Introduction: Computing Runtimes

Daniel Kane

Department of Computer Science and Engineering
University of California, San Diego

Algorithmic Design and Techniques
Algorithms and Data Structures

Learning Objectives

- Describe some of the issues involved with computing the runtime of an actual program.
- Understand why finding exact runtimes is a problem.

Outline

1. Revisit Fibonacci

2. Other Things to Consider

Runtime Analysis

Function FibList(n)

```
create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i-1] + F[i-2]
return F[n]
```

Runtime Analysis

Function FibList(n)

```
create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

$2n + 2$ lines of code. Does this really describe the runtime of the algorithm?

Individual Lines

```
Function FibList(n)
create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i−1] + F[i−2]
return F[n]
```

Depends on memory management system.

Individual Lines

```
Function FibList(n)

create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Assignment.

Individual Lines

```
Function FibList(n)

create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Assignment.

Individual Lines

```
Function FibList(n)

create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Increment, comparison, branch.

Individual Lines

Function FibList(n)

```
create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

Lookup, assignment, addition of big integers.

Individual Lines

Function FibList(n)

```
create an array F[0...n]
F[0] ← 0
F[1] ← 1
for i from 2 to n:
    F[i] ← F[i−1] + F[i−2]
return F[n]
```

Lookup, return.

Outline

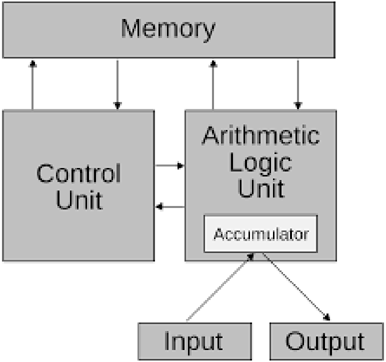
1. Revisit Fibonacci
2. Other Things to Consider

Computing Runtime

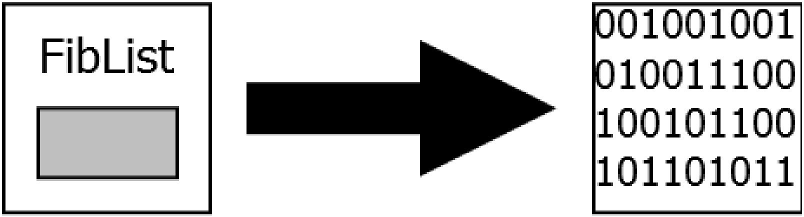
To figure out how long this simple program would actually take to run on a real computer, we would also need to know things like:

Speed of the Computer

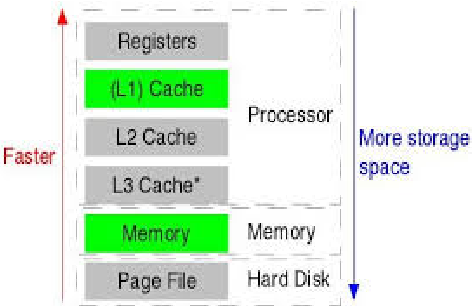
The System Architecture



The Compiler Being Used



Details of the Memory Hierarchy



Problem

- Figuring out accurate runtime is a huge mess

Problem

- Figuring out accurate runtime is a huge mess
- In practice, you might not even know some of these details

Goal

Want to:

- Measure runtime without knowing these details.
- Get results that work for large inputs.

Introduction: Asymptotic Notation

Daniel Kane

Department of Computer Science and Engineering
University of California, San Diego

Algorithmic Design and Techniques
Algorithms and Data Structures

Learning Objectives

- Understand the basic idea behind asymptotic runtimes.
- Describe some of the advantages to using asymptotic runtimes.

Last Time

Computing Runtimes Hard

- Depends on fine details of program.
- Depends on details of computer.

Idea

All of these issues can multiply runtimes by (large) constant.

Idea

All of these issues can multiply runtimes by (large) constant. So measure runtime in a way that ignores constant multiples.

Problem

Unfortunately, 1 second, 1 hour, 1 year only differ by constant multiples.

Solution

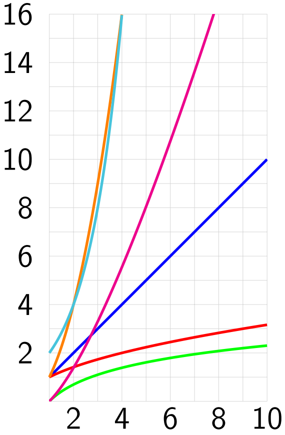
Consider asymptotic runtimes. How does runtime scale with input size.

Approximate Runtimes

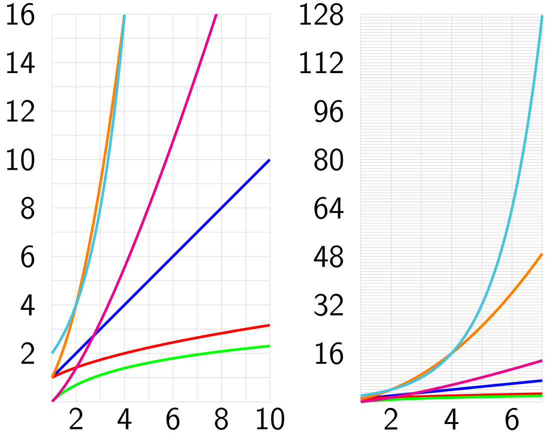
	n	$n \log n$	n^2	2^n
$n = 20$	1 sec	1 sec	1 sec	1 sec
$n = 50$	1 sec	1 sec	1 sec	13 day
$n = 10^2$	1 sec	1 sec	1 sec	$4 \cdot 10^{13}$ year
$n = 10^6$	1 sec	1 sec	17 min	
$n = 10^9$	1 sec	30 sec	30 year	
max n	10^9	$10^{7.5}$	$10^{4.5}$	30

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$

$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$



$$\log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec 2^n$$



Introduction: Big-O Notation

Daniel Kane

Department of Computer Science and Engineering
University of California, San Diego

Algorithmic Design and Techniques
Algorithms and Data Structures

Learning Objectives

- Understand the meaning of Big-O notation.
- Describe some of the advantages and disadvantages of using Big-O notation.

Big-O Notation

Definition

$f(n) = O(g(n))$ (f is Big-O of g) or $f \preceq g$ if there exist constants N and c so that for all $n \geq N$, $f(n) \leq c \cdot g(n)$.

Big-O Notation

Definition

$f(n) = O(g(n))$ (f is Big-O of g) or $f \preceq g$ if there exist constants N and c so that for all $n \geq N$, $f(n) \leq c \cdot g(n)$.

f is bounded above by some constant multiple of g.

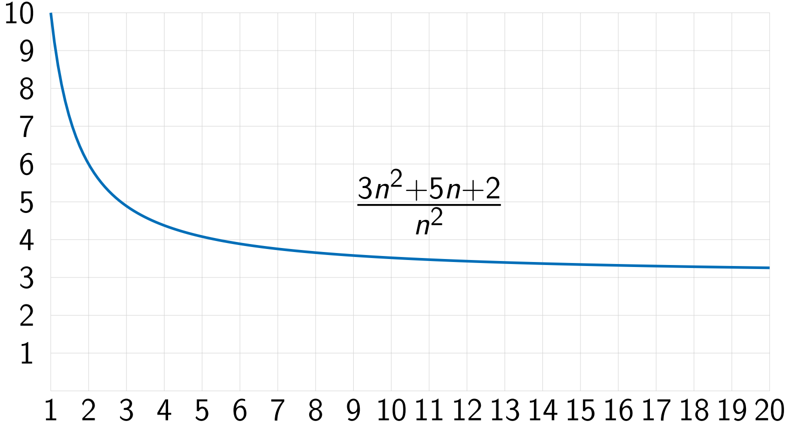
Big-O Notation

Example

$3n^2 + 5n + 2 = O(n^2)$ since if $n \geq 1$,
$3n^2 + 5n + 2 \leq 3n^2 + 5n^2 + 2n^2 = 10n^2$.

Growth Rate

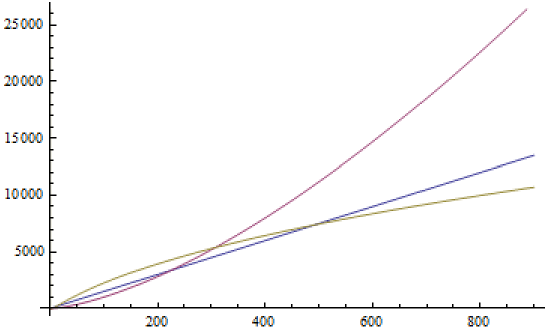
$3n^2 + 5n + 2$ has the same growth rate as n^2



Using Big-O

We will use Big-O notation to report algorithm runtimes. This has several advantages.

Clarifies Growth Rate



Cleans up Notation

- $O(n^2)$ vs. $3n^2 + 5n + 2$.
- $O(n)$ vs. $n + \log_2(n) + \sin(n)$.

Cleans up Notation

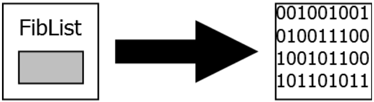
- $O(n^2)$ vs. $3n^2 + 5n + 2$.
- $O(n)$ vs. $n + \log_2(n) + \sin(n)$.
- $O(n \log(n))$ vs. $4n \log_2(n) + 7$.
    - Note: $\log_2(n), \log_3(n), \log_x(n)$ differ by constant multiples, don't need to specify which.

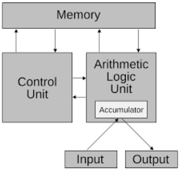
Cleans up Notation

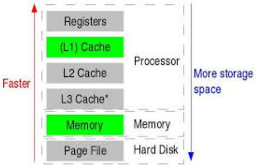
- $O(n^2)$ vs. $3n^2 + 5n + 2$.
- $O(n)$ vs. $n + \log_2(n) + \sin(n)$.
- $O(n \log(n))$ vs. $4n \log_2(n) + 7$.
  - Note: $\log_2(n), \log_3(n), \log_x(n)$ differ by constant multiples, don't need to specify which.
- Makes algebra easier.

Can Ignore Complicated Details

No longer need to worry about:







Warning

- Using Big-O loses important information about constant multiples.
- Big-O is *only* asymptotic.

Introduction: Using Big-O

Daniel Kane

Department of Computer Science and Engineering
University of California, San Diego

Algorithmic Design and Techniques
Algorithms and Data Structures

Learning Objectives

- Manipulate expressions involving Big-O and other asymptotic notation.
- Compute algorithm runtimes in terms of Big-O.

Big-O Notation

Definition

$f(n) = O(g(n))$ (f is Big-O of g) or $f \preceq g$ if there exist constants N and c so that for all $n \geq N$, $f(n) \leq c \cdot g(n)$.

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3),\ \frac{n^2}{3} = O(n^2)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3),\ \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2),\ \sqrt{n} = O(n)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3),\ \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2),\ \sqrt{n} = O(n)$$

$n^a \prec b^n$ $(a > 0, b > 1)$:

$$n^5 = O(\sqrt{2}^n),\ n^{100} = O(1.1^n)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3),\ \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2),\ \sqrt{n} = O(n)$$

$n^a \prec b^n$ $(a > 0, b > 1)$:

$$n^5 = O(\sqrt{2}^n),\ n^{100} = O(1.1^n)$$

$(\log n)^a \prec n^b$ $(a, b > 0)$:

$$(\log n)^3 = O(\sqrt{n}),\ n \log n = O(n^2)$$

Common Rules

Multiplicative constants can be omitted:

$$7n^3 = O(n^3),\ \frac{n^2}{3} = O(n^2)$$

$n^a \prec n^b$ for $0 < a < b$:

$$n = O(n^2),\ \sqrt{n} = O(n)$$

$n^a \prec b^n$ $(a > 0, b > 1)$:

$$n^5 = O(\sqrt{2}^n),\ n^{100} = O(1.1^n)$$

$(\log n)^a \prec n^b$ $(a, b > 0)$:

$$(\log n)^3 = O(\sqrt{n}),\ n \log n = O(n^2)$$

Smaller terms can be omitted :

$$n^2 + n = O(n^2),\ 2^n + n^9 = O(2^n)$$

Recall Algorithm

Function FibList(n)

create an array $F[0 \ldots n]$
$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for i from 2 to n:
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

Big-O in Practice

Operation	Runtime

Big-O in Practice

Operation	Runtime
create an array $F[0 \ldots n]$	$O(n)$

Big-O in Practice

Operation	Runtime
create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$

Big-O in Practice

Operation	Runtime
create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$

Big-O in Practice

Operation	Runtime
create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times

Big-O in Practice

Operation	Runtime
create an array $F[0 \dots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times
$\quad F[i] \leftarrow F[i-1] + F[i-2]$	$O(n)$

Big-O in Practice

Operation	Runtime
create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times
$\quad F[i] \leftarrow F[i-1] + F[i-2]$	$O(n)$
return $F[n]$	$O(1)$

Big-O in Practice

Operation	Runtime
create an array $F[0 \ldots n]$	$O(n)$
$F[0] \leftarrow 0$	$O(1)$
$F[1] \leftarrow 1$	$O(1)$
for i from 2 to n:	Loop $O(n)$ times
$\quad F[i] \leftarrow F[i-1] + F[i-2]$	$O(n)$
return $F[n]$	$O(1)$

Total:

$$O(n)+O(1)+O(1)+O(n)\cdot O(n)+O(1) = O(n^2).$$

Other Notation

Definition

For functions $f, g : \mathbb{N} \to \mathbb{R}^+$ we say that:

- $f(n) = \Omega(g(n))$ or $f \succeq g$ if for some c, $f(n) \geq c \cdot g(n)$ (f grows no slower than g).
- $f(n) = \Theta(g(n))$ or $f \asymp g$ if $f = O(g)$ and $f = \Omega(g)$ (f grows at the same rate as g).

Other Notation

Definition

For functions $f, g : \mathbb{N} \to \mathbb{R}^+$ we say that:

- $f(n) = o(g(n))$ or $f \prec g$ if $f(n)/g(n) \to 0$ as $n \to \infty$ (f grows slower than g).

Asymptotic Notation

- Lets us ignore messy details in analysis.
- Produces clean answers.
- Throws away a lot of practically useful information.