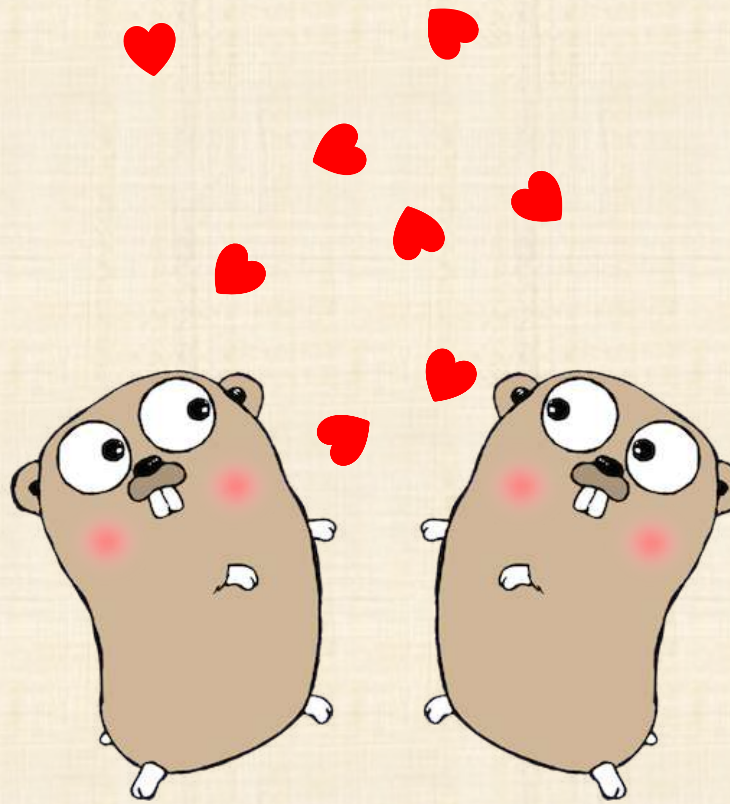
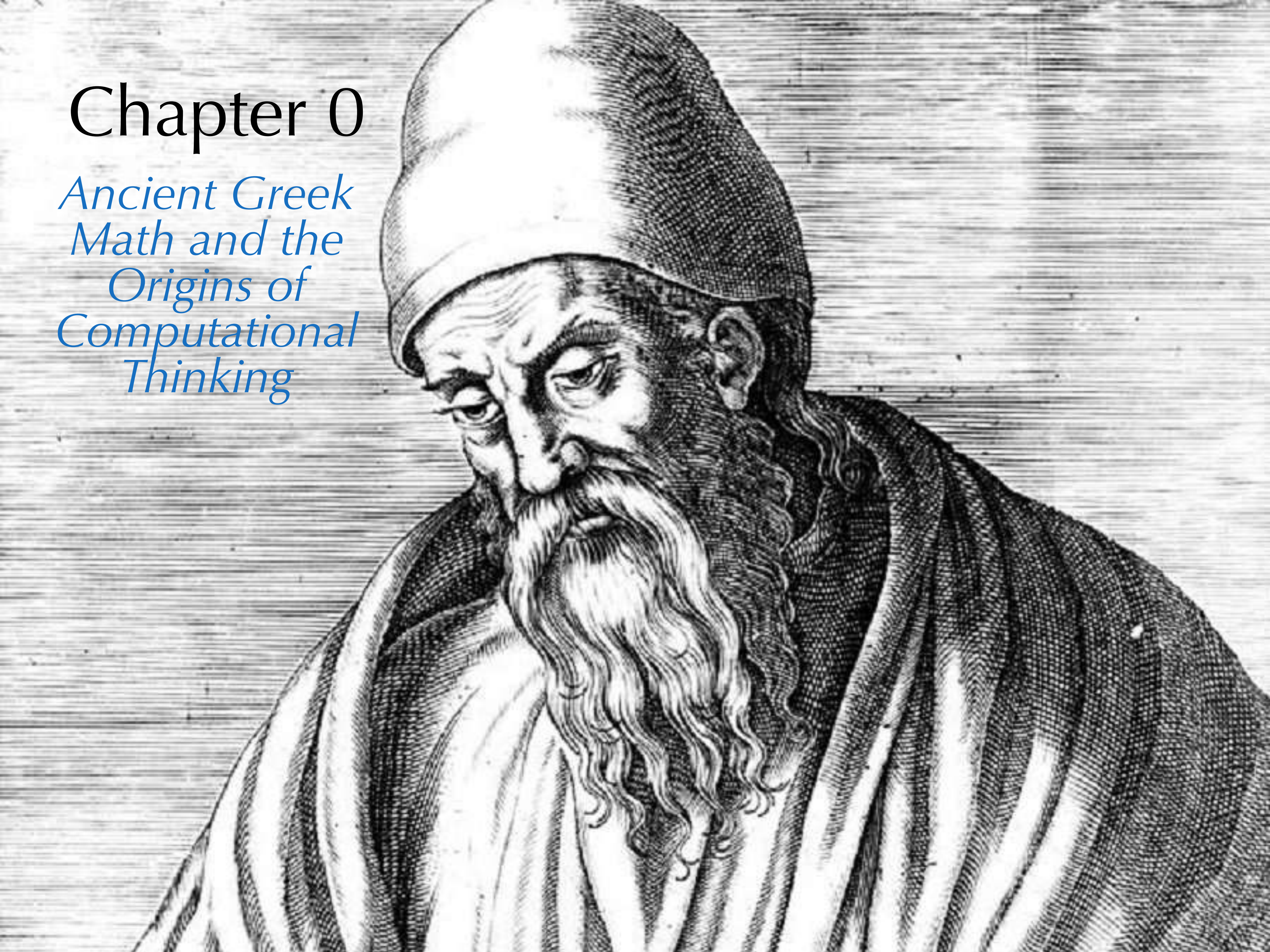
WELCOME



Phillip Compeau, Ph.D.
Asst. Dept. Head for Education
Computational Biology Department
School of Computer Science
Carnegie Mellon University

Chapter 0

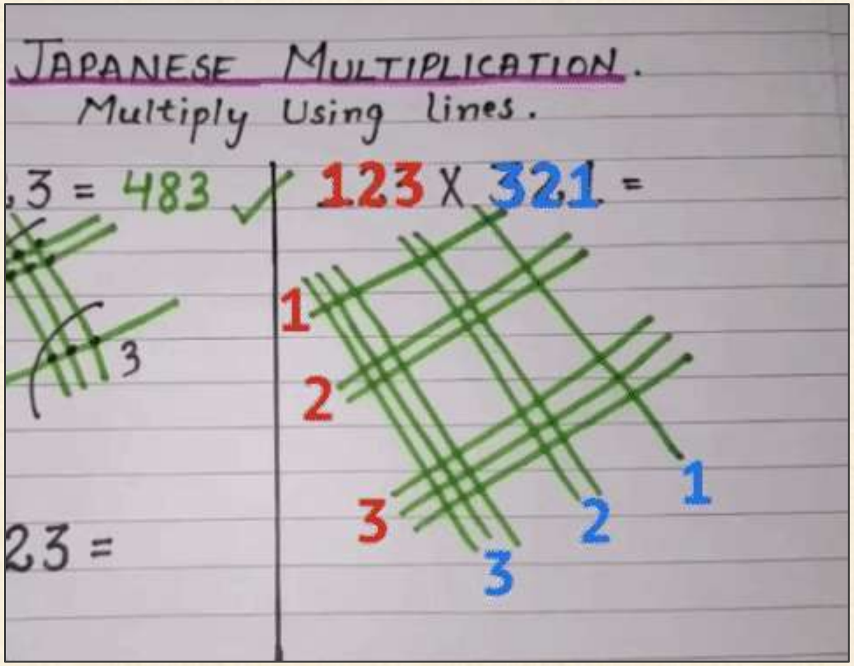
Ancient Greek Math and the Origins of Computational Thinking



OUR FIRST COMPUTATIONAL PROBLEM

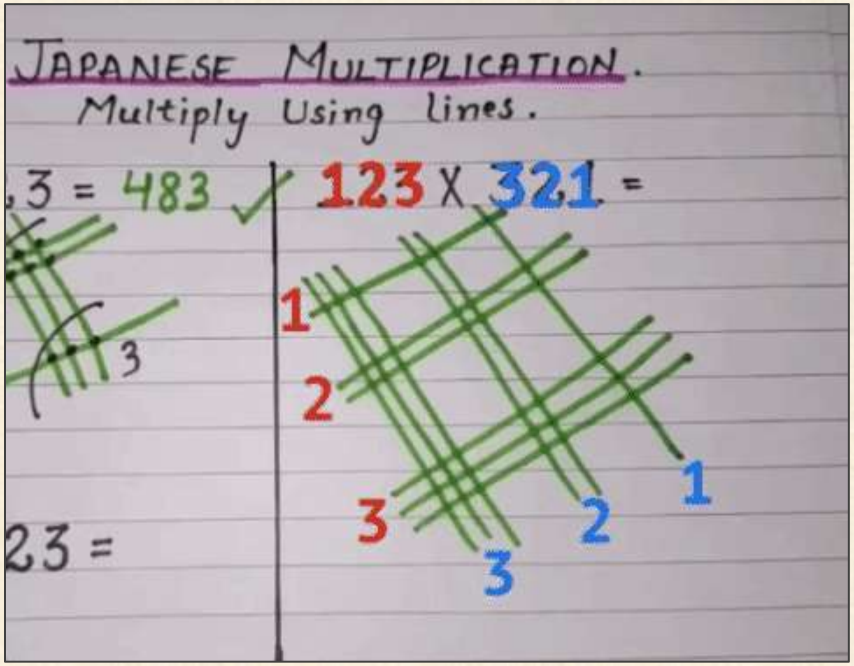
Algorithms are Everywhere

Algorithm: a sequence of steps used to solve a problem.



Algorithms are Everywhere

Programming: converting an algorithm into code.



Our First Computational Problem

Computational problem: *input* data along with a specified *output* involving the input data that can be interpreted in *only one way*.

Our First Computational Problem

Computational problem: *input* data along with a specified *output* involving the input data that can be interpreted in *only one way*.

GCD Problem

- **Input:** Integers a and b.
- **Output:** The greatest common divisor of a and b, denoted GCD(a, b).

Our First Computational Problem

Computational problem: *input* data along with a specified *output* involving the input data that can be interpreted in *only one way*.

GCD Problem

- **Input:** Integers a and b.
- **Output:** The greatest common divisor of a and b, denoted $\mathrm{GCD}(a, b)$.

a and b are called **variables**; they can *change* depending on what values we want them to have.

Our First Computational Problem

Computational problem: *input* data along with a specified *output* involving the input data that can be interpreted in *only one way*.

GCD Problem

- **Input:** Integers x and y.
- **Output:** The greatest common divisor of x and y, denoted GCD(x, y).

STOP: Does this substitution change the computational problem?

Trivial Algorithm for Computing a GCD

Divisors of 378	1	2	3	6	7	9	14	18	21	27	42	54	63	126	189	378
Divisors of 273	1		3		7		13		21		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.

Trivial Algorithm for Computing a GCD

Divisors of 378	**1**	2	**3**	6	**7**	9	14	18	**21**	27	42	54	63	126	189	378
Divisors of 273	**1**		**3**		**7**		13		**21**		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a, b*):

Trivial Algorithm for Computing a GCD

Divisors of 378	1	2	3	6	7	9	14	18	21	27	42	54	63	126	189	378
Divisors of 273	1		3		7		13		21		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a*, *b*):
   - Is *n* a divisor of *a*?

Trivial Algorithm for Computing a GCD

Divisors of 378	1	2	3	6	7	9	14	18	21	27	42	54	63	126	189	378
Divisors of 273	1		3		7		13		21		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a*, *b*):
   - Is *n* a divisor of *a*?
   - Is *n* a divisor of *b*?

Trivial Algorithm for Computing a GCD

Divisors of 378	1	2	3	6	7	9	14	18	21	27	42	54	63	126	189	378
Divisors of 273	1		3		7		13		21		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a, b*):
   - Is *n* a divisor of *a*?
   - Is *n* a divisor of *b*?
   - If the answer to both of these questions is "Yes", update our largest common divisor found to be equal to *n*.

Trivial Algorithm for Computing a GCD

Divisors of 378	1	2	3	6	7	9	14	18	21	27	42	54	63	126	189	378
Divisors of 273	1		3		7		13		21		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a*, *b*):
   - Is *n* a divisor of *a*?
   - Is *n* a divisor of *b*?
   - If the answer to both of these questions is "Yes", update our largest common divisor found to be equal to *n*.
3. After ranging through all these integers, the largest common divisor found must be GCD(*a*, *b*).

Trivial Algorithm for Computing a GCD

Divisors of 378	1	2	3	6	7	9	14	18	21	27	42	54	63	126	189	378
Divisors of 273	1		3		7		13		21		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a*, *b*):
   - Is *n* a divisor of *a*?
   - Is *n* a divisor of *b*?
   - If the answer to both of these questions is "Yes", update our largest common divisor found to be equal to *n*.
3. After ranging through all these integers, the largest common divisor found must be GCD(*a*, *b*).

STOP: Why might we want a faster approach?

A PAINLESS INTRO TO PSEUDOCODE/CONTROL FLOW

Programming Languages are Plentiful

Pseudocode and the Three Bears

Pseudocode: A way of describing algorithms by emphasizing ideas that is “just right”…

Not too vague, like human language

Not too precise, like a specific programming language

Illustrating Pseudocode with a Simple Problem

Minimum of Two Numbers Problem
- **Input:** Numbers *a* and *b*.
- **Output:** The minimum value of *a* and *b*.

Illustrating Pseudocode with a Simple Problem

Minimum of Two Numbers Problem

- **Input:** Numbers *a* and *b*.
- **Output:** The minimum value of *a* and *b*.

Seminal idea in computer science: being able to **branch** based on testing a condition.

Algorithms are Just Like Functions

Computer Science

Input → Algorithm → **Output**

Math

Input → Function → **Output**

Our First Function

```
Min2(a, b)
    if a > b
        return b
    else
        return a
```

Our First Function

```
Min2(a, b)
    if a > b
        return b
    else
        return a
```

Min2: **name** of the function.

Our First Function

```
Min2(a, b)
    if a > b
        return b
    else
        return a
```

a, b: input “**argument**”/”**parameter**” variables.

Our First Function

```
Min2(a, b)
    if a > b
        return b
    else
        return a
```

if $a > b$: **if statement** (allows us to branch)

Our First Function

```
Min2(a, b)
    if a > b
        return b
    else
        return a
```

If the "if statement" is true, we enter the **if block**.

return *b*: **return statement** (provides output).

Our First Function

```
Min2(a, b)
        if a > b
                return b
        else
                return a
```

If the "if statement" is false, we *skip* the if block and enter the **else block**.

else: indicates where to go when if statement is false.

Our First Function

```
Min2(a, b)
    if a > b
        return b
    else
        return a
```

STOP: Does **Min2** still return the desired answer if *a* and *b* are equal?

General Form of If Statements

Control flow: The sequence of steps that a computer takes when executing a program.

SomeFunction(*parameters*)
    execute instructions *A*
    if condition *X* is **true**
        execute instructions *Y*
    else
        execute instructions *Z*
    execute instructions *B*

General Form of If Statements

"if", "else", "return", "true", etc. are **keywords:** words with *reserved meanings* in most languages.

SomeFunction(*parameters*)
    execute instructions *A*
    if condition *X* is **true**
        execute instructions *Y*
    else
        execute instructions *Z*
    execute instructions *B*

General Form of If Statements

STOP: Will *A* always be executed? Will *B* always be executed?

SomeFunction(*parameters*)
  execute instructions *A*
  if condition *X* is **true**
    execute instructions *Y*
  else
    execute instructions *Z*
  execute instructions *B*

It's Your Turn …

Minimum of Three Numbers Problem

- **Input:** Numbers *a, b,* and *c*.
- **Output:** The minimum value of *a, b,* and *c*.

Exercise: Write a (pseudocode) function **Min3** that solves this problem.

Control Flow Can Get Tricky Quickly

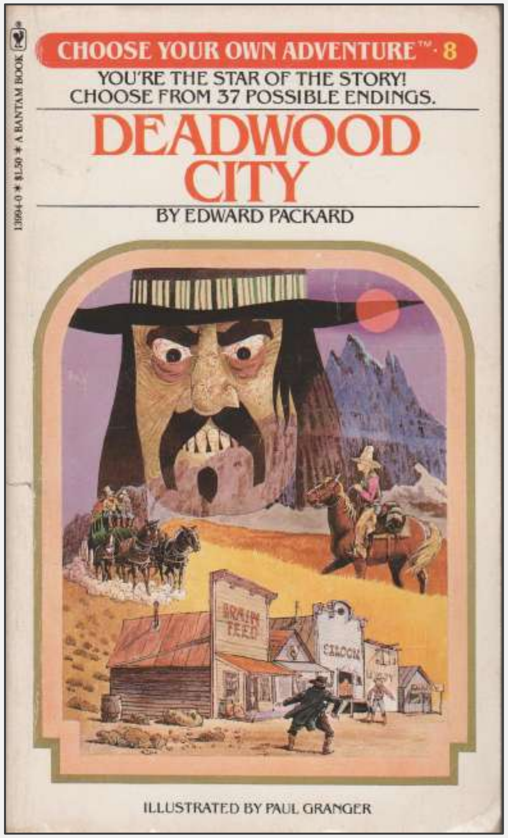
```
Min3(a, b, c)
    if a > b
        if b > c
            return c
        else
            return b
```

Control Flow Can Get Tricky Quickly

```
Min3(a, b, c)
    if a > b
        if b > c
            return c
        else
            return b
    else
        if a > c
            return c
        else
            return a
```

Control Flow Can Get Tricky Quickly

```
Min3(a, b, c)
    if a > b
        if b > c
            return c
        else
            return b
    else
        if a > c
            return c
        else
            return a
```



The colored lines represent a **nested** if statement.

If Statements

```
Min3(a, b, c)
    if a > b
        if b > c
            return c
        else
            return b
    else
        if a > c
            return c
        else
            return a
```

STOP: Where have we seen the colored code?

If Statements

Min3(a, b, c)

```
Min3(a, b, c)
    if a > b
        if b > c
            return c
        else                    Min2(b, c)
            return b
    else
        if a > c
            return c
        else                    Min2(a, c)
            return a
```

STOP: Where have we seen the colored code?

If Statements

```
Min3(a, b, c)
        if a > b
                return Min2(b, c)
        else
                return Min2(a, c)
```

If Statements

```
Min3(a, b, c)
    if a > b
        return Min2(b, c)
    else
        return Min2(a, c)
```

Subroutine: a function used within another function.

If Statements

```
Min3(a, b, c)
    if a > b
        return Min2(b, c)
    else
        return Min2(a, c)
```

Subroutine: a function used within another function.

Exercise: Write pseudocode for a function **Min4**(*a*, *b*, *c*, *d*) that computes the minimum of four numbers.

Multiple Approaches Exist for Solving Even a Simple Problem

```
Min4(a, b, c, d)
    if a > b
        return Min3(a, c, d)
    else
        return Min3(b, c, d)
```

Multiple Approaches Exist for Solving Even a Simple Problem

```
Min4(a, b, c, d)
    if a > b
        return Min3(a, c, d)
    else
        return Min3(b, c, d)
```

```
Min4(a, b, c, d)
    return Min2(Min2(a, b), Min2(c, d))
```

Multiple Approaches Exist for Solving Even a Simple Problem

```
Min4(a, b, c, d)
    if a > b
        return Min3(a, c, d)
    else
        return Min3(b, c, d)
```

```
Min4(a, b, c, d)
    return Min2(Min2(a, b), Min2(c, d))
```

STOP: Which of these do you prefer?

Party Trick: Knowing Day of the Week of Your Birthday

The Doomsday Algorithm

These **doomsdays** occur on *Thursdays* in 2019:

– 1/3
– 2/28
– 3/0
– 4/4
– 5/9
– 6/6
– 7/11
– 8/8
– 9/5
– 10/10
– 11/7
– 12/12

STOP: How can we use this information to quickly find the day of the week for any given date in 2019?

The Doomsday Algorithm

```
Doomsday(day, month)
    if month = 1
        if day = 3, 10, 17, 24, or 31
            return "Thursday"
        else
            if day = 4, 11, 18, or 25  this is ugly!
                return "Friday"
        etc.
    else
        if month = 2
            if day = 7, 14, 21, or 28
                return "Thursday"
            else
                if day = 1, 8, 15, or 22  this is ugly!
                    return "Friday"
            etc.
        else
            if month = 3 this is ugly!
            etc.
```

The "Else" Statements Aren't Needed…

```
Doomsday(day, month)
    if month = 1
        if day = 3, 10, 17, 24, or 31
            return "Thursday"
        if day = 4, 11, 18, or 25
            return "Friday"
        etc.
    if month = 2
        if day = 7, 14, 21, or 28
            return "Thursday"
        if day = 1, 8, 15, or 22
            return "Friday"
        etc.
    if month = 3
        etc.
```

Introducing "Else If"

```
Doomsday(day, month)
    if month = 1
        if day = 3, 10, 17, 24, or 31
            return "Thursday"
        else if day = 4, 11, 18, or 25
            return "Friday"
        etc.
    else if month = 2
        if day = 7, 14, 21, or 28
            return "Thursday"
        else if day = 1, 8, 15, or 22
            return "Friday"
        etc.
    else if month = 3
        etc.
```

LOOPS AND THE TRIVIAL GCD ALGORITHM

How Do We Convert this to Pseudocode?

Divisors of 378	**1**	2	**3**	6	**7**	9	14	18	**21**	27	42	54	63	126	189	378
Divisors of 273	**1**		**3**		**7**		13		**21**		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a, b*):
   - Is *n* a divisor of *a*?
   - Is *n* a divisor of *b*?
   - If the answer to both of these questions is "Yes", update our largest common divisor found to be equal to *n*.
3. After ranging through all these integers, the largest common divisor found must be GCD(*a*, *b*).

How Do We Convert this to Pseudocode?

Divisors of 378	1	2	3	6	7	9	14	18	21	27	42	54	63	126	189	378
Divisors of 273	1		3		7		13		21		39			91		273

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. **For every integer *n* between 1 and min(*a*, *b*):**
   - Is *n* a divisor of *a*?
   - Is *n* a divisor of *b*?
   - If the answer to both of these questions is "Yes", update our largest common divisor found to be equal to *n*.
3. After ranging through all these integers, the largest common divisor found must be GCD(*a*, *b*).

Key Point: how can we do something "for every integer" in a range?

A Simpler Problem: Factorial

Factorial Problem

- **Input:** An integer n.
- **Output:** $n! = n * (n - 1) * (n - 2) * \ldots * 2 * 1$.

A Simpler Problem: Factorial

Factorial Problem

- **Input:** An integer n.
- **Output:** $n! = n * (n - 1) * (n - 2) * \ldots * 2 * 1$.

Factorial(n)
    $p \leftarrow 1$
    $i \leftarrow 1$
    while $i \leq n$
        $p \leftarrow p \cdot i$
        $i \leftarrow i + 1$
    return p

A Simpler Problem: Factorial

$p \leftarrow 1$: **declaring** an intermediate variable p equal to 1 (p will eventually hold the factorial product)

Factorial(n)
    $p \leftarrow 1$
    $i \leftarrow 1$
    while $i \leq n$
        $p \leftarrow p \cdot i$
        $i \leftarrow i + 1$
    return p

A Simpler Problem: Factorial

$p \leftarrow 1$: the *variable* on the left of $\leftarrow$ receives the *value* of the right side.

Factorial(n)
    $p \leftarrow 1$
    $i \leftarrow 1$
    while $i \leq n$
        $p \leftarrow p \cdot i$
        $i \leftarrow i + 1$
    return p

A Simpler Problem: Factorial

$i \leftarrow 1$: i will allow us to "range" over all integers up to n.

Factorial(n)

$p \leftarrow 1$
$i \leftarrow 1$
while $i \le n$
  $p \leftarrow p \cdot i$
  $i \leftarrow i + 1$
return p

A Simpler Problem: Factorial

while $i \leq n$: example of a **while loop**. Just like an if statement – if $i \leq n$ is true, we enter **while block**.

Factorial(n)
    $p \leftarrow 1$
    $i \leftarrow 1$
    while $i \leq n$
        $p \leftarrow p \cdot i$ ($p = 1$)
        $i \leftarrow i + 1$ ($i = 2$)
    return p

A Simpler Problem: Factorial

The difference: after the **while block**, we test $i \leq n$ *again* and (if true) enter the while block *again*.

Factorial(n)
    $p \leftarrow 1$
    $i \leftarrow 1$
    while $i \leq n$
        $p \leftarrow p \cdot i$ ($p = 2$)
        $i \leftarrow i + 1$ ($i = 3$)
    return p

A Simpler Problem: Factorial

Factorial(n)
$p \leftarrow 1$
$i \leftarrow 1$
while $i \leq n$
$p \leftarrow p \cdot i$
$i \leftarrow i + 1$
return p

A Simpler Problem: Factorial

n	p	i	Is $i \leq n$?	Updated value of p	Updated value of i
4	1	1	Yes	$1 \cdot 1 = 1$	$1 + 1 = 2$

```
Factorial(n)
    p ← 1
    i ← 1
    while i ≤ n
        p ← p · i
        i ← i + 1
    return p
```

A Simpler Problem: Factorial

n	p	i	Is $i \leq n$?	Updated value of p	Updated value of i
4	1	1	Yes	$1 \cdot 1 = 1$	$1 + 1 = 2$
4	1	2	Yes	$1 \cdot 2 = 2$	$2 + 1 = 3$

Factorial(n)
$p \leftarrow 1$
$i \leftarrow 1$
while $i \leq n$
$p \leftarrow p \cdot i$
$i \leftarrow i + 1$
return p

A Simpler Problem: Factorial

n	p	i	Is $i \leq n$?	Updated value of p	Updated value of i
4	1	1	Yes	$1 \cdot 1 = 1$	$1 + 1 = 2$
4	1	2	Yes	$1 \cdot 2 = 2$	$2 + 1 = 3$
4	2	3	Yes	$2 \cdot 3 = 6$	$3 + 1 = 4$

Factorial(n)
  $p \leftarrow 1$
  $i \leftarrow 1$
  while $i \leq n$
    $p \leftarrow p \cdot i$
    $i \leftarrow i + 1$
  return p

A Simpler Problem: Factorial

n	p	i	Is $i \leq n$?	Updated value of p	Updated value of i
4	1	1	Yes	$1 \cdot 1 = 1$	$1 + 1 = 2$
4	1	2	Yes	$1 \cdot 2 = 2$	$2 + 1 = 3$
4	2	3	Yes	$2 \cdot 3 = 6$	$3 + 1 = 4$
4	6	4	Yes	$6 \cdot 4 = 24$	$4 + 1 = 5$

```
Factorial(n)
    p ← 1
    i ← 1
    while i ≤ n
        p ← p · i
        i ← i + 1
    return p
```

A Simpler Problem: Factorial

n	p	i	Is $i \leq n$?	Updated value of p	Updated value of i
4	1	1	Yes	$1 \cdot 1 = 1$	$1 + 1 = 2$
4	1	2	Yes	$1 \cdot 2 = 2$	$2 + 1 = 3$
4	2	3	Yes	$2 \cdot 3 = 6$	$3 + 1 = 4$
4	6	4	Yes	$6 \cdot 4 = 24$	$4 + 1 = 5$
4	24	5	No		

Factorial(n)
    $p \leftarrow 1$
    $i \leftarrow 1$
    while $i \leq n$
        $p \leftarrow p \cdot i$
        $i \leftarrow i + 1$
    return p

A Simpler Problem: Factorial

STOP: What happens if we remove $i \leftarrow i + 1$?

Factorial(n)
    $p \leftarrow 1$
    $i \leftarrow 1$
    while $i \leq n$
        $p \leftarrow p \cdot i$
        $i \leftarrow i + 1$
    return p

A Simpler Problem: Factorial

STOP: What happens if we remove $i \leftarrow i + 1$?

Infinite loop: a loop that never terminates.

```
Factorial(n)
    p ← 1
    i ← 1
    while i ≤ n
        p ← p · i
    return p
```

For Loops Simplify Ranging

For loop: a way of simplifying the process of "ranging" through a collection of values.

AnotherFactorial(n)
    $p \leftarrow 1$
    for every integer i from 1 to n
        $p \leftarrow p \cdot i$
    return p

Note: While Loops are More General

```
PittsburghFebruary()
    while temperature is below freezing
        daydream about moving south
```

Returning to the Trivial GCD

A **trivial** (obvious) algorithm solving the GCD problem.

1. Start our largest common divisor at 1.
2. For every integer *n* between 1 and min(*a*, *b*):
   - Is *n* a divisor of *a*?
   - Is *n* a divisor of *b*?
   - If the answer to both of these questions is "Yes", update our largest common divisor found to be equal to *n*.
3. After ranging through all these integers, the largest common divisor found must be GCD(*a*, *b*).

Exercise: Write pseudocode for a function **TrivialGCD**(*a*, *b*) representing this algorithm (assume any subroutines you like).

Returning to the Trivial GCD

```
TrivialGCD(a, b)
    d ← 1
    m ← Min2(a, b)  (subroutine!)
    for every integer p from 1 to m
        if p is a divisor of both a and b
            d ← p
    return d
```

Returning to the Trivial GCD

TrivialGCD(*a*, *b*)
    d ← 1
    m ← **Min2**(*a*, *b*) (subroutine!)
    for every integer *p* from 1 to *m*
        if *p* is a divisor of both *a* and *b*
            d ← *p*
    return *d*

We should discuss how a computer determines if one number is a divisor of another…

Integer Division

The **integer division** of *x*/*y* is defined by taking the integer part of the division and "throwing away" the remainder.

14/3 = ?　　102/12 = ?　　11/2 = ?

Integer Division

The **integer division** of *x*/*y* is defined by taking the integer part of the division and "throwing away" the remainder.

$14/3 = 4 \qquad 102/12 = 8 \qquad 11/2 = 5$

Integer Division

The **integer division** of *x*/*y* is defined by taking the integer part of the division and "throwing away" the remainder.

14/3 = 4 102/12 = 8 11/2 = 5

STOP: How does *p* being a divisor of *n* relate to integer division and remainder?

Integer Division

The **integer division** of *x*/*y* is defined by taking the integer part of the division and "throwing away" the remainder.

14/3 = 4   102/12 = 8   11/2 = 5

Exercise: Write pseudocode for functions **IntegerDivision**(*n*, *p*) and **Remainder**(*n*, *p*) corresponding to the integer division and remainder formed by *n*/*p*. Your only allowable arithmetic operations are addition, subtraction, and multiplication.

Integer Division is "Repeated Subtraction"

```
IntegerDivision(n, p)
    c ← 0
    n ← n – p
    while n ≥ 0
        c ← c + 1
        n ← n – p
    return c
```

Note: We can check the correctness of our function by *testing* it on various outputs.

Remainder() Uses IntegerDivision() as a Subroutine

```
Remainder(n, p)
    return n – p * IntegerDivision(n, p)
```

Remainder(14,3) = 14 – 3*IntegerDivision(14,3) = 2

Remainder() Uses IntegerDivision() as a Subroutine

```
Remainder(n, p)
        return n – p * IntegerDivision(n, p)
```

Remainder(14,3) = 14 – 3*IntegerDivision(14,3) = 2

Remainder(102,12) = 102 – 12*IntegerDivision(102,12) = 6

Remainder() Uses IntegerDivision() as a Subroutine

```
Remainder(n, p)
        return n – p * IntegerDivision(n, p)
```

Remainder(14,3) = 14 – 3*IntegerDivision(14,3) = 2

Remainder(102,12) = 102 – 12*IntegerDivision(102,12) = 6

Remainder(11,2) = 11 – 2*IntegerDivision(11,2) = 1

Remainder() and Doomsday

```
Doomsday(day, month)
        if month = 1
                if day = 3, 10, 17, 24, or 31
                        return “Friday”
                else if day = 4, 11, 18, or 25
                        return “Saturday”
                etc.
        else if month = 2
                if day = 7, 14, 21, or 28
                        return “Monday”
                else if day = 1, 8, 15, or 22
                        return “Tuesday”
                etc.
        else if month = 3
                etc.
```

STOP: How would **Remainder** be helpful here?

TrivialGCD is Now Good to Go

```
TrivialGCD(a, b)
    d ← 1
    m ← Min2(a, b)  (subroutine!)
    for every integer p from 1 to m
        if p is a divisor of both a and b
            d ← p
    return d
```

TrivialGCD is Now Good to Go

TrivialGCD(*a*, *b*)
 $d \leftarrow 1$
 $m \leftarrow$ **Min2**(*a*, *b*) (subroutine!)
 for every integer *p* from 1 to *m*
  if Remainder(*a*, *p*) = 0 **and Remainder**(*b*, *p*) = 0
   $d \leftarrow p$
 return *d*

Note: The word "**and**" is a keyword too. More later...

EUCLID'S INSIGHT AND THE WORLD'S FIRST NONTRIVIAL ALGORITHM

Euclid's Theorem

Euclid's Theorem: If $a > b$, then

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a-b, b).$$

Euclid's Theorem

Euclid's Theorem: If $a > b$, then

$$\text{GCD}(a, b) = \text{GCD}(a-b, b).$$

Two Pressing Questions:

1. How can we demonstrate this for any possible pair of integers?
2. Why do we really *care* that this is true computationally?

Euclid's Theorem

Euclid's Theorem: If $a > b$, then

$$\text{GCD}(a, b) = \text{GCD}(a-b, b).$$

Two Pressing Questions:

1. **How can we demonstrate this for any possible pair of integers? – *Let's prove it!***
2. Why do we really *care* that this is true computationally?

Proof of Euclid's Theorem

Euclid's Theorem: If $a > b$, then

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a-b, b).$$

Common problem-solving technique in mathematics: sometimes, we can prove a *more general* statement.

Proof of Euclid's Theorem

Euclid's Theorem: If $a > b$, then

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a-b, b).$$

Common problem-solving technique in mathematics: sometimes, we can prove a *more general* statement.

More general statement: If $a > b$, then *all* shared divisors of a and b is the same as *all* shared divisors of $a - b$ and b.

Proof of Euclid's Theorem

We prove the general statement with two facts:

1. Any shared divisor of a and b must also be a divisor of $a - b$.
2. Any shared divisor of $a - b$ and b must also be a divisor of a.

More general statement: If $a > b$, then *all* shared divisors of a and b is the same as *all* shared divisors of $a - b$ and b.

Proof of Euclid's Theorem

We prove the general statement with two facts:

1. **Any shared divisor of *a* and *b* must also be a divisor of *a* – *b*.**
2. Any shared divisor of $a - b$ and b must also be a divisor of a.

Say d is a divisor of a and b. There must be some integers x and y such that

$$dx = a, dy = b$$

Proof of Euclid's Theorem

We prove the general statement with two facts:

1. **Any shared divisor of *a* and *b* must also be a divisor of *a* – *b*.**
2. Any shared divisor of $a - b$ and b must also be a divisor of a.

Say d is a divisor of a and b. There must be some integers x and y such that

$$dx = a, dy = b$$

$$a - b = dx - dy = d(x - y)$$

Proof of Euclid's Theorem

We prove the general statement with two facts:

1. **Any shared divisor of *a* and *b* must also be a divisor of *a* – *b*.**
2. Any shared divisor of $a - b$ and b must also be a divisor of a.

Say d is a divisor of a and b. There must be some integers x and y such that

$$dx = a, dy = b$$

$$a - b = dx - dy = d(x - y)$$

So d is a divisor of $a - b$ as well.

Proof of Euclid's Theorem

We prove the general statement with two facts:

1. Any shared divisor of a and b must also be a divisor of $a - b$.
2. **Any shared divisor of $a - b$ and b must also be a divisor of a.**

Say e is a divisor of $a - b$ and b. There must be some integers p and q such that

$$ep = a - b, \; eq = b$$

Proof of Euclid's Theorem

We prove the general statement with two facts:

1. Any shared divisor of a and b must also be a divisor of $a - b$.
2. **Any shared divisor of $a - b$ and b must also be a divisor of a.**

Say e is a divisor of $a - b$ and b. There must be some integers p and q such that

$$ep = a - b,\ eq = b$$

$$a = (a - b) + b = ep + eq = e(p + q)$$

Proof of Euclid's Theorem

We prove the general statement with two facts:

1. Any shared divisor of a and b must also be a divisor of $a - b$.
2. **Any shared divisor of $a - b$ and b must also be a divisor of a.**

Say e is a divisor of $a - b$ and b. There must be some integers p and q such that

$$ep = a - b, \; eq = b$$

$$a = (a - b) + b = ep + eq = e(p + q)$$

So e is a divisor of a as well.

Euclid's Theorem

Euclid's Theorem: If $a > b$, then

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a-b, b).$$

Two Pressing Questions:

1. How can we demonstrate this for any possible pair of integers?
2. **Why do we really *care* that this is true computationally?**

Euclid's Algorithm in Action

Euclid's Theorem: If $a > b$, then

$$\text{GCD}(a, b) = \text{GCD}(a-b, b).$$

$$\text{GCD}(378, 273) = \text{GCD}(105, 273)$$

Euclid's Algorithm in Action

Euclid's Theorem: If $a > b$, then

$$\text{GCD}(a, b) = \text{GCD}(a-b, b).$$

$$\begin{aligned}\text{GCD}(378, 273) &= \text{GCD}(105, 273)\\ &= \text{GCD}(105, 168)\end{aligned}$$

Euclid's Algorithm in Action

Euclid's Theorem: If $a > b$, then

$$\text{GCD}(a, b) = \text{GCD}(a-b, b).$$

$$\begin{aligned} \text{GCD}(378, 273) &= \text{GCD}(105, 273) \\ &= \text{GCD}(105,168) \\ &= \text{GCD}(105,63) \end{aligned}$$

Euclid's Algorithm in Action

Euclid's Theorem: If $a > b$, then

$$\text{GCD}(a, b) = \text{GCD}(a-b, b).$$

$$\begin{aligned} \text{GCD}(378, 273) &= \text{GCD}(105, 273) \\ &= \text{GCD}(105,168) \\ &= \text{GCD}(105,63) \\ &= \text{GCD}(42,63) \end{aligned}$$

Euclid's Algorithm in Action

Euclid's Theorem: If $a > b$, then

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a-b, b).$$

$$\begin{aligned}
\mathrm{GCD}(378, 273) &= \mathrm{GCD}(105, 273) \\
&= \mathrm{GCD}(105,168) \\
&= \mathrm{GCD}(105,63) \\
&= \mathrm{GCD}(42,63) \\
&= \mathrm{GCD}(42,21)
\end{aligned}$$

Euclid's Algorithm in Action

Euclid's Theorem: If $a > b$, then

$$\text{GCD}(a, b) = \text{GCD}(a-b, b).$$

$$\begin{aligned}\text{GCD}(378, 273) &= \text{GCD}(105, 273)\\ &= \text{GCD}(105,168)\\ &= \text{GCD}(105,63)\\ &= \text{GCD}(42,63)\\ &= \text{GCD}(42,21)\\ &= \text{GCD}(21,21)\end{aligned}$$

Euclid's Algorithm in Action

Euclid's Theorem: If $a > b$, then

$$\mathrm{GCD}(a, b) = \mathrm{GCD}(a-b, b).$$

$$\begin{aligned}
\mathrm{GCD}(378, 273) &= \mathrm{GCD}(105, 273)\\
&= \mathrm{GCD}(105,168)\\
&= \mathrm{GCD}(105,63)\\
&= \mathrm{GCD}(42,63)\\
&= \mathrm{GCD}(42,21)\\
&= \mathrm{GCD}(21,21)\\
&= 21
\end{aligned}$$

Grace Hopper on the Value of Speed

Source: https://youtu.be/9eyFDBPk4Yw

Euclid's Algorithm in Action

Exercise: Brainstorm how we could write a function in pseudocode to compute the GCD of two numbers by repeatedly applying Euclid's Theorem.

$$
\begin{aligned}
\mathrm{GCD}(378, 273) &= \mathrm{GCD}(105, 273) \\
&= \mathrm{GCD}(105,168) \\
&= \mathrm{GCD}(105,63) \\
&= \mathrm{GCD}(42,63) \\
&= \mathrm{GCD}(42,21) \\
&= \mathrm{GCD}(21,21) \\
&= 21
\end{aligned}
$$

Pseudocode for Euclid's Algorithm

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

Pseudocode for Euclid's Algorithm

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

STOP: If we change **return** *a* to **return** *b*, how does it change the algorithm?

Euclid's Algorithm Illustrated

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

a	b	Is $a \neq b$?	Updated value of a	Updated value of b
378	273	Yes	105	273

Euclid's Algorithm Illustrated

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

a	b	Is $a \neq b$?	Updated value of a	Updated value of b
378	273	Yes	105	273
105	273	Yes	105	168

Euclid's Algorithm Illustrated

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

a	b	Is $a \neq b$?	Updated value of a	Updated value of b
378	273	Yes	105	273
105	273	Yes	105	168
105	168	Yes	105	63

Euclid's Algorithm Illustrated

EuclidGCD(a, b)
    while $a \neq b$
        if $a > b$
            $a \leftarrow a - b$
        else
            $b \leftarrow b - a$
    return a

a	b	Is $a \neq b$?	Updated value of a	Updated value of b
378	273	Yes	105	273
105	273	Yes	105	168
105	168	Yes	105	63
105	63	Yes	42	63

Euclid's Algorithm Illustrated

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

a	b	Is $a \neq b$?	Updated value of a	Updated value of b
378	273	Yes	105	273
105	273	Yes	105	168
105	168	Yes	105	63
105	63	Yes	42	63
42	63	Yes	42	21

Euclid's Algorithm Illustrated

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

a	b	Is $a \neq b$?	Updated value of a	Updated value of b
378	273	Yes	105	273
105	273	Yes	105	168
105	168	Yes	105	63
105	63	Yes	42	63
42	63	Yes	42	21
42	21	Yes	21	21

Euclid's Algorithm Illustrated

```
EuclidGCD(a, b)
    while a ≠ b
        if a > b
            a ← a – b
        else
            b ← b – a
    return a
```

a	b	Is $a \neq b$?	Updated value of a	Updated value of b
378	273	Yes	105	273
105	273	Yes	105	168
105	168	Yes	105	63
105	63	Yes	42	63
42	63	Yes	42	21
42	21	Yes	21	21
21	21	No		

ARRAYS AND A FIRST ATTEMPT AT PRIME FINDING

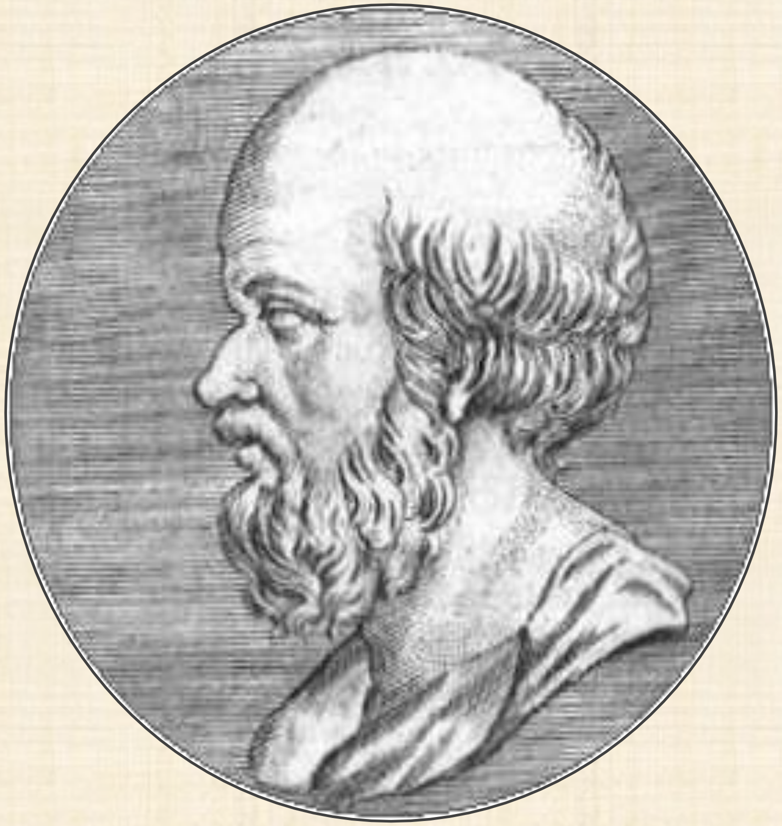
Who First Computed Earth's Circumference?

Source: https://youtu.be/G8cbIWMv0rI

Eratosthenes's Insight

Eratosthenes of Cyrene (276 – 195 BC)

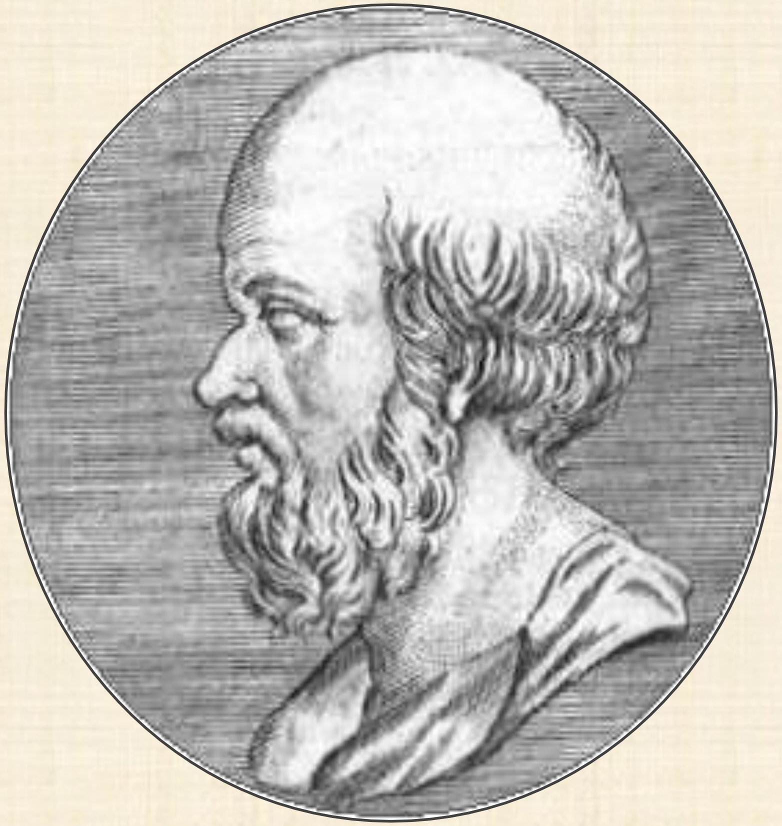
Also: first nontrivial algorithm for identifying prime numbers (soon)



Eratosthenes of Cyrene (276 – 195 BC)

Also: first nontrivial algorithm for identifying prime numbers (soon)

Recall that a positive integer is **prime** if its only divisors are 1 and itself (and **composite** otherwise).



Testing if a Number is Prime

Prime Number Problem

- **Input:** An integer *n*.
- **Output:** "Yes" if *n* is prime, and "No" otherwise.

Testing if a Number is Prime

Prime Number Problem

- **Input:** An integer n.
- **Output:** "Yes" if n is prime, and "No" otherwise.

Decision problem: a computational problem that always returns a "Yes"/"No" answer.

(Decision problems may sound simple, but they lie at the dark heart of computer science.)

Testing if a Number is Prime

Prime Number Problem

- **Input:** An integer *n*.
- **Output:** "Yes" if *n* is prime, and "No" otherwise.

We use the keywords **true** and **false** to represent "Yes" and "No".

A variable taking **true** or **false** is called a **Boolean variable**.

Testing if a Number is Prime

Prime Number Problem

- **Input:** An integer n.
- **Output:** "Yes" if n is prime, and "No" otherwise.

```
IsPrime(n)
    if n = 1
        return false
    for every integer p from 2 to n – 1
        if p is a divisor of n
            return false
    return true
```

Testing if a Number is Prime

STOP: How does this change the algorithm?

IsPrime(n)
    if $n = 1$
        return false
    for every integer p from **1** to $n - 1$
        if p is a divisor of n
            return false
    return true

Running **IsPrime**() on Multiple *n*

p	1	2	3	4	5	6	7	8	9	10	11
Is p prime?	false	true	true	false	true	false	true	false	false	false	true

IsPrime(n)
    if $n = 1$
        return false
    for every integer p from 2 to $n - 1$
        if p is a divisor of n
            return false
    return true

Running **IsPrime**() on Multiple *n*

p	1	2	3	4	5	6	7	8	9	10	11
Is p prime?	false	true	true	false	true	false	true	false	false	false	true

STOP: Do you see any improvements to **IsPrime**()?

```
IsPrime(n)
    if n = 1
        return false
    for every integer p from 2 to n – 1
        if p is a divisor of n
            return false
    return true
```

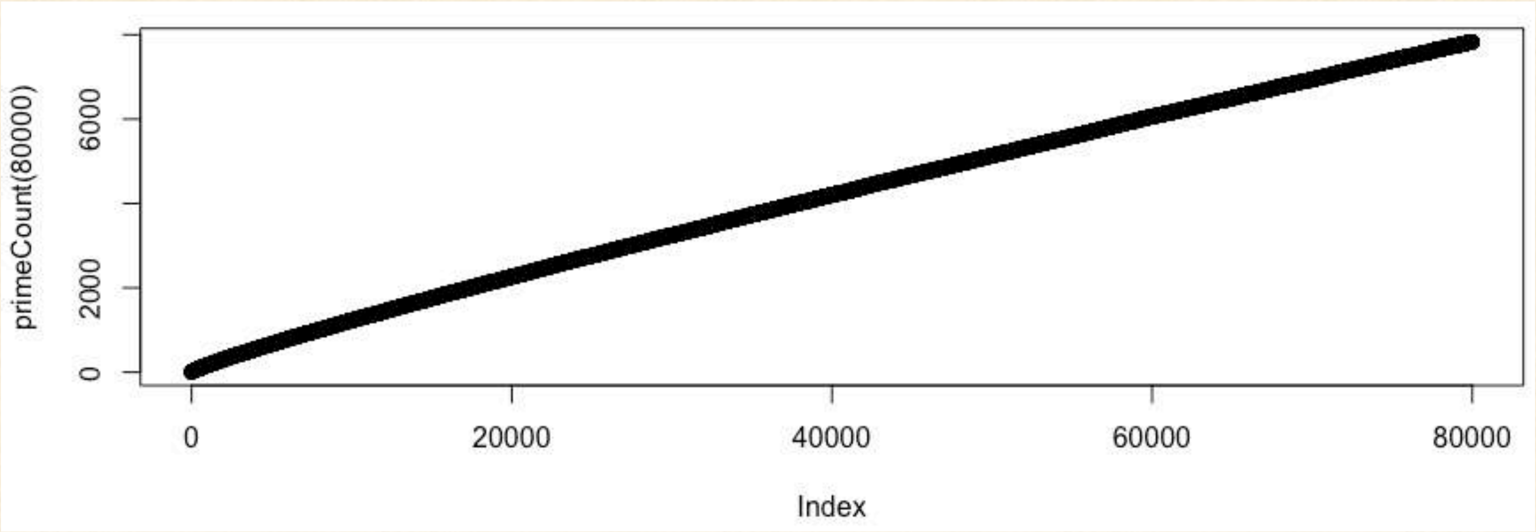
Running **IsPrime**() on Multiple n

p	1	2	3	4	5	6	7	8	9	10	11
Is p prime?	false	true	true	false	true	false	true	false	false	false	true

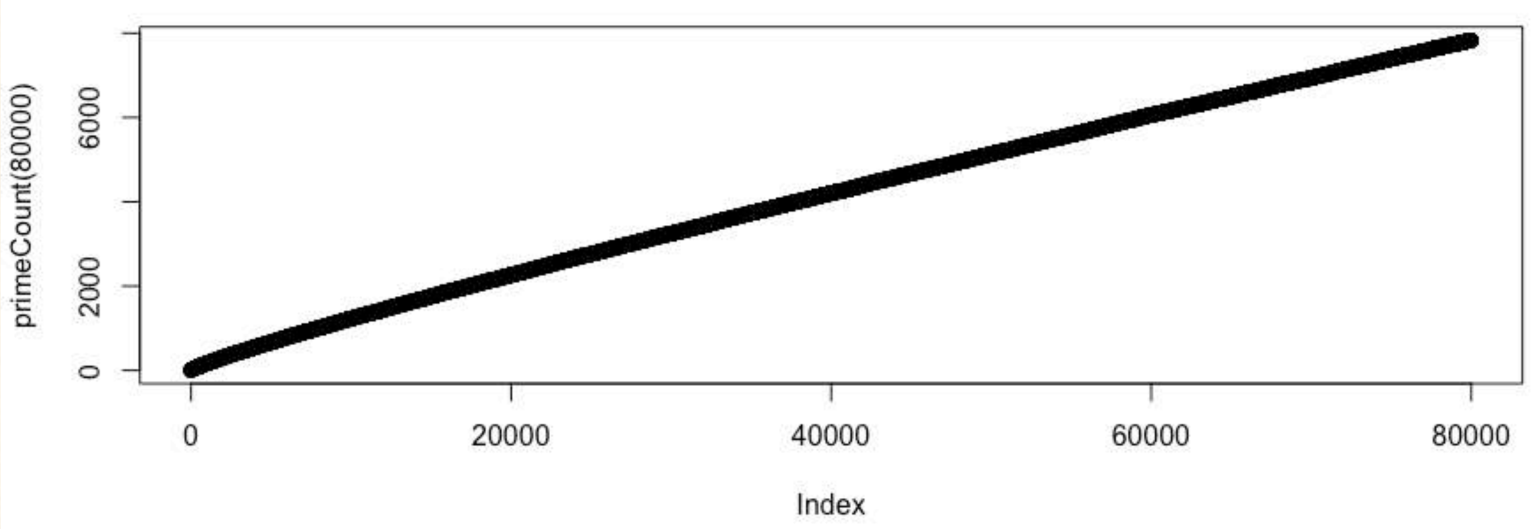
Theorem: If $ab = n$, a or b must be at most $\sqrt{n}$.

IsPrime(n)
  if $n = 1$
    return false
  for every integer p from 2 to $\sqrt{n}$
    if p is a divisor of n
      return false
  return true

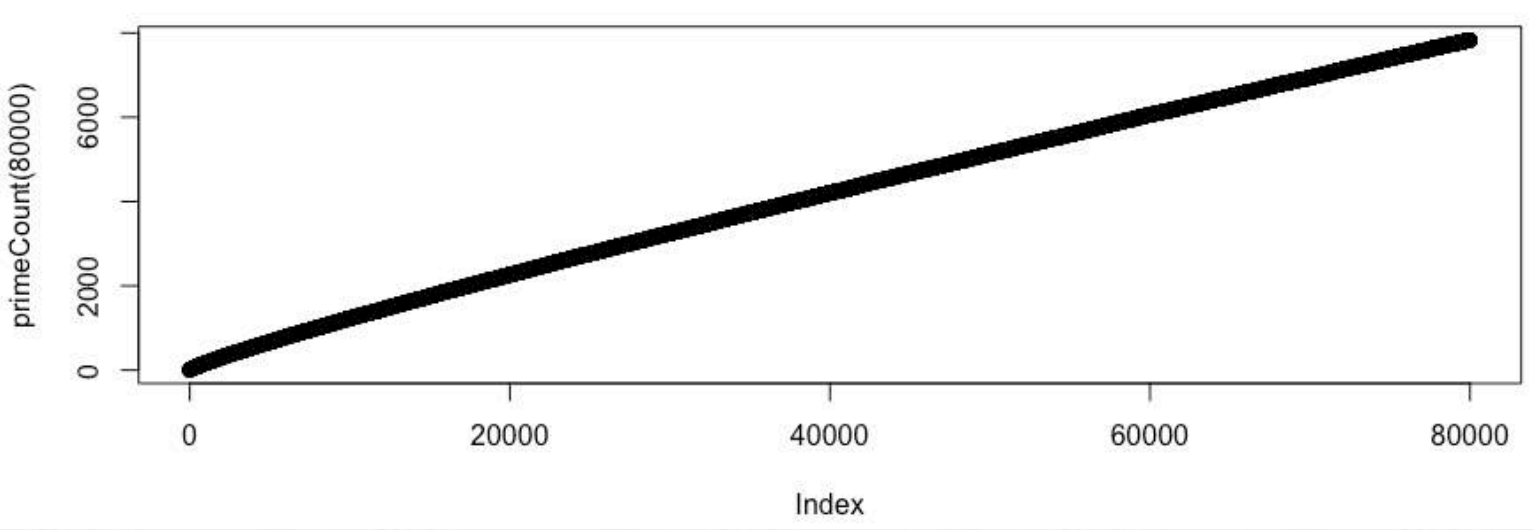
Do the Primes Really Go on Forever?



Do the Primes Really Go on Forever?



Do the Primes Really Go on Forever?



Euclid's Theorem #2: There are infinitely many primes.

First, a Simpler Fact

Simpler Fact: Every composite integer greater than 1 has at least one prime factor.

First, a Simpler Fact

Simpler Fact: Every composite integer greater than 1 has at least one prime factor.

Consider any composite integer *n*; since it is composite, it has factors other than itself and 1.

First, a Simpler Fact

Simpler Fact: Every composite integer greater than 1 has at least one prime factor.

Consider any composite integer *n*; since it is composite, it has factors other than itself and 1.

Take the smallest factor *p* of *n* other than 1. *p* must be prime, since any factor that it would have other than 1 and itself would also be a factor of *n* (but we assumed *p* was the smallest such factor).

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Proof by contradiction: Assume the opposite of what we want to prove, and show that it leads to a **contradiction**, a statement that we know is false.

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Proof by contradiction: Assume the opposite of what we want to prove, and show that it leads to a **contradiction**, a statement that we know is false.

STOP: What is the opposite of what we want to prove in this case?

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Assume that there are finitely many primes. This means that there must be some number n of them, and we can label them $p_1, p_2, \ldots, p_n$.

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Assume that there are finitely many primes. This means that there must be some number n of them, and we can label them $p_1, p_2, \ldots, p_n$.

Consider the number formed by multiplying all these primes together:

$$p = (p_1)\,(p_2)\,\ldots\,(p_n)\;.$$

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

STOP: Is p prime or composite? Why?

Consider the number formed by multiplying all these primes together:

$$p = (p_1)\ (p_2)\ \ldots\ (p_n)\ .$$

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Answer: Composite, because p has many factors other than 1 and itself.

Consider the number formed by multiplying all these primes together:

$$p = (p_1)\ (p_2)\ \ldots\ (p_n)\ .$$

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Now take the number that is 1 larger than p:

$$q = p + 1 = (p_1)\ (p_2)\ \dots\ (p_n) + 1.$$

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Now take the number that is 1 larger than p:

$$q = p + 1 = (p_1)\,(p_2)\,\ldots\,(p_n) + 1.$$

STOP: Is q prime or composite? Why?

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Now take the number that is 1 larger than p:

$$q = p + 1 = (p_1)\,(p_2)\,\ldots\,(p_n) + 1.$$

STOP: Is q prime or composite? Why?

Answer: q must be composite, because it is clearly larger than all known primes!

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Now take the number that is 1 larger than p:

$$q = p + 1 = (p_1)\ (p_2)\ \dots\ (p_n) + 1.$$

Yet look what happens when we divide q by each of the known primes.

$$q\ /\ p_1 = (p_2)\ \dots\ (p_n) + 1/\ p_1\ .$$

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Now take the number that is 1 larger than p:

$$q = p + 1 = (p_1)\ (p_2)\ \ldots\ (p_n) + 1.$$

Yet look what happens when we divide q by each of the known primes.

$$q\ /\ p_1 = (p_2)\ \ldots\ (p_n) + 1/\ p_1\ .$$

$$q\ /\ p_2 = (p_1)\ (p_3)\ \ldots\ (p_n) + 1/\ p_2\ .$$

Proof of Euclid's Theorem #2

Euclid's Theorem #2: There are infinitely many prime numbers.

Now take the number that is 1 larger than p:

$$q = p + 1 = (p_1)\ (p_2)\ \ldots\ (p_n) + 1.$$

Yet look what happens when we divide q by each of the known primes.

$$q\ /\ p_1 = (p_2)\ \ldots\ (p_n) + 1/\ p_1\ .$$

$$q\ /\ p_2 = (p_1)\ (p_3)\ \ldots\ (p_n) + 1/\ p_2\ .$$

The remainder is always 1!

Remember Our Fact …

Fact: Every composite integer greater than 1 has at least one prime factor.

Yet look what happens when we divide q by each of the known primes.

$$q \,/\, p_1 = (p_2) \ldots (p_n) + 1/\, p_1 \,.$$

$$q \,/\, p_2 = (p_1)\,(p_3) \ldots (p_n) + 1/\, p_2 \,.$$

The remainder is always 1!

Remember Our Fact ...

Fact: Every composite integer greater than 1 has at least one prime factor.

q is composite, so it has a prime factor. But none of the primes p_i is a factor. ***Contradiction!***

Yet look what happens when we divide q by each of the known primes.

$$q \,/\, p_1 = (p_2) \ldots (p_n) + 1/\, p_1 \,.$$

$$q \,/\, p_2 = (p_1)\,(p_3) \ldots (p_n) + 1/\, p_2 \,.$$

The remainder is always 1!

The Theorem is Proved ☺

Euclid's Theorem #2: There are infinitely many prime numbers.

q is composite, so it has a prime factor. But none of the primes p_i is a factor. ***Contradiction!***

Yet look what happens when we divide q by each of the known primes.

$$q \,/\, p_1 = (p_2) \ldots (p_n) + 1/\,p_1\,.$$

$$q \,/\, p_2 = (p_1)\,(p_3) \ldots (p_n) + 1/\,p_2\,.$$

The remainder is always 1!

Returning to Factorials

Array: an ordered table/list of variables.

Factorial Array Problem

- **Input:** An integer n.
- **Output:** An array containing all the $n+1$ factorials $0! = 1, 1!, 2!, \ldots, n!$.

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
1	1	2	6	24	120	720

a

Returning to Factorials

Array: an ordered table/list of variables.

Factorial Array Problem

- **Input:** An integer n.
- **Output:** An array containing all the $n+1$ factorials $0! = 1, 1!, 2!, \ldots, n!$.

$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$
1	1	2	6	24	120	720

a

0-based indexing: starting numbering at 0, not 1.

Returning to Factorials

Array: an ordered table/list of variables.

Factorial Array Problem

- **Input:** An integer n.
- **Output:** An array containing all the $n+1$ factorials $0! = 1, 1!, 2!, \ldots, n!$.

FactorialArray(n)

$a \leftarrow$ array of length $n+1$
$a[0] \leftarrow 1$
for every integer k from 1 to n
  $a[k] \leftarrow a[k-1] \cdot k$
return a

Trivial Prime Finding

Prime Number Array Problem

- **Input:** An integer n.
- **Output:** An array *primes* of length $n + 1$ such that for every nonnegative integer $p \leq n$, $primes[p]$ is **true** if p is prime and **false** otherwise.

0	1	2	3	4	5	6	7	8	9
false	false	true	true	false	true	false	true	false	false

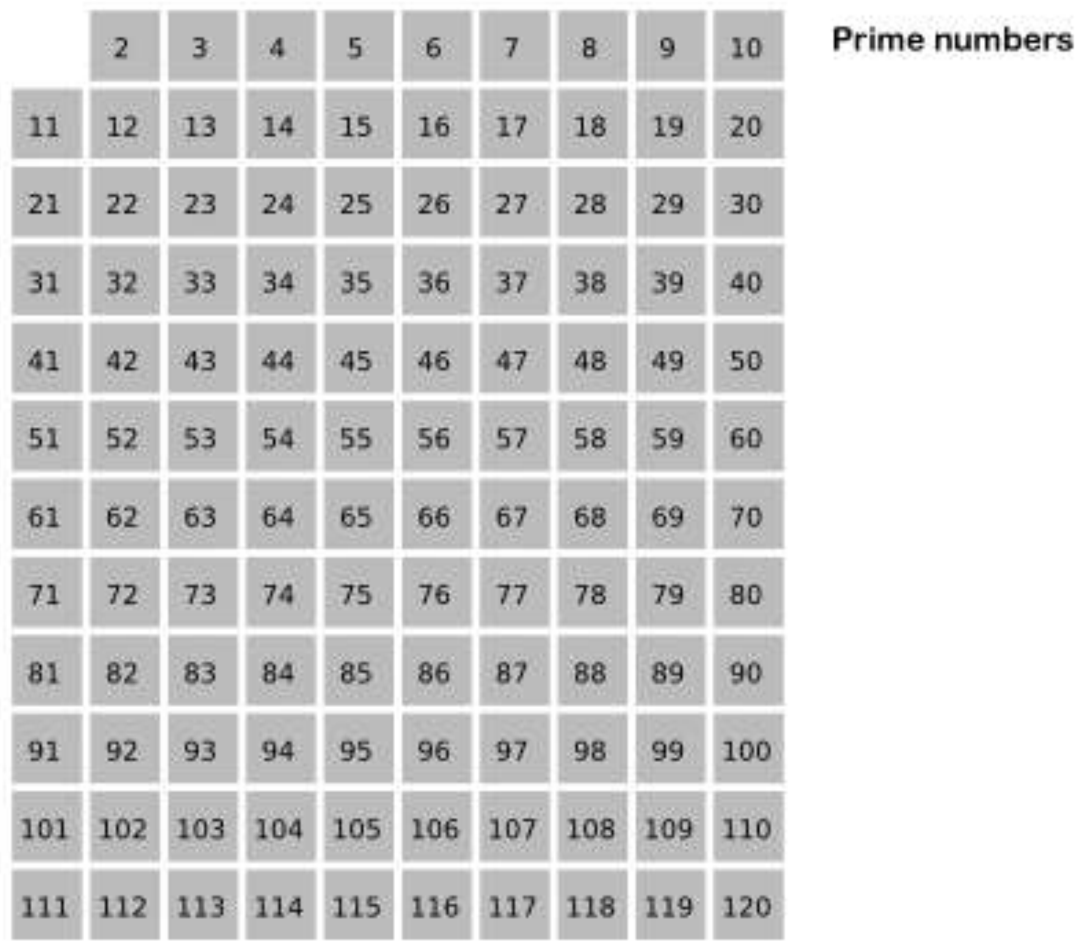
Trivial Prime Finding

```
TrivialPrimeFinder(n)
    primes ← array of n + 1 false boolean variables
    for every integer p from 2 to n
        if IsPrime(p) = true
            primes[p] ← true
  return primes
```

0	1	2	3	4	5	6	7	8	9
false	false	true	true	false	true	false	true	false	false

THE WORLD'S SECOND NONTRIVIAL ALGORITHM

Eratosthenes's Nontrivial Algorithm



https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes#/media/File:Sieve_of_Eratosthenes_animation.gif

The Sieve of Eratosthenes

Prime Number Array Problem

- **Input:** An integer n.
- **Output:** An array *primes* of length $n + 1$ such that for every nonnegative integer $p \leq n$, *primes*[p] is **true** if p is prime and **false** otherwise.

Exercise: Write a pseudocode function **SieveOfEratosthenes**() that solves this problem by implementing the Sieve of Eratosthenes. (Use a subroutine if it's helpful.)

Implementing SieveOfEratosthenes

```
SieveOfEratosthenes(n)
    primes ← array of n + 1 true booleans
    primes[0] ← false
    primes[1] ← false
    for every integer p from 2 to √n
        if primes[p] = true
            primes ← CrossOff(primes, p)
    return primes
```

```
CrossOff(primes, p)
    for every multiple k of p from 2p to n
        primes[k] ← false
    return primes
```

Implementing SieveOfEratosthenes

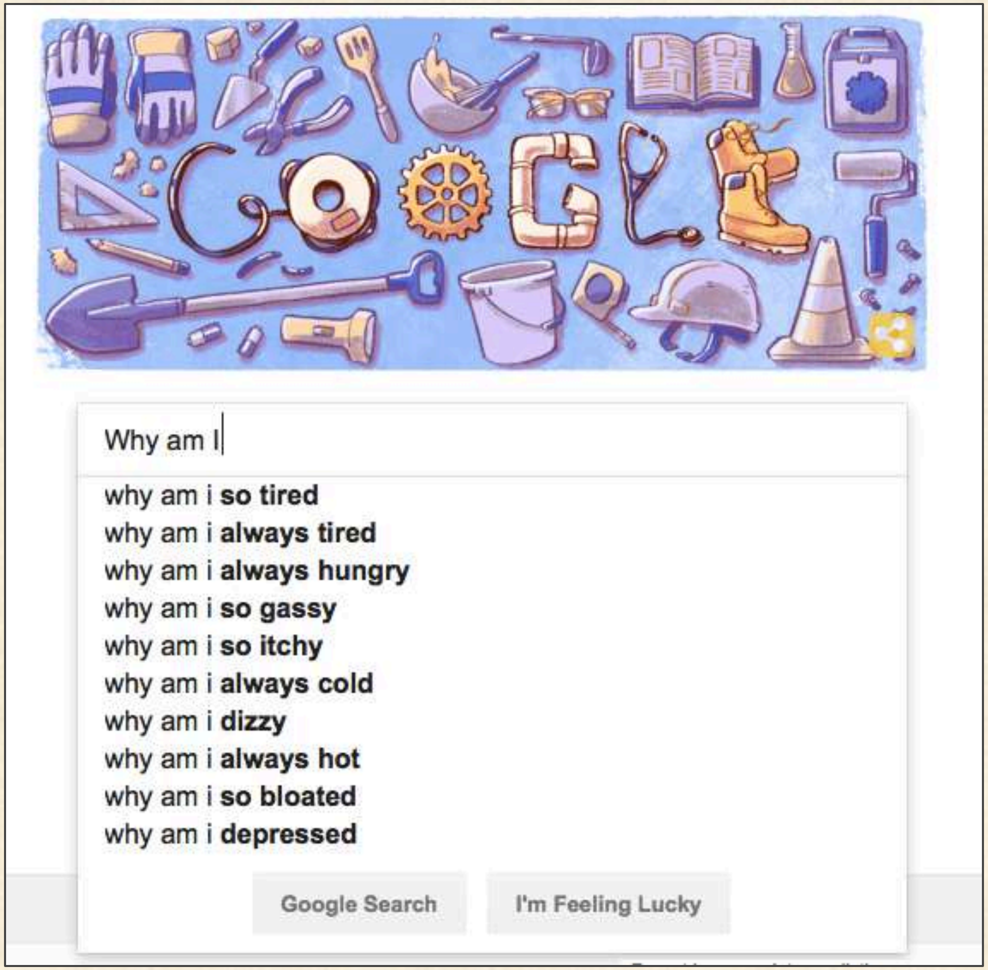
Next time, let's implement the sieve of Eratosthenes in Go, and compare it to the trivial prime finder in terms of speed. *Can it really be that much faster?*

But … what practical use are there for primes in the 21st Century?

CONCLUSION: PUBLIC KEY CRYPTOGRAPHY

Encryption is Vital to Internet Security

Encryption: transforming a message so that it cannot be read by an eavesdropper but can be **decrypted** by the recipient.



Most Encryption Schemes are Symmetric

A **symmetric** encryption scheme uses the same **key** for encrypting/decrypting a message.

HELLO

encrypt
+1 letter

decrypt
-1 letter

IFMMP

Most Encryption Schemes are Symmetric

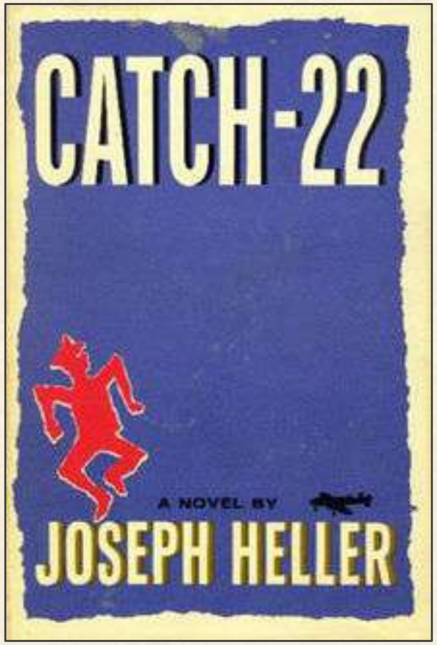
A **symmetric** encryption scheme uses the same **key** for encrypting/decrypting a message.

```
HELLO
```

encrypt +1 letter ↓ ↑ decrypt -1 letter

```
IFMMP
```

Evenif we have a complicated key, it must be *private*: the sender and receiver must agree on the key in advance.



Primes Save the Day

Public key encryption (late 1970s): knowing the key doesn’t make it automatically easy to decrypt!

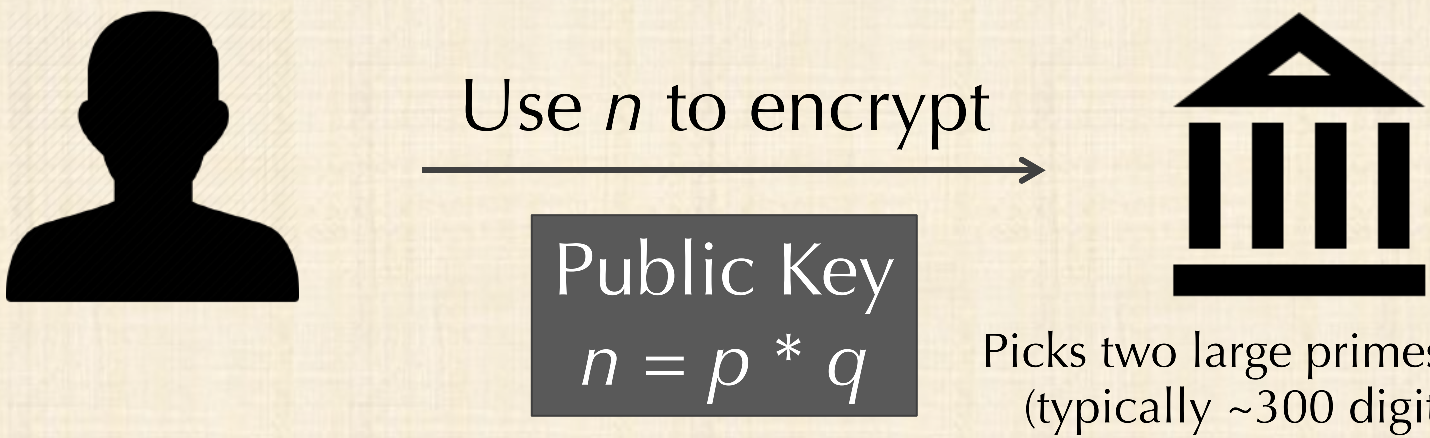
Primes Save the Day

Public key encryption (late 1970s): knowing the key doesn't make it automatically easy to decrypt!

Picks two large primes p and q
(typically ~300 digits long)

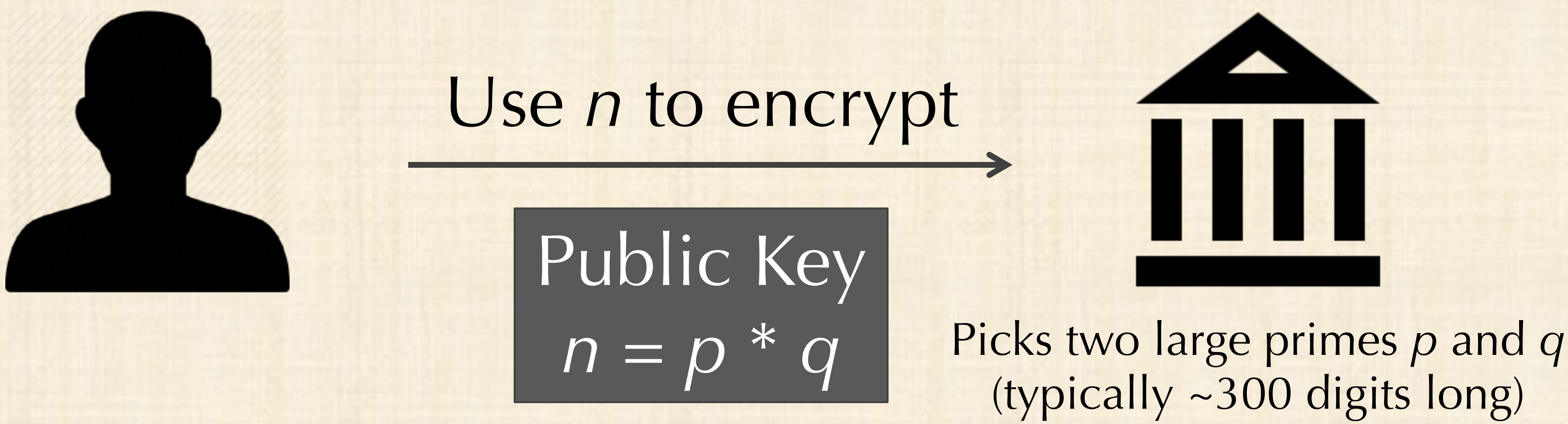
Primes Save the Day

Public key encryption (late 1970s): knowing the key doesn't make it automatically easy to decrypt!



Primes Save the Day

Public key encryption (late 1970s): knowing the key doesn't make it automatically easy to decrypt!

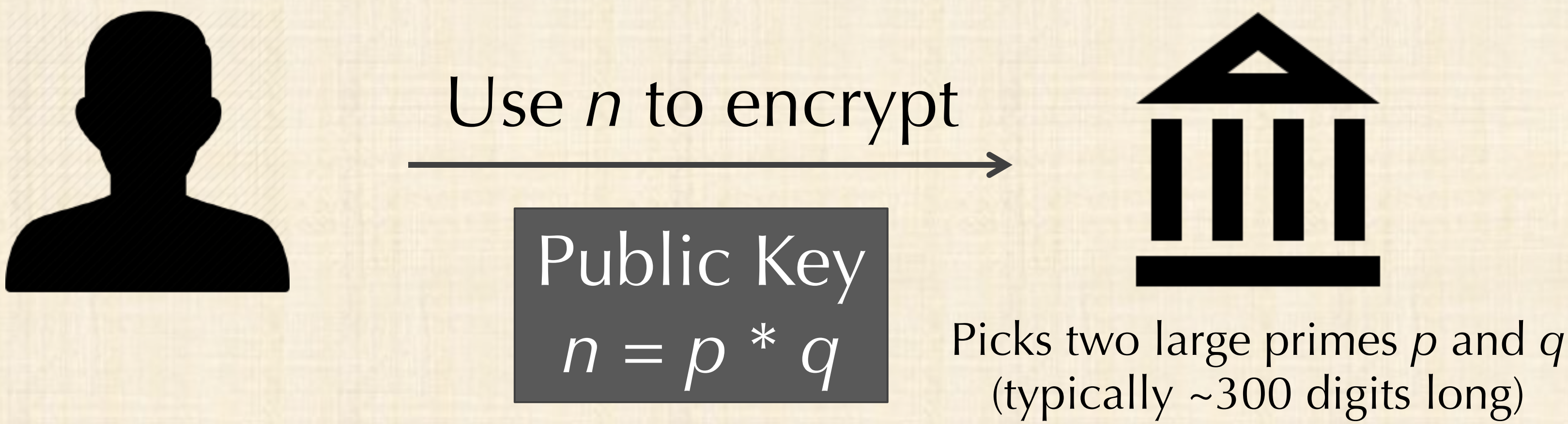


Key Point: The only way to decrypt is by knowing the primes p and q. This makes the key ***asymmetric***.

"But an eavesdropper just has to factor *n*!"

Integer Factorization Problem

- **Input:** An integer *n*.
- **Output:** The factorization of *n*.



"But an eavesdropper just has to factor *n*!"

Integer Factorization Problem

- **Input:** An integer *n*.
- **Output:** The factorization of *n*.

Use *n* to encrypt

Public Key
$n = p * q$

Picks two large primes *p* and *q* (typically ~300 digits long)

Key Point: No one has ever found a "fast" solution to this problem for 600-digit integers ...

Summing Up