Intro: Why Study Algorithms?

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

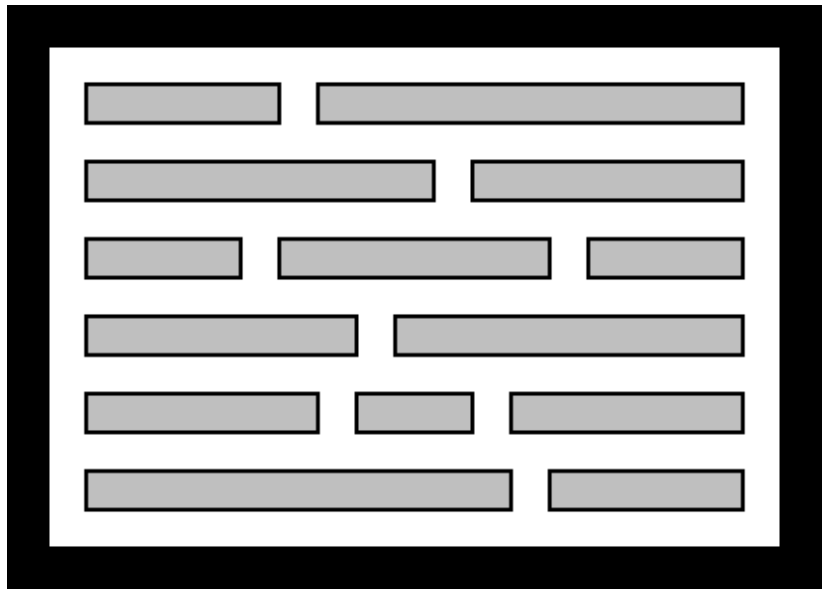
Learning Objectives

- Understand the type of problem that will be covered in this class.
- Recognize some problems for which sophisticated algorithms might not be necessary.
- Describe some artificial intelligence problems which go beyond the scope of this course.

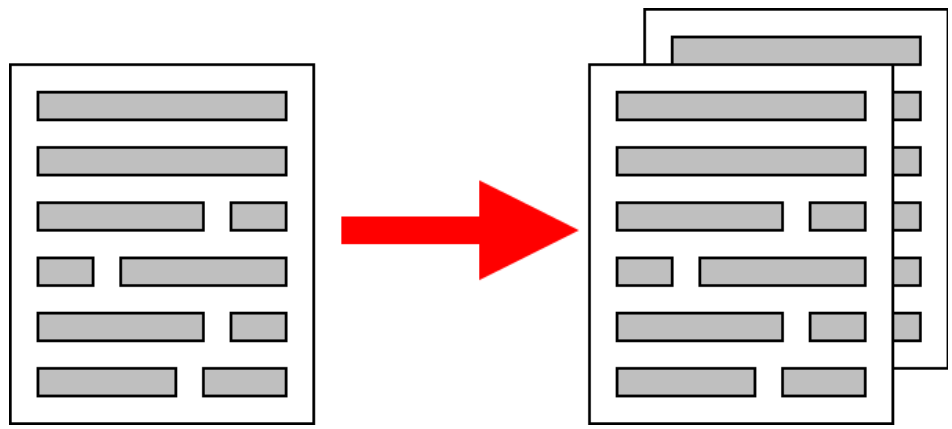
Straightforward Programming Problems

- Has straightforward implementation.
- Natural solution is already efficient.

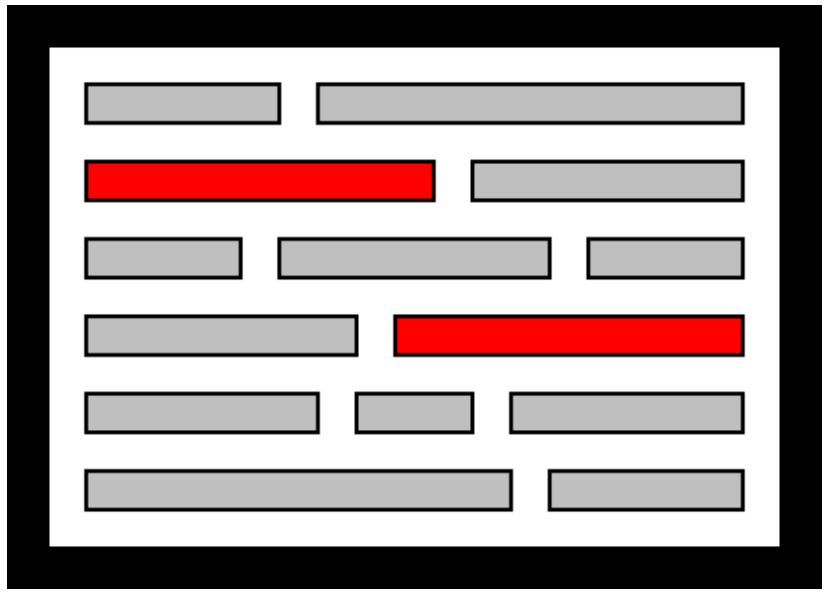
Display given text



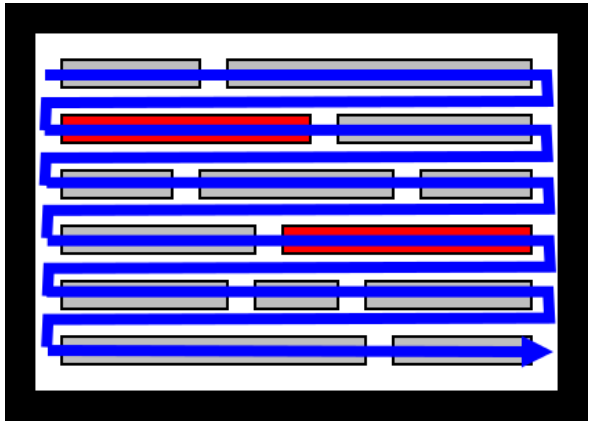
Copy a File



Search for a Given Word



Search for a Given Word



Linear Scan.

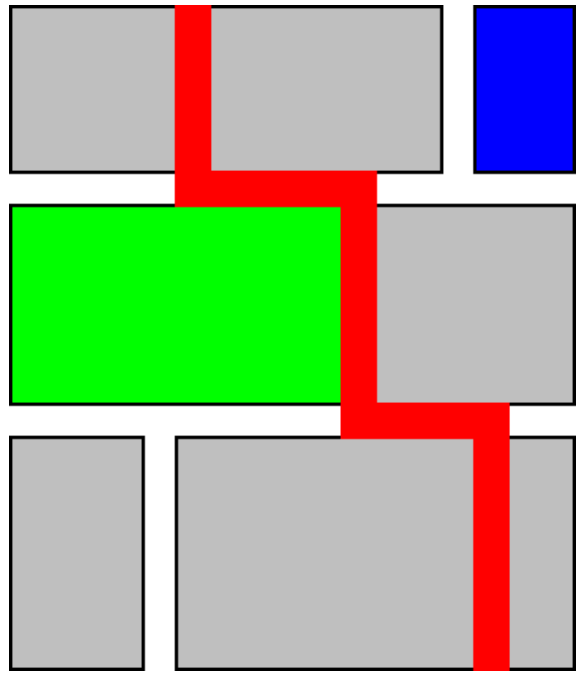
Simple Programming Problems

- Has linear scan.
- Cannot do much better.
- The obvious program works.

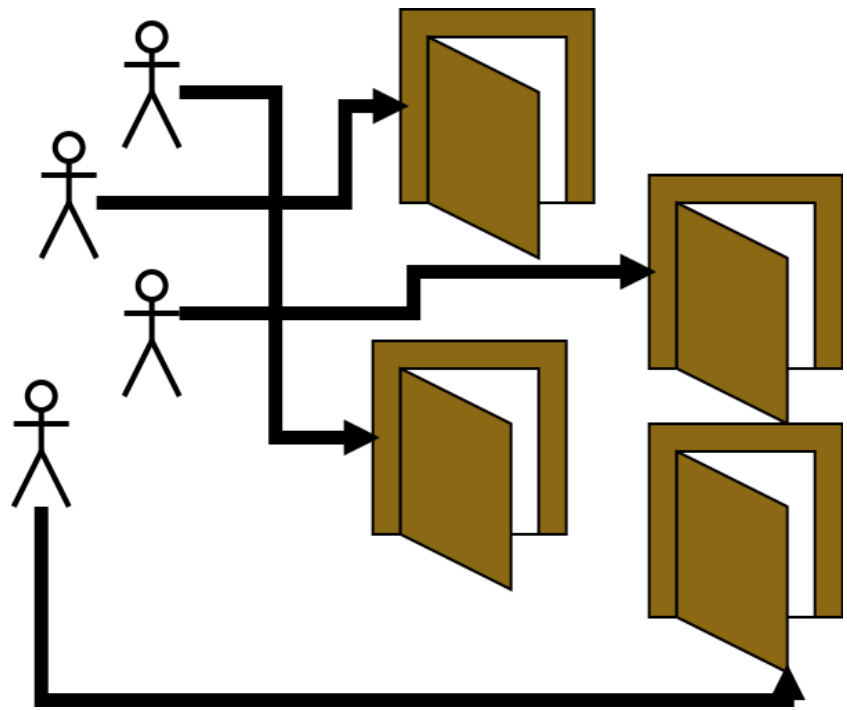
Algorithms Problems

Not so clear what to do.

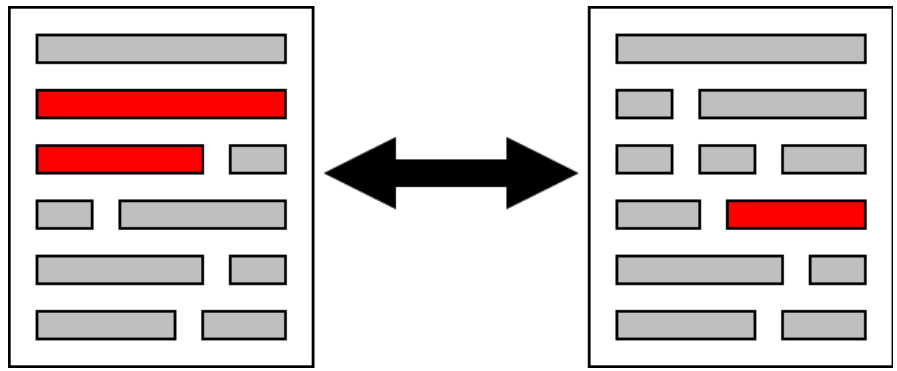
Find the Shortest Path Between Locations



Find the Best Assignment of Students to Dorm Rooms



Measure Similarity of Documents



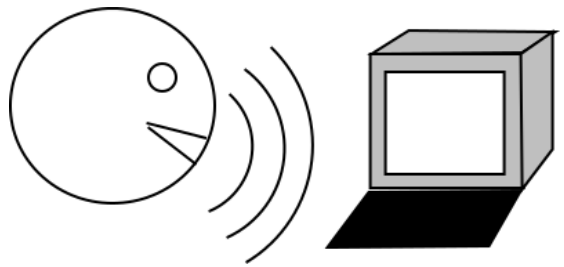
Algorithms Problems

- Not clear how to do
- Simple ideas too slow
- Room for optimization

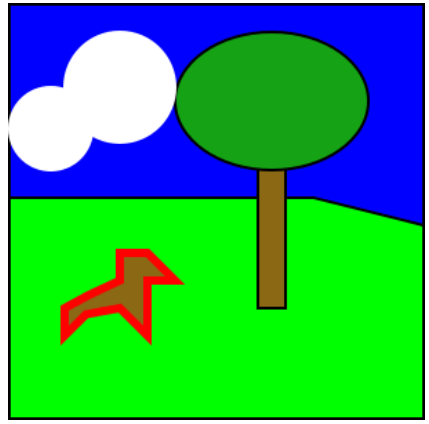
Artificial Intelligence Problems

Hard to even clearly state.

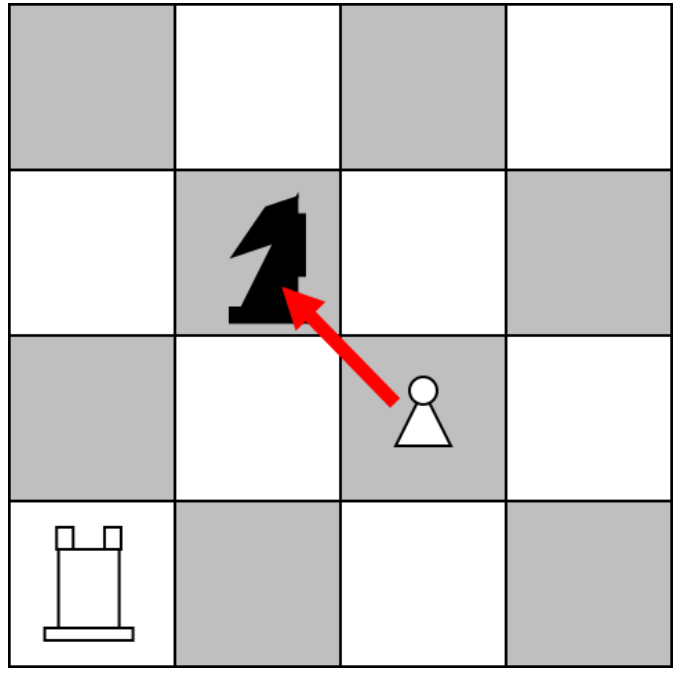
Understand Natural Language



Identify Objects In Photographs



Play Games Well



What We'll Cover

Focus on algorithms problems.

- Clearly formulated.
- Hard to do efficiently.

Intro: Coming Up

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

Coming Up

Cover two algorithm problems:

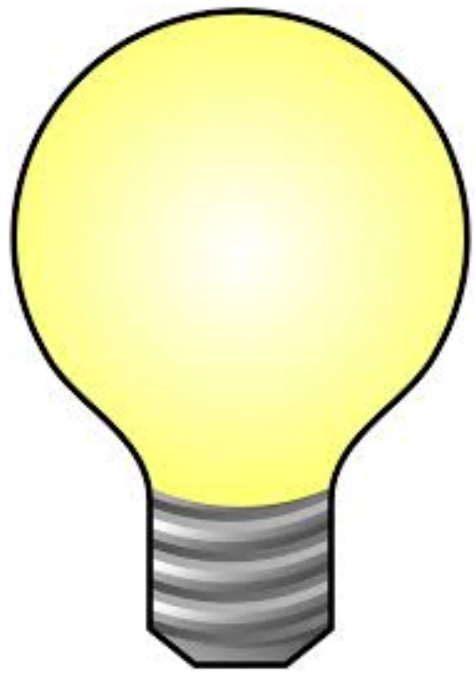
- Fibonacci Numbers
- Greatest Common Divisors

Examples of why algorithms are important.

Straightforward Algorithm

Take Too Long

Slightly More Complicated Algorithm



That is Very Fast

Intro: Fibonacci Numbers

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

Learning Objectives

- Understand the definition of the Fibonacci numbers.
- Show that the naive algorithm for computing them is slow.
- Efficiently compute large Fibonacci numbers.

Outline

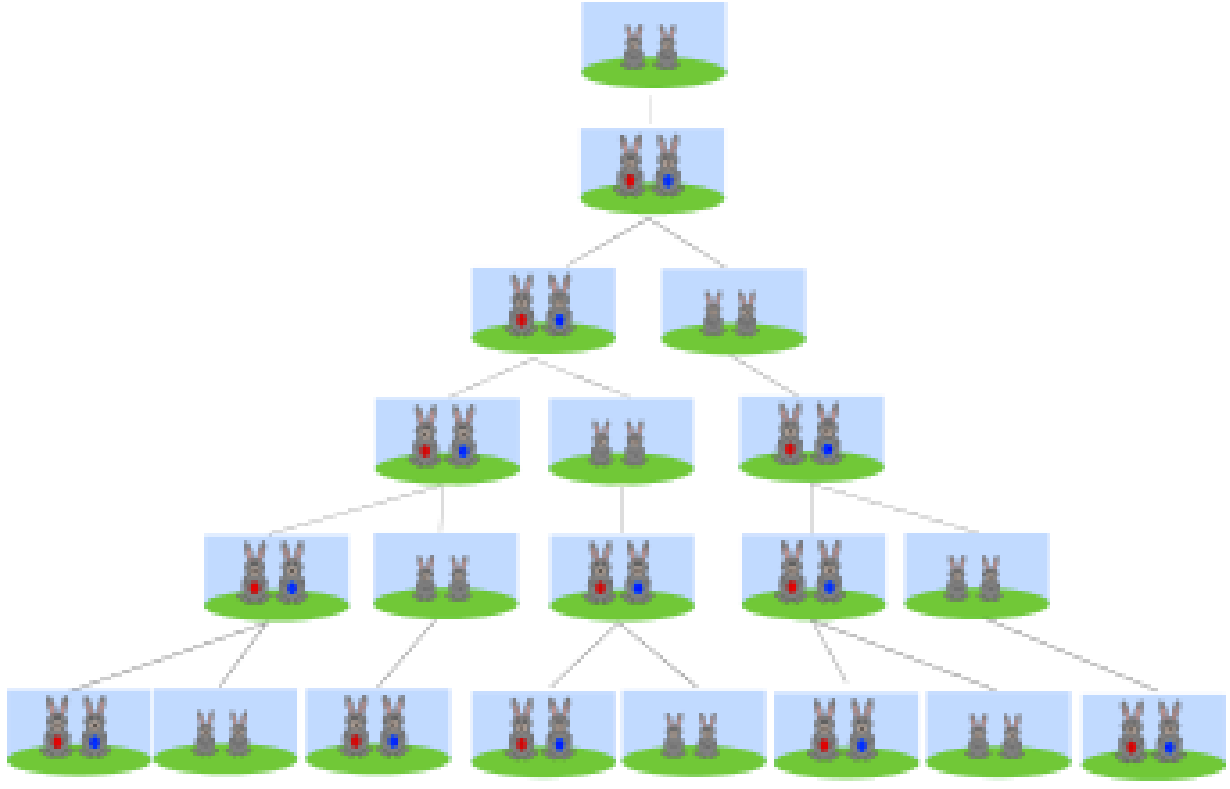
1. Problem Overview
2. Naive Algorithm
3. Efficient Algorithm

Definition

$$F_n = \begin{cases} 0, & n = 0\,, \\ 1, & n = 1\,, \\ F_{n-1} + F_{n-2}, & n > 1\,. \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Developed to Study Rabbit Populations



Rapid Growth

Lemma

$F_n \geq 2^{n/2}$ for $n \geq 6$.

Proof

By induction

Base case: $n = 6, 7$ (by direct computation).

Inductive step:

$$F_n = F_{n-1} + F_{n-2} \geq 2^{(n-1)/2} + 2^{(n-2)/2} \geq 2 \cdot 2^{(n-2)/2} = 2^{n/2}. \quad \square$$

Formula

Theorem

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right).$$

Example

$$
\begin{aligned}
F_{20} &= 6765 \\
F_{50} &= 12586269025 \\
F_{100} &= 354224848179261915075 \\
F_{500} &= 139423224561697880139724382870407283950070256587697307264108962948325571622863290691557658876222521294125
\end{aligned}
$$

Computing Fibonacci numbers

Compute F_n

Input: An integer $n \geq 0$.

Output: F_n.

Outline

1. Problem Overview
2. Naive Algorithm
3. Efficient Algorithm

Algorithm

FibRecurs(n)

```
if n ≤ 1:
    return n
else:
    return FibRecurs(n − 1) + FibRecurs(n − 2)
```

Running time

Let $T(n)$ denote the number of lines of code executed by `FibRecurs`(n).

If $n \leq 1$

FibRecurs(n)

```
if n ≤ 1:
  return n
else:
  return FibRecurs(n − 1) + FibRecurs(n − 2)
```

$T(n) = 2.$

If $n \geq 2$

FibRecurs(n)

```
if n ≤ 1:
    return n
else:
    return FibRecurs(n − 1) + FibRecurs(n − 2)
```

$$T(n) = 3 + T(n-1) + T(n-2).$$

Running Time

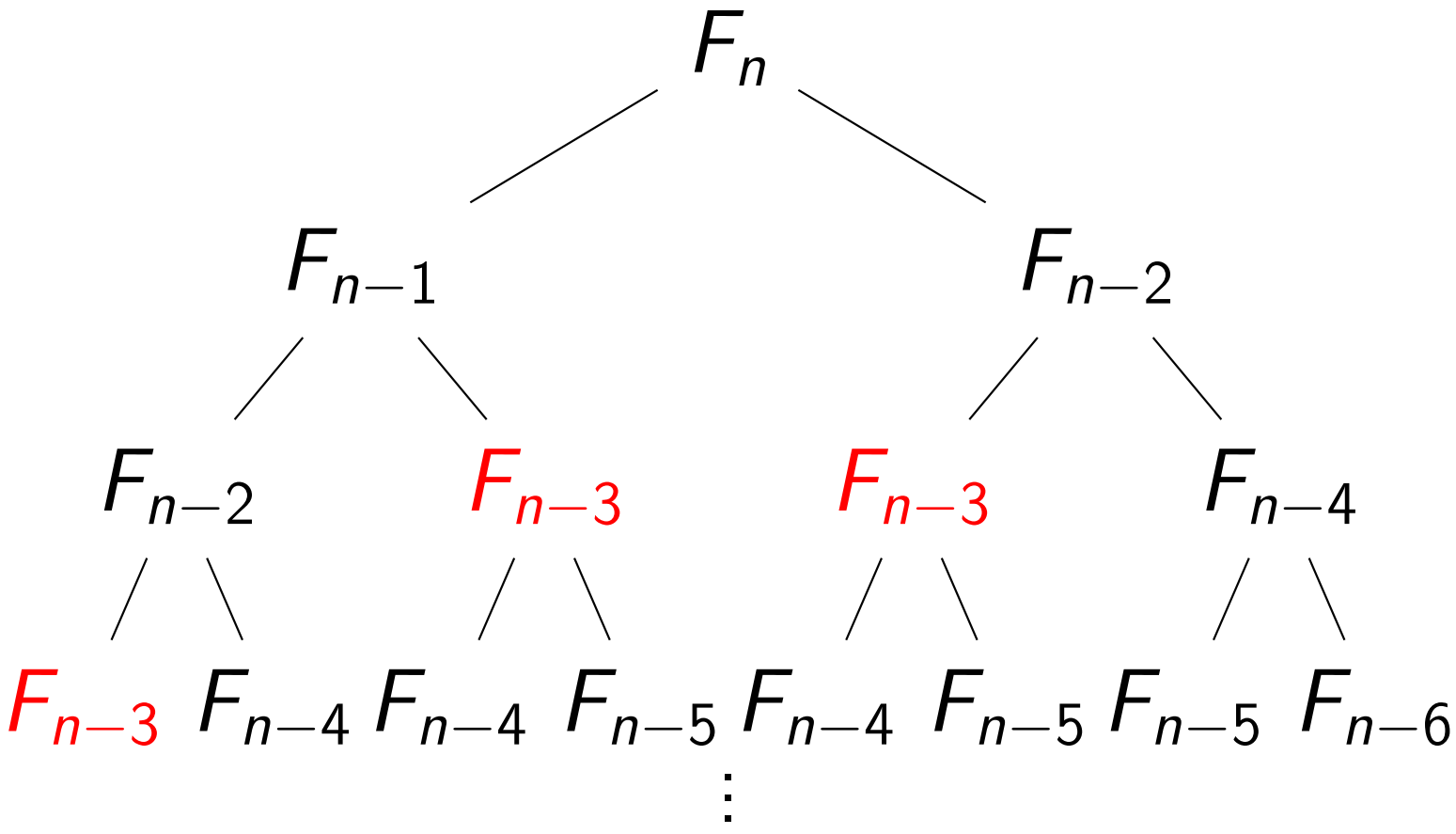
$$T(n) = \begin{cases} 2 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + 3 & \text{else.} \end{cases}$$

Therefore $T(n) \geq F_n$

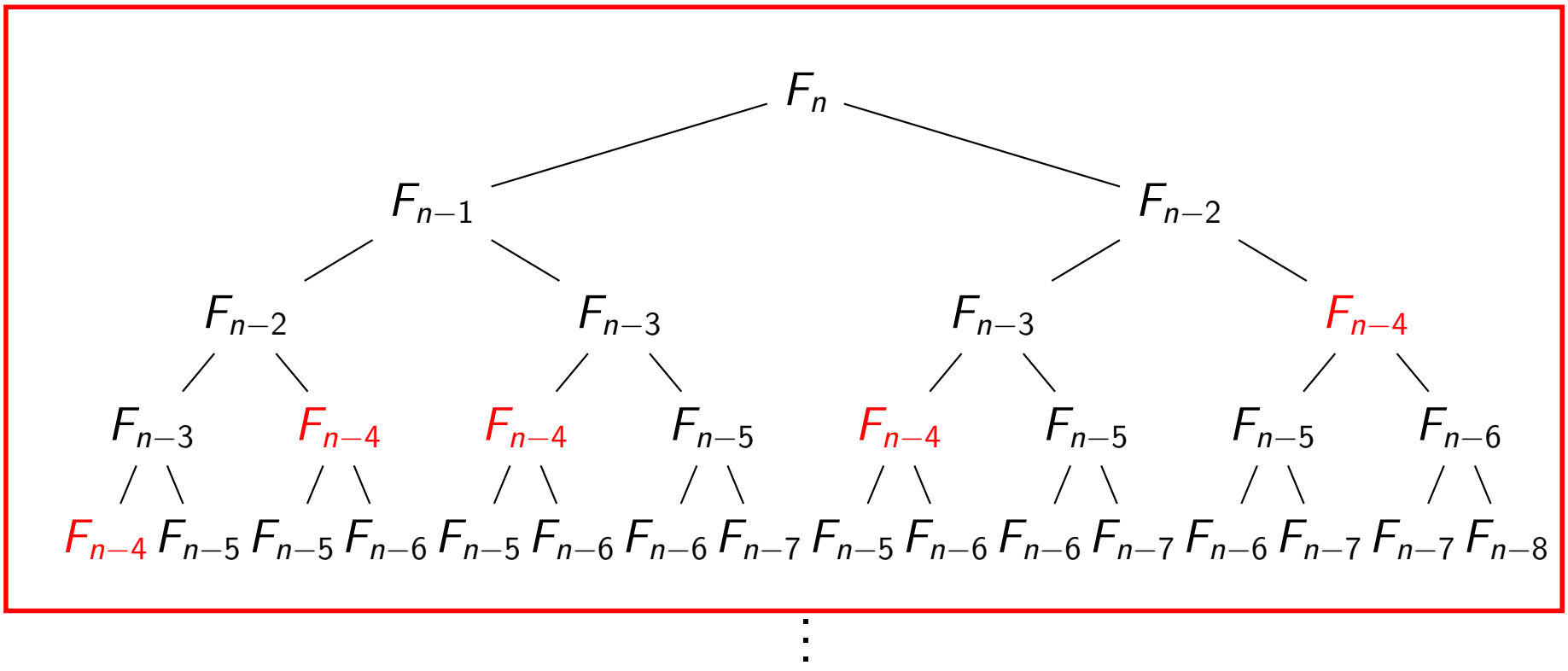
$T(100) \approx 1.77 \cdot 10^{21}$ (1.77 sextillion)

Takes 56,000 years at 1GHz.

Why so slow?



Why so slow?


F_n
F_{n-1} F_{n-2}
F_{n-2} F_{n-3} F_{n-3} F_{n-4}
F_{n-3} F_{n-4} F_{n-4} F_{n-5} F_{n-4} F_{n-5} F_{n-5} F_{n-6}
F_{n-4} F_{n-5} F_{n-5} F_{n-6} F_{n-5} F_{n-6} F_{n-6} F_{n-7} F_{n-5} F_{n-6} F_{n-6} F_{n-7} F_{n-6} F_{n-7} F_{n-7} F_{n-8}

Outline

1. Problem Overview
2. Naive Algorithm
3. Efficient Algorithm

Another Algorithm

Imitate hand computation:

0, 1, 1, 2, 3, 5, 8

$0 + 1 = 1$

$1 + 1 = 2$

$1 + 2 = 3$

$2 + 3 = 5$

$3 + 5 = 8$

New Algorithm

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]
```

- $T(n) = 2n + 2$. So $T(100) = 202$.
- Easy to compute.

Summary

- Introduced Fibonacci numbers.
- Naive algorithm takes ridiculously long time on small examples.
- Improved algorithm incredibly fast.

Moral: The right algorithm makes all the difference.

Intro: Greatest Common Divisors I

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

Learning Objectives

- Define greatest common divisors.
- Compute greatest common divisors inefficiently.

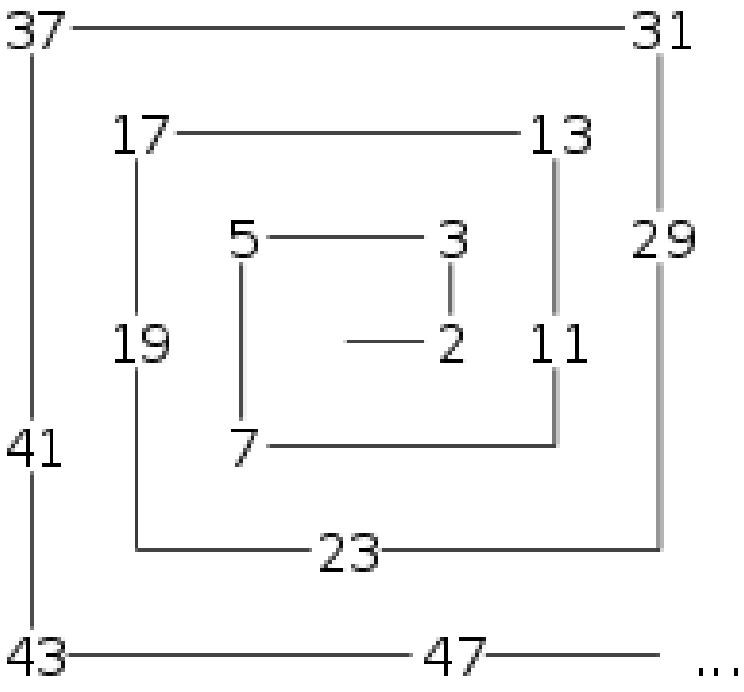
GCDs

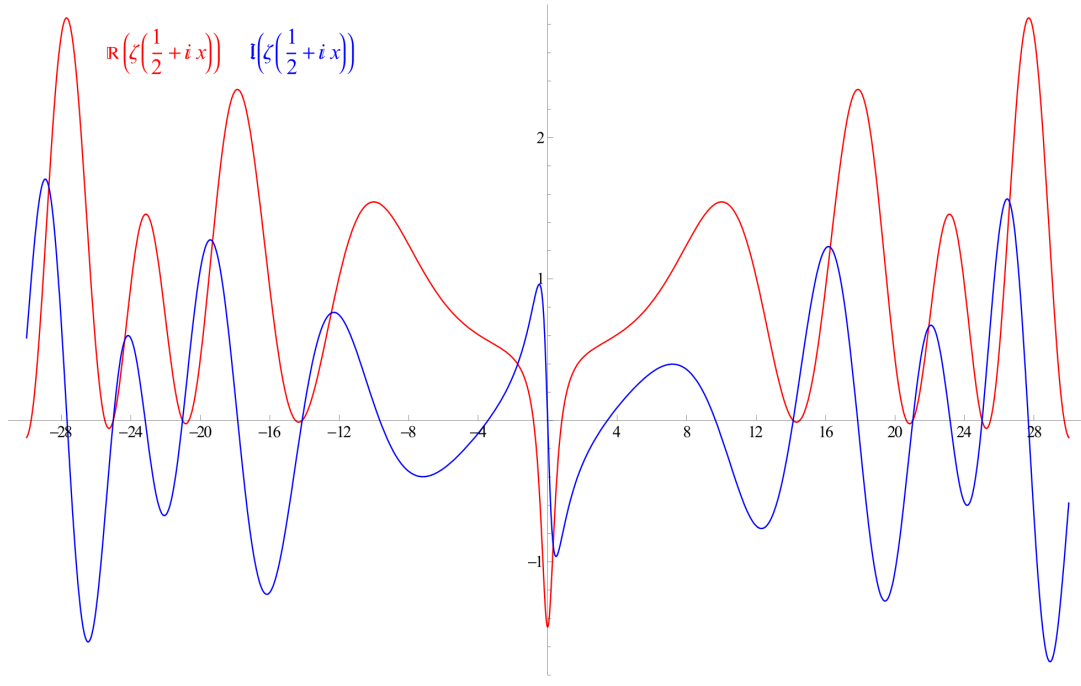
- Put fraction $\frac{a}{b}$ in simplest form.
- Divide numerator and denominator by d, to get $\frac{a/d}{b/d}$.
  - Need d to divide a and b.
  - Want d to be as large as possible.

Definition

For integers, a and b, their greatest common divisor or $\gcd(a, b)$ is the largest integer d so that d divides both a and b.

Number Theory





Cryptography

Computation

Compute GCD

Input: Integers $a, b \geq 0$.

Output: $\gcd(a, b)$.

Run on large numbers like

$$\gcd(3918848, 1653264).$$

Naive Algorithm

Function NaiveGCD(a, b)

```
best ← 0
for d from 1 to a + b:
    if d|a and d|b:
        best ← d
return best
```

- Runtime approximately $a + b$.
- Very slow for 20 digit numbers.

Intro: Greatest Common Divisors II

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

Learning Objectives

- Implement the Euclidean Algorithm.
- Approximate the runtime.

GCDs

Definition

For integers, a and b, their greatest common divisor or $\gcd(a, b)$ is the largest integer d so that d divides both a and b.

Compute GCD

Input: Integers $a, b \geq 0$.

Output: $\gcd(a, b)$.

Key Lemma

Lemma

Let a' be the remainder when a is divided by b, then

$$\gcd(a, b) = \gcd(a', b) = \gcd(b, a').$$

Proof

Proof (sketch)

- $a = a' + bq$ for some q
- d divides a and b if and only if it divides a' and b

Euclidean Algorithm

```
Function EuclidGCD(a, b)
if b = 0:
    return a
a' ← the remainder when a is
     divided by b
return EuclidGCD(b, a')
```

Produces correct result by Lemma.

Example

$$
\begin{aligned}
&\gcd(3918848, 1653264)\\
=&\gcd(1653264, 612320)\\
=&\gcd(612320, 428624)\\
=&\gcd(428624, 183696)\\
=&\gcd(183696, 61232)\\
=&\gcd(61232, 0)\\
=&61232.
\end{aligned}
$$

Runtime

- Each step reduces the size of numbers by about a factor of 2.
- Takes about $\log(ab)$ steps.
- GCDs of 100 digit numbers takes about 600 steps.
- Each step a single division.

Summary

- Naive algorithm is too slow.
- The correct algorithm is much better.
- Finding the correct algorithm requires knowing something interesting about the problem.

Intro: Computing Runtimes

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

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]
```

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

Assignment.

Individual Lines

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 $F[n]$

Assignment.

Individual Lines

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 $F[n]$

Increment, comparison, branch.

Individual Lines

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 $F[n]$

Lookup, assignment, addition of big integers.

Individual Lines

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 $F[n]$

Lookup, return.

Outline

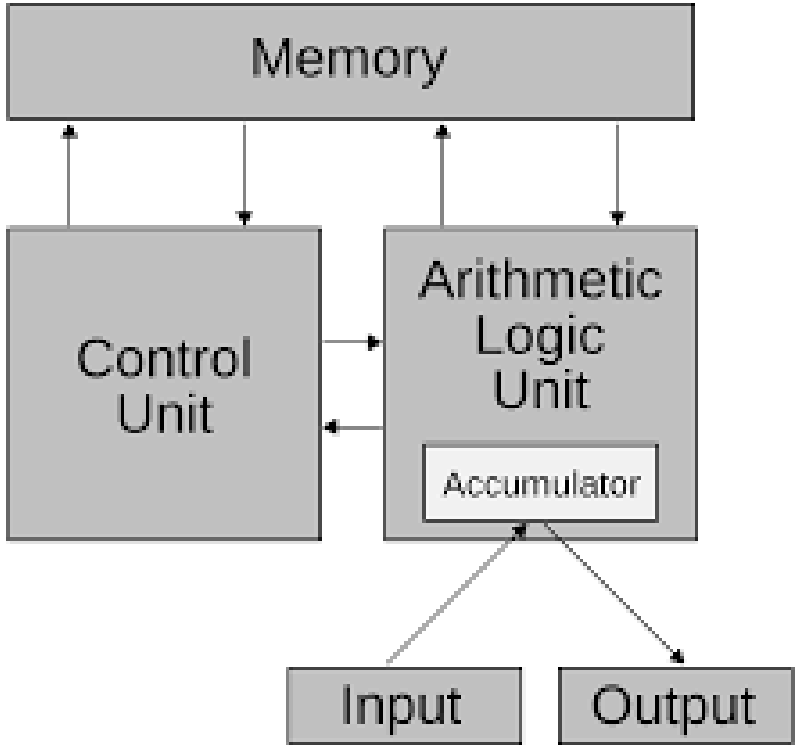
1. Revisit Fibonacci
2. Other Things to Consider

Computing Runtime

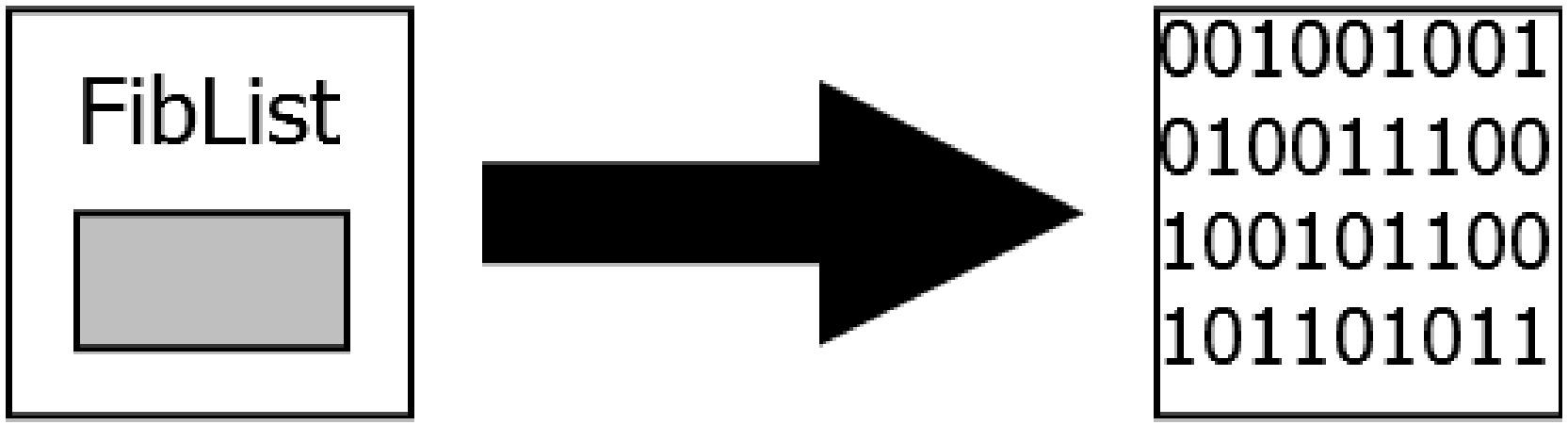
To figure out how long this simple program would actually take to run on a real computer, we would also need to know things like:

Speed of the Computer

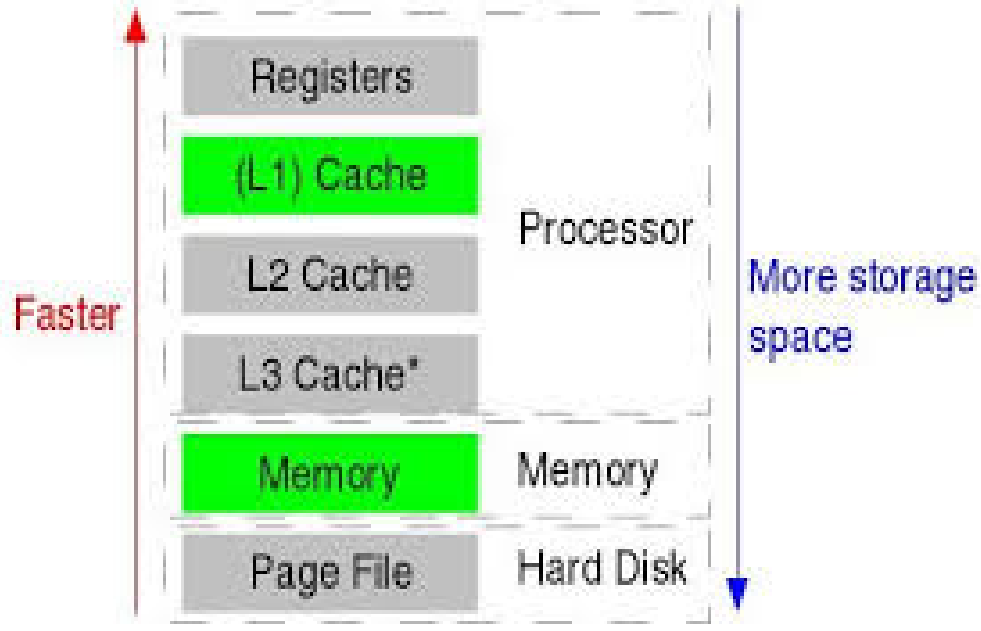
The System Architecture



The Compiler Being Used



Details of the Memory Hierarchy



Problem

- Figuring out accurate runtime is a huge mess
- 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.

Intro: Asymptotic Notation

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

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. So measure runtime in a way that ignores constant multiples.

Problem

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

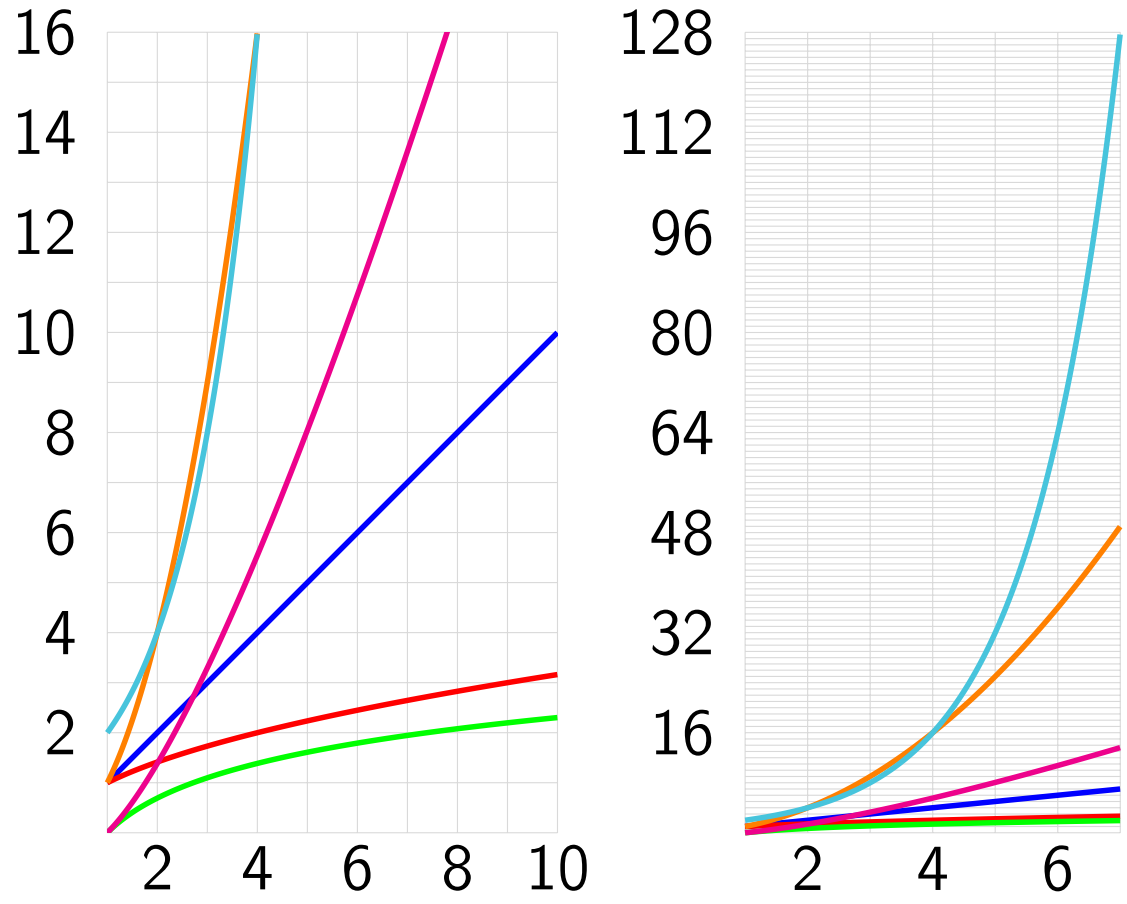
Solution

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

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



Intro: Big-O Notation

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

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 $n \geq N$, $f(n) \leq c \cdot g(n)$.

f is bounded above by some constant multiple of g.

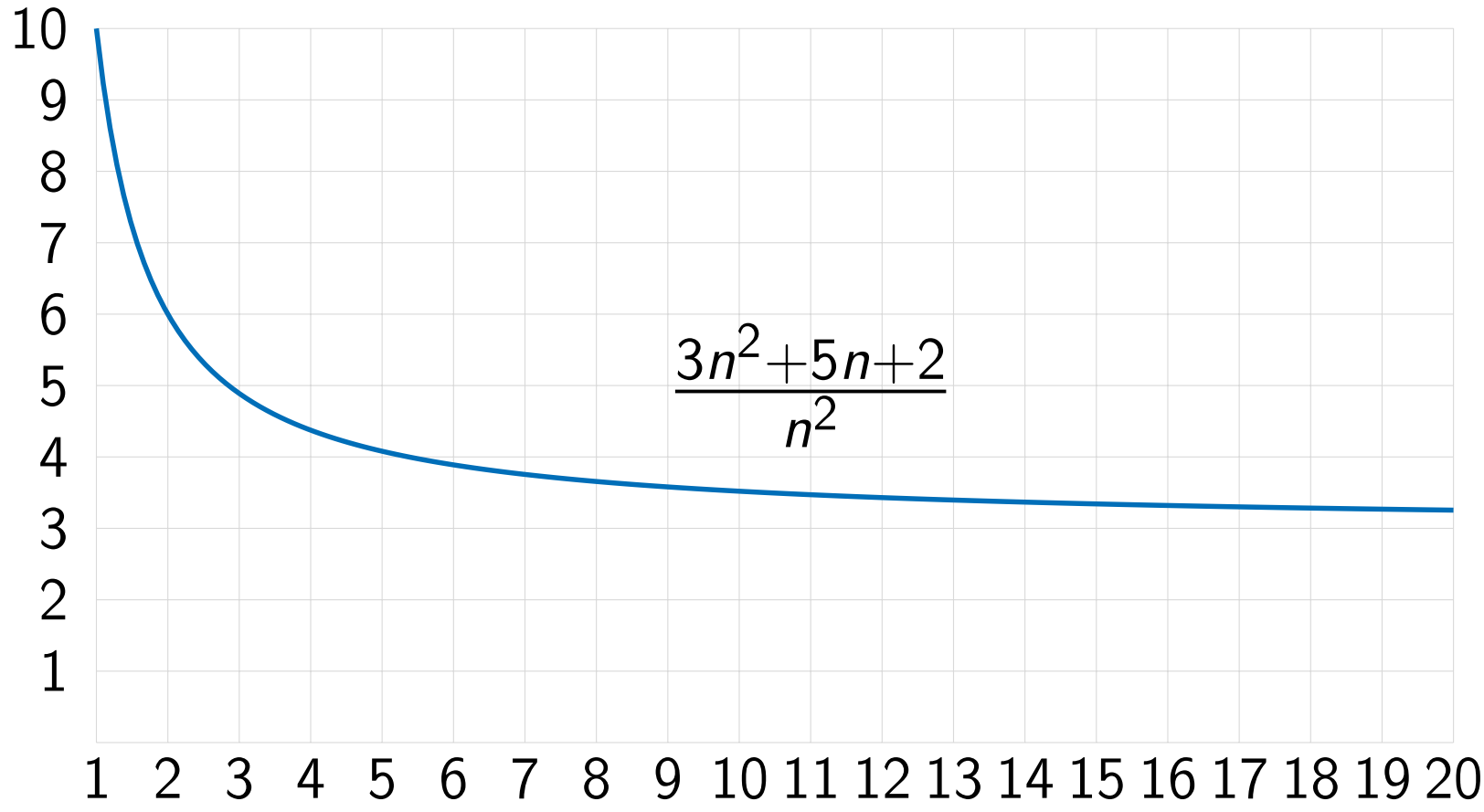
Big-O Notation

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

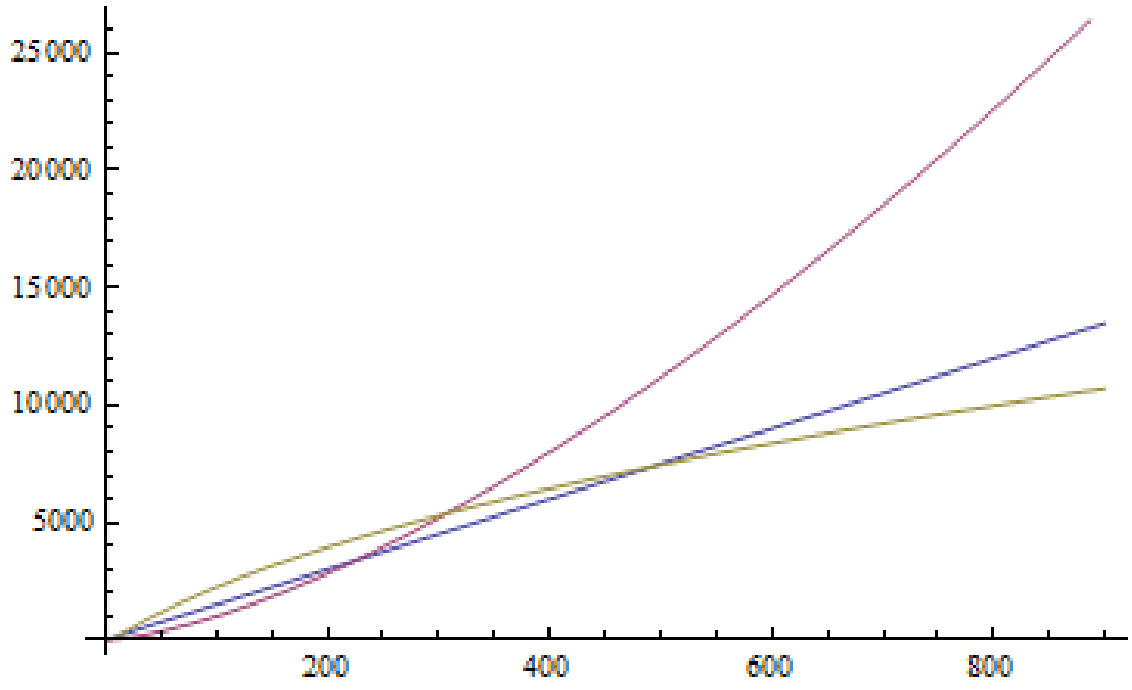
$3n^2 + 5n + 2$ has the same growth rate as n^2



Using Big-O

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

Clarifies Growth Rate

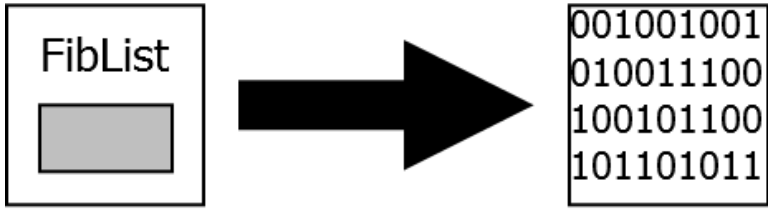


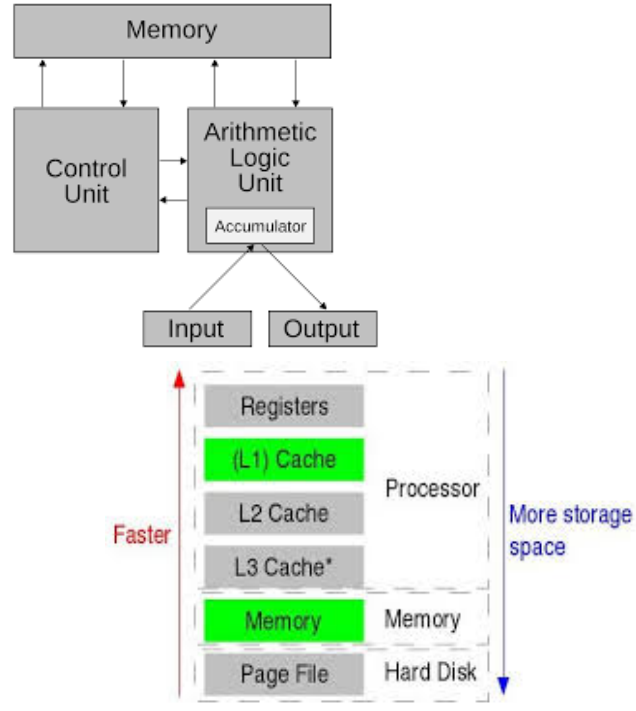
Cleans up Notation

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

Intro: Using Big-O

Daniel Kane

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

Data Structures and Algorithms
Algorithmic Toolbox

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:

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

$n^a \prec n^b$ 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
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:	

add two big number, proportion to n

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

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.

Intro: Course Overview

Daniel Kane

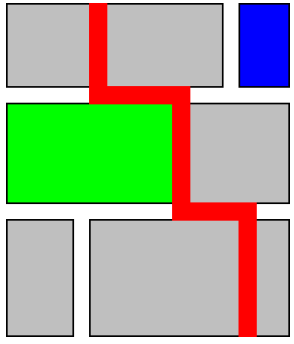
Department of Computer Science and Engineering
University of California, San Diego

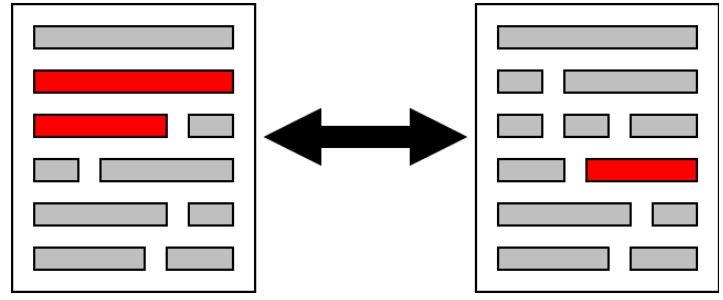
Data Structures and Algorithms
Algorithmic Toolbox

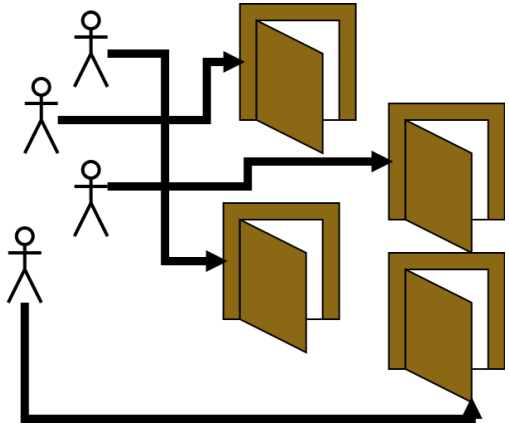
Algorithm Design is Hard

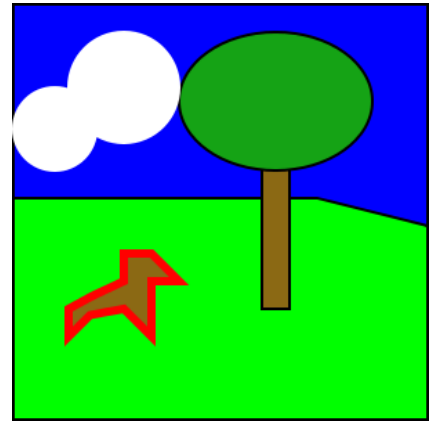
- Algorithms very general.
- No generic procedure for designing good algorithms.
- Finding good algorithms often requires coming up with unique insights.

Algorithms Solve Many Different Problems

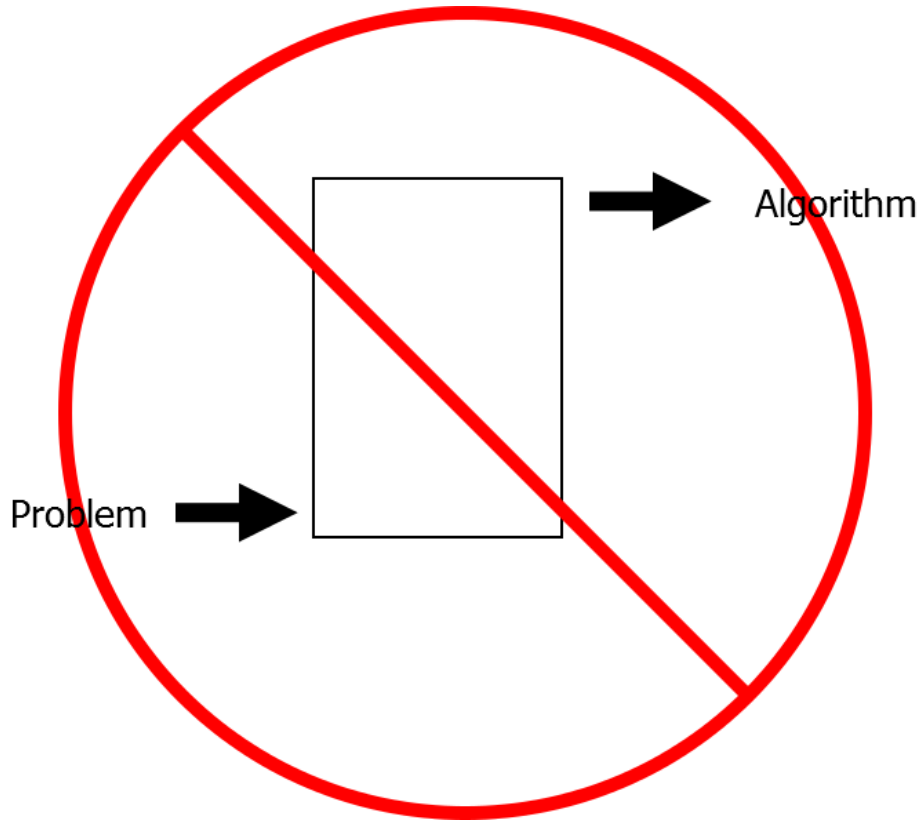




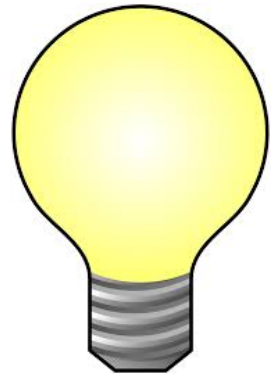




No Generic Procedure to Create Algorithms



Finding Algorithm Often Requires Unique Insights



Toolbox

What can we teach you?

- Practice designing algorithms.
- Common tools used in algorithm design.
- We will discuss three of the most common algorithmic design techniques:
  - Greedy Algorithms
  - Divide and Conquer
  - Dynamic Programming

Levels of Design

Naive Algorithm: Definition to algorithm. Slow.

Algorithm by way of standard Tools: Standard techniques.

Optimized Algorithm: Improve existing algorithm.

Magic Algorithm: Unique insight.

The Rest of the Course

- Each unit covers a technique.
- Exercises help build intuition.