CSE 373: Asymptotic Analysis

Michael Lee

Monday Jan 8, 2017

Warmup

Warmup

Remind your neighbor: what fields did our array list iterator need?

Comparing algorithms

Goal: *compare* algorithms

Comparing algorithms

Goal: *compare* algorithms

What are we comparing? Lots of metrics we could pick!

- Time needed to run
- Memory used
- Number of network calls made
- Amount of data we save to the disk
- etc...

Comparing algorithms

Goal: *compare* algorithms

What are we comparing? Lots of metrics we could pick!

- Time needed to run
- Memory used
- Number of network calls made
- Amount of data we save to the disk
- etc...

(Some metrics are intangible: clarity, security... Hard to measure those.)

Comparing algorithms

Goal: *compare* algorithms

What are we comparing? Lots of metrics we could pick!

- Time needed to run
- Memory used
- Number of network calls made
- Amount of data we save to the disk
- etc...

(Some metrics are intangible: clarity, security... Hard to measure those.)

Today: focus on comparing algorithms based on *how long it takes them to run in the worst case*.

An idea: let's time our algorithms!

Goal: find the number of primes below n

An idea: let's time our algorithms!

Goal: find the number of primes below n

Time taken for $n = 18000$

Algorithm	Time (in ms)
Algo 1	0.0018
Algo 2	35.58
Algo 3	100.75

An idea: let's time our algorithms!

Goal: find the number of primes below n

Time taken for $n = 18000$

Algorithm	Time (in ms)
Algo 1	0.0018
Algo 2	35.58
Algo 3	100.75

Which algorithm is better?

An idea: let's time our algorithms!

Goal: find the number of primes below *n*

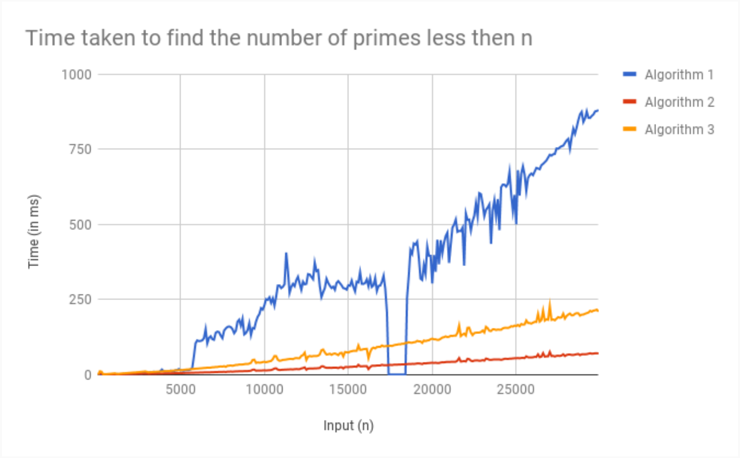
Time taken for $n = 18000$

Algorithm	Time (in ms)
Algo 1	0.0018
Algo 2	35.58
Algo 3	100.75

~~Which algorithm is better?~~

This is a trick question. Why isn't this table enough to let us decide which algorithm is better?

An idea: let's time our algorithms!



Which algorithm is better?

Our goal

What we want:

- To see **overall** trends as input increases
  (considering a single data point isn't useful)

Our goal

What we want:

- To see **overall** trends as input increases
  (considering a single data point isn't useful)
- Final result is independent of incidental factors
  (CPU speed, other programs that may be running, battery life, programming language, coding tricks...)

Our goal

What we want:

- To see **overall** trends as input increases
  (considering a single data point isn't useful)
- Final result is independent of incidental factors
  (CPU speed, other programs that may be running, battery life, programming language, coding tricks...)
- Rigorously discover overall trends without resorting to testing
  (what if we miss worst-case input? best-case input?)

Our goal

What we want:

- To see **overall** trends as input increases
  (considering a single data point isn't useful)
- Final result is independent of incidental factors
  (CPU speed, other programs that may be running, battery life, programming language, coding tricks...)
- Rigorously discover overall trends without resorting to testing
  (what if we miss worst-case input? best-case input?)
- A way to analyze before coding!

Our process

Two step process:

1. **Model** what we care about as a mathematical function
2. **Analyze** that function using asymptotic analysis

Modeling: Assumptions

Assumption: basic operations take “constant” time

- Arithmetic (for fixed-width numbers)
- Variable assignment
- Accessing a field or array index
- Printing something out
- etc...

Modeling: Assumptions

Assumption: basic operations take "constant" time

- Arithmetic (for fixed-width numbers)
- Variable assignment
- Accessing a field or array index
- Printing something out
- etc...

Warning: These assumptions are over-simplifications.

But they're very useful approximations!

Modeling: Complex statements

- **Consecutive statements**
  Sum of time of each statement

Modeling: Complex statements

- **Consecutive statements**
  Sum of time of each statement
- **Function calls**
  Time of function's body

Modeling: Complex statements

- **Consecutive statements**
  Sum of time of each statement
- **Function calls**
  Time of function's body
- **Conditionals**
  Time of condition + max(if branch, else branch)

Modeling: Complex statements

- **Consecutive statements**
  Sum of time of each statement
- **Function calls**
  Time of function's body
- **Conditionals**
  Time of condition + max(if branch, else branch)
- **Loops**
  Number of iterations $\times$ time for loop body

Modeling: exercise

Goal: return 'true' if a **sorted** array of ints contains duplicates

Modeling: exercise

Goal: return 'true' if a **sorted** array of ints contains duplicates

Algorithm 1: compare each pair of elements

```
public boolean hasDuplicate1(int[] array) {
    for (int i = 0; i < array.length; i++)
        for (int j = 0; j < array.length; j++)
            if (i != j && array[i] == array[j])
                return true;
    return false;
}
```

Modeling: exercise

Goal: return 'true' if a **sorted** array of ints contains duplicates

Algorithm 1: compare each pair of elements

```java
public boolean hasDuplicate1(int[] array) {
    for (int i = 0; i < array.length; i++)
        for (int j = 0; j < array.length; j++)
            if (i != j && array[i] == array[j])
                return true;
    return false;
}
```

Algorithm 2: compare each consecutive pairs of elements

```java
public boolean hasDuplicate2(int[] array) {
    for (int i = 0; i < array.length - 1; i++)
        if (array[i] == array[i + 1])
            return true;
    return false;
}
```

Modeling: exercise

Goal: return 'true' if a **sorted** array of ints contains duplicates

Algorithm 1: compare each pair of elements

```java
public boolean hasDuplicate1(int[] array) {
    for (int i = 0; i < array.length; i++)
        for (int j = 0; j < array.length; j++)
            if (i != j && array[i] == array[j])
                return true;
    return false;
}
```

Algorithm 2: compare each consecutive pairs of elements

```java
public boolean hasDuplicate2(int[] array) {
    for (int i = 0; i < array.length - 1; i++)
        if (array[i] == array[i + 1])
            return true;
    return false;
}
```

Exercise: create a **mathematical function** modeling the amount of time taken in the worst case

Our process

Two step process:

1. **Model** what we care about as a mathematical function
2. **Analyze** that function using asymptotic analysis

Our process

Two step process:

1. **Model** what we care about as a mathematical function
2. **Analyze** that function using asymptotic analysis
   Specifically: have a way to **compare** two functions

Next time

Next time: how do we compare functions?