big-O? (informal)
- Why do we Big-O?
- How do we Big-O?

Big-O Process

Big-O Process

1. Define the “input size”
    - What’s our “n”?
    - Is it the length of the vector? Is it the value of an integer?
    - The inputs to the function are a good place to look!

Big-O Process

1. Define the "input size"
    - What's our "n"?
    - Is it the length of the vector? Is it the value of an integer?
    - The inputs to the function are a good place to look!
2. Count the number of operations
    - We've already practiced this!

Big-O Process

1. Define the "input size"
   - What's our "n"?
   - Is it the length of the vector? Is it the value of an integer?
   - The inputs to the function are a good place to look!
2. Count the number of operations
   - We've already practiced this!
3. Simplify
   - Some simplification rules we'll get into. (n -> inf!)

Concrete Examples

Count the number of operations

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

Count the number of operations

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. Define the “input size”
2. Count the number of operations
3. Simplify

Count the number of operations

```cpp
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. **Define the “input size”**
2. Count the number of operations
3. Simplify

Count the number of operations

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. **Define the "input size"** **The value of "number" variable**
2. Count the number of operations
3. Simplify

Count the number of operations

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. Define the “input size” **The value of “number” variable**
2. **Count the number of operations**
3. Simplify

Count the number of operations

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. Define the “input size” **The value of “number” variable**
2. **Count the number of operations** **4**
3. Simplify

Count the number of operations

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. Define the “input size”
2. Count the number of operations
3. **Simplify**

The value of “number” variable

4

Count the number of operations

```cpp
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. Define the "input size" **The value of "number" variable**
2. Count the number of operations **4**
3. **Simplify** **O(4)**

Count the number of operations

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

1. Define the "input size" **The value of "number" variable**
2. Count the number of operations **4**
3. **Simplify** **O(1)**

What’s the runtime?

```
void doThings(int number) {
 int x = 4;
 int y = x + y;
 std::cout << "hi" << std::endl;
 std::cout << number << std::endl;
}
```

O(1), constant time

What's the runtime?

```
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

1. Define the "input size" n
2. Count the number of operations
3. Simplify

What's the runtime?

```
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

1. **Define the "input size" n**
2. Count the number of operations
3. Simplify

What’s the runtime?

```
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

1. **Define the “input size” n** **The value of “start” variable**
2. Count the number of operations
3. Simplify

What's the runtime?

```
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

1. Define the "input size" n   **The value of "start" variable**
2. **Count the number of operations**
3. Simplify

What's the runtime?

```
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

1. Define the "input size" n
2. **Count the number of operations**
3. Simplify

The value of "start" variable

3N+4

What's the runtime?

```
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

1. Define the "input size" n
2. Count the number of operations
3. **Simplify**

The value of "start" variable

3N+4

What's the runtime?

```cpp
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

1. Define the "input size" n
2. Count the number of operations
3. **Simplify**

The value of "start" variable

3N+4

O(N)

What's the runtime?

```
void countDown(int start) {
 while(start >= 0) {
   std::cout << start << std::endl;
   start--;
 }
 std::cout << "Blast Off!" << std::endl;
}
```

O(N), linear time

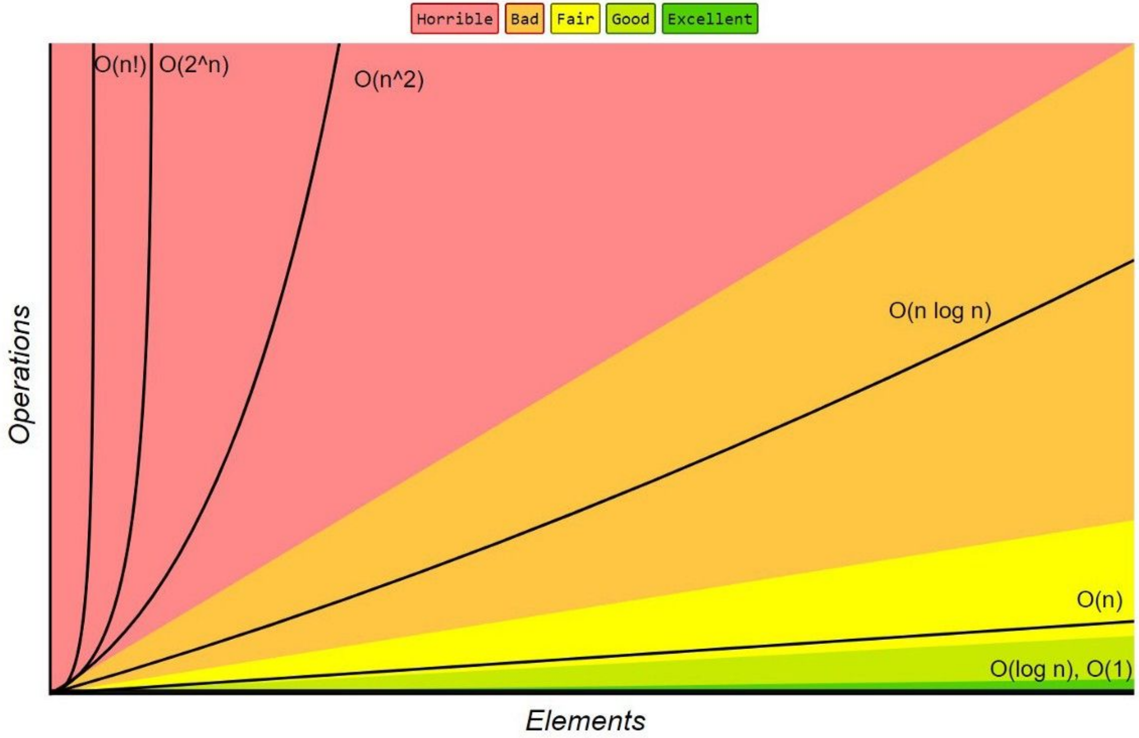
Common Big-O Classes

Fast

Slow

Class	How we say it	N = 1,000,000
O(1)	“constant time”	1
O(log N)	“logarithmic time”	6
O(N)	“linear time”	1,000,000
O(N * log N)	“N log N time”, but technically “quasilinear”	6,000,000
$O(N^2)$	“quadratic time”	1,000,000,000,000
$O(2^N)$	“exponential time”	99006562292958982506979236163019032507336242 4178756733286... (301,030 digits, will not fit here)
O(N!)	“factorial time”	82639316883312400623766461031726662911353479 789638730451... (5,565,709 digits, will not fit here)

Common Big-O Classes



Source: https://www.bigocheatsheet.com/

What’s the runtime?

```
void printElements(const std::vector<int>& vec) {
 std::cout << "Printing..." << std::endl;
 for(int i = 0; i < vec.size(); i++) {
   for(int j = 0; j < vec.size(); j++) {
     std::cout << vec[i] << " " << vec[j] << " ";
   }
 }
 std::cout << std::endl;
}
```

What's the runtime?

```
void printElements(const std::vector<int>& vec) {
  std::cout << "Printing..." << std::endl;
  for(int i = 0; i < vec.size(); i++) {
    for(int j = 0; j < vec.size(); j++) {
      std::cout << vec[i] << " " << vec[j] << " ";
    }
  }
  std::cout << std::endl;
}
```

1. Define the "input size"
2. Count the number of operations
3. Simplify

Simplification Rules

Simplification Rules

1. Simplify constant time:
   - 23 -> O(1)

Simplification Rules

1. Simplify constant time:
    - 23 -> O(1)
2. Drop multiplicative constants
    - 7 * N -> O(N)

Simplification Rules

1. Simplify constant time:
   - 23 -> O(1)
2. Drop multiplicative constants
   - 7 * N -> O(N)
3. Drop all lower-order terms:
   - $N + N^2$ -> $O(N^2)$

What’s the runtime?

```
void printElements(const std::vector<int>& vec) {
 std::cout << "Printing..." << std::endl;
 for(int i = 0; i < vec.size(); i++) {
   for(int j = 0; j < 10; j++) {
     std::cout << vec[i] << " " << vec[j] << " ";
   }
 }
 std::cout << std::endl;
}
```

HW1 RELEASED BY EOD!

First coding homework with coding!
Start early so can debug issues!

Can you?

- **Describe**
  - the value of algorithmic analysis
  - the common complexity classes and their relative sizes

- **Practice**
  - determining the time complexities of algorithms using Big-O notation
  - determining the space complexity of algorithms using Big-O notation

How was the pace today?

Wrap-Up

- Big-O gives us a way to **evaluate algorithms** through their **time** and **space complexity**

- **Common complexity classes**
  - $O(1) < O(N) < O(N \log N) < O(N^2) < O(2^N) < O(N!)$

- **Big-O process**:
  - (1) Define the input size, (2) count the operations, (3) simplify terms

Announcements

- **HW 0 is due!**
  - Due tonight 08/25 @ 11:59PM
- **HW 1 will be out!**
  - By EoD!
  - Due 09/06 @ 1:59PM
- **First quiz on Tuesday**!

Next time!

- Formal big-O introduction!
- Introduction to other asymptotic analyses besides Big-O!
- More time/space comlexity practice with and without recursion

COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 2: Measuring Performance I

Chris Lucas (cflucas@ncat.edu)