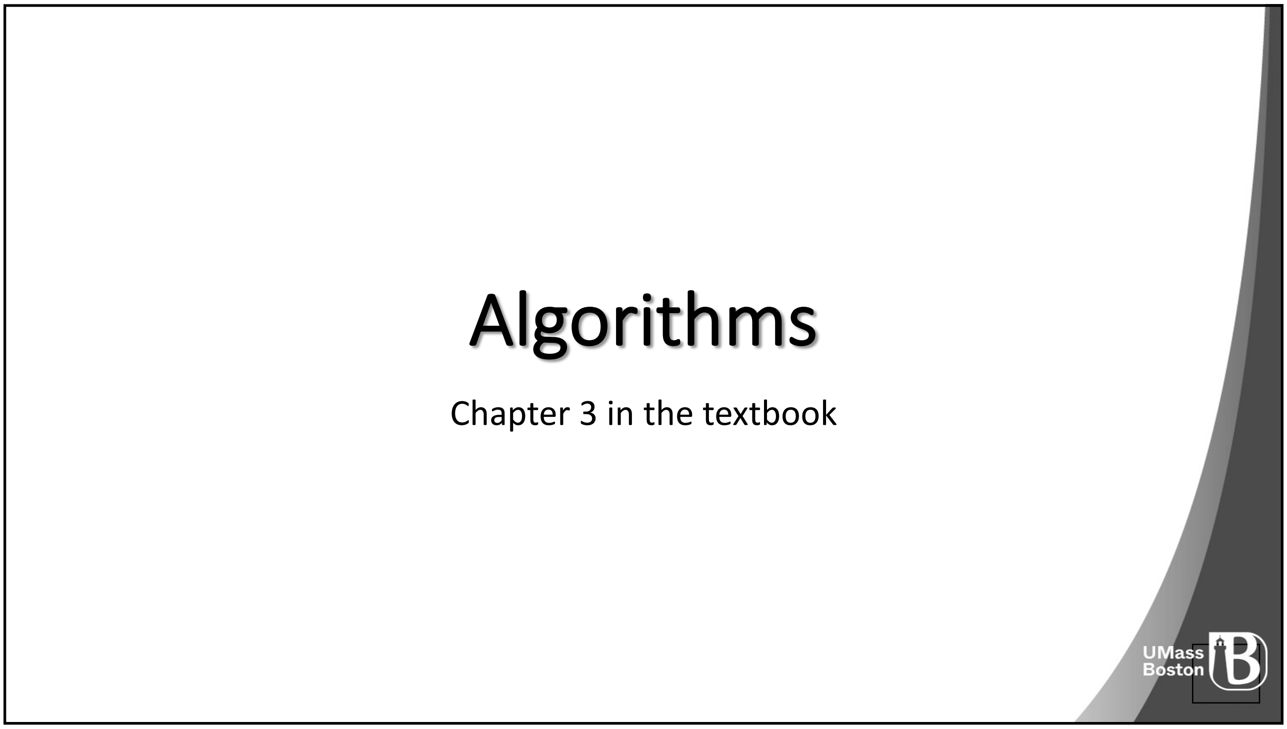
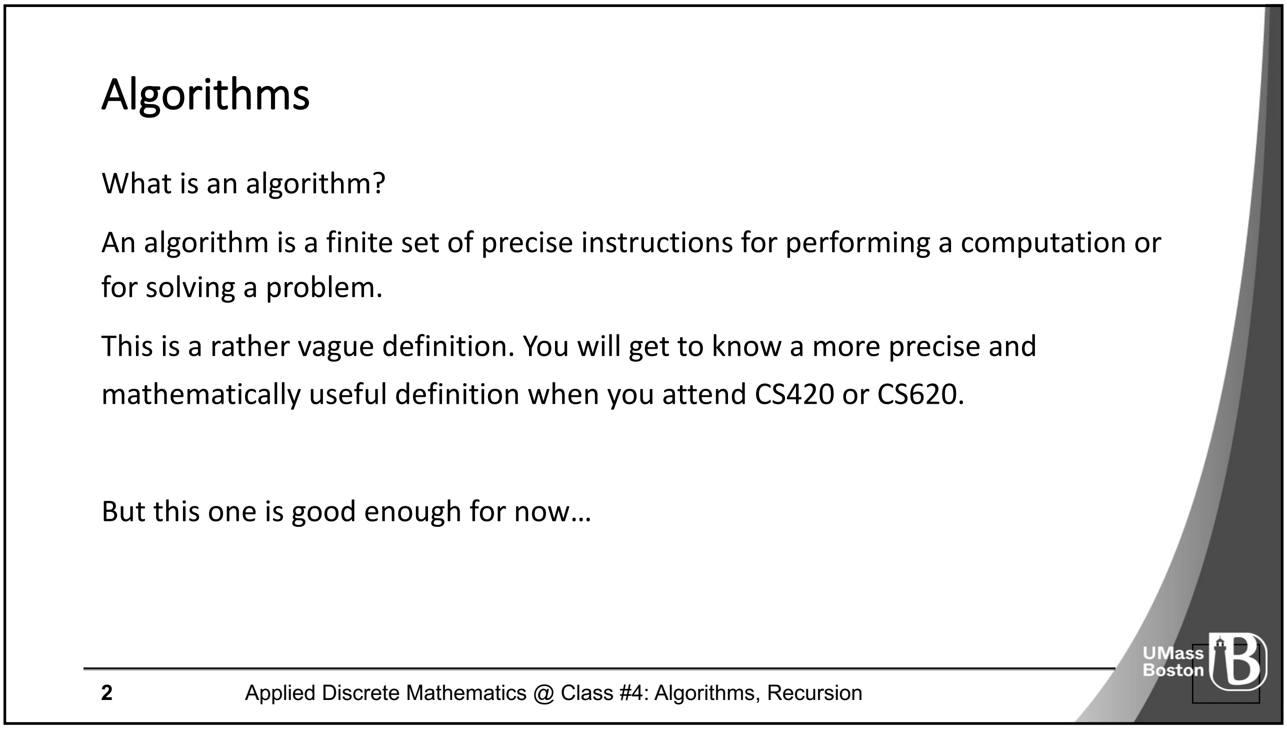


Algorithms

Chapter 3 in the textbook

1



Algorithms

What is an algorithm?

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

This is a rather vague definition. You will get to know a more precise and mathematically useful definition when you attend CS420 or CS620.

But this one is good enough for now…

2

Algorithms

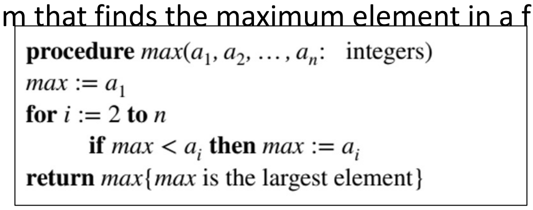
Properties of algorithms:

- **Input** from a specified set,
- **Output** from a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step and
- **Generality** for a class of problems.

Algorithm Examples

We will use a pseudocode to specify algorithms, which slightly reminds us of Basic and Pascal.

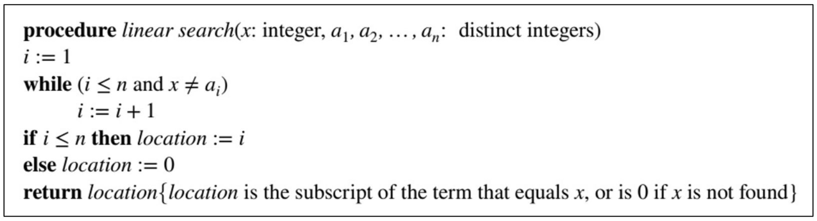
Example: an algorithm that finds the maximum element in a finite sequence



```
procedure $max(a_1, a_2, \ldots, a_n$: integers)
$max := a_1$
for $i := 2$ to $n$
        if $max < a_i$ then $max := a_i$
return $max${$max$ is the largest element}
```

Algorithm Examples

Another example: a linear search algorithm, that is, an algorithm that linearly searches a sequence for a particular element.

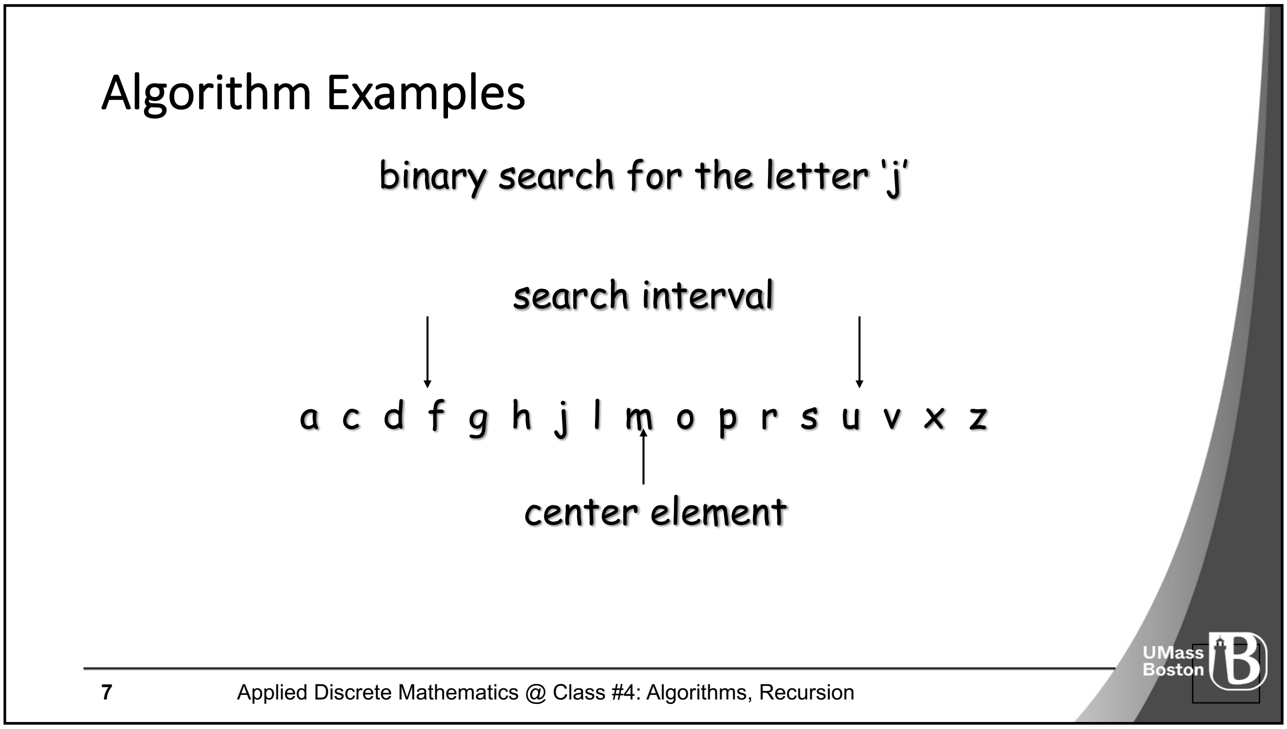


```
procedure linear search(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
i := 1
while ($i \leq n$ and $x \neq a_i$)
        i := i + 1
if $i \leq n$ then location := i
else location := 0
return location{location is the subscript of the term that equals x, or is 0 if x is not found}
```

Algorithm Examples

If the terms in a sequence are ordered, a binary search algorithm is more efficient than linear search.

The binary search algorithm iteratively restricts the relevant search interval until it closes in on the position of the element to be located.



Algorithm Examples

binary search for the letter ‘j’

search interval

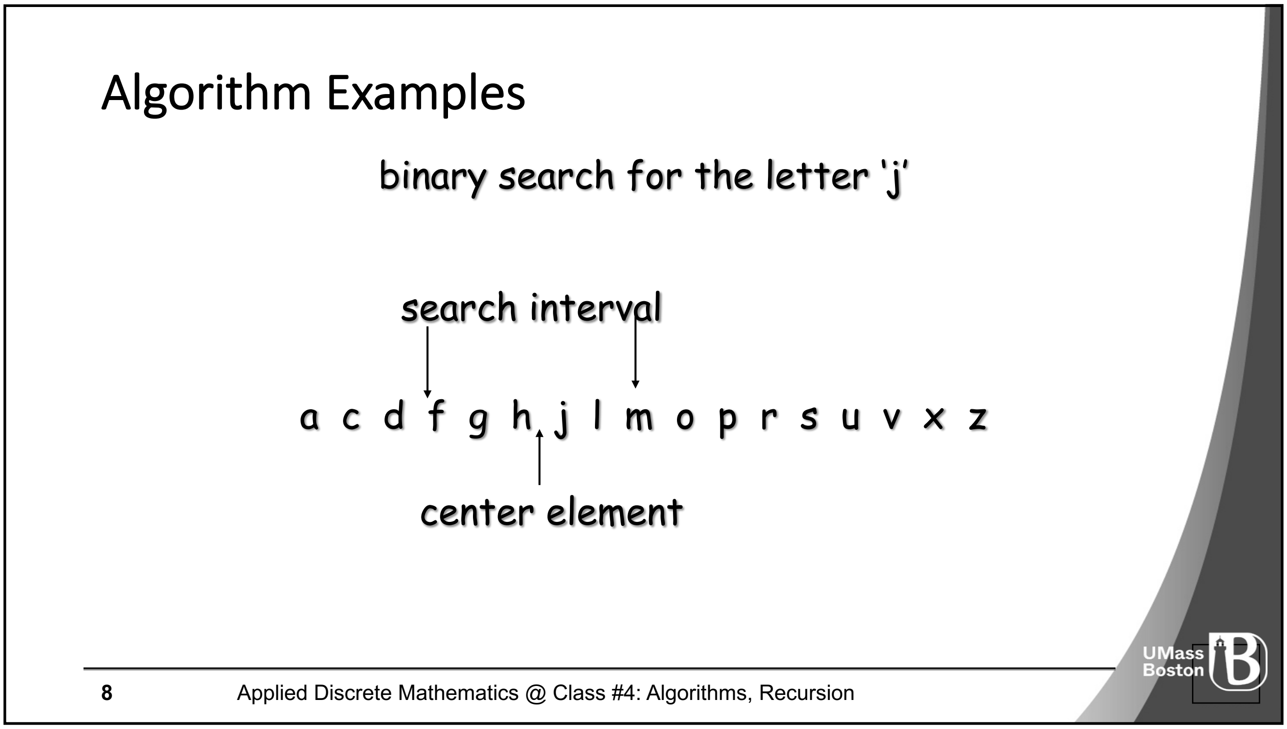
a c d f g h j l m o p r s u v x z

center element

7 Applied Discrete Mathematics @ Class #4: Algorithms, Recursion

UMass Boston

7



Algorithm Examples

binary search for the letter ‘j’

search interval

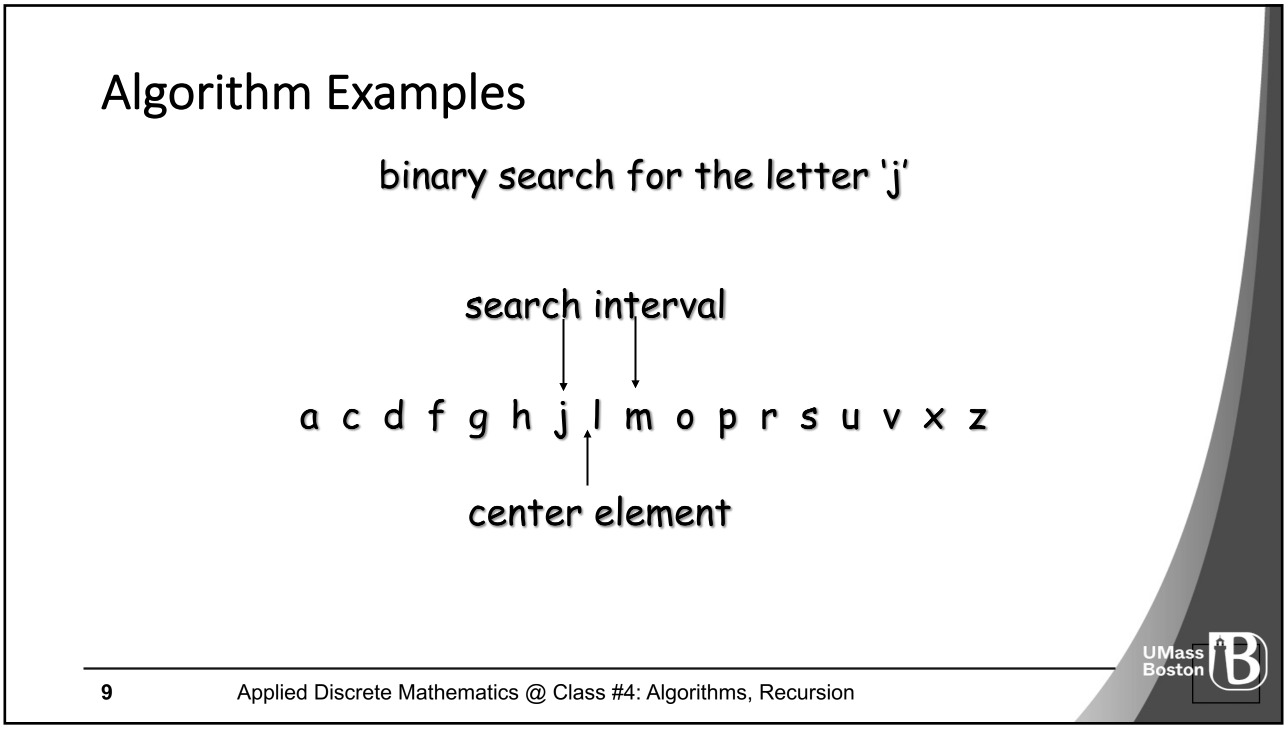
a c d f g h j l m o p r s u v x z

center element

8 Applied Discrete Mathematics @ Class #4: Algorithms, Recursion

UMass Boston

8



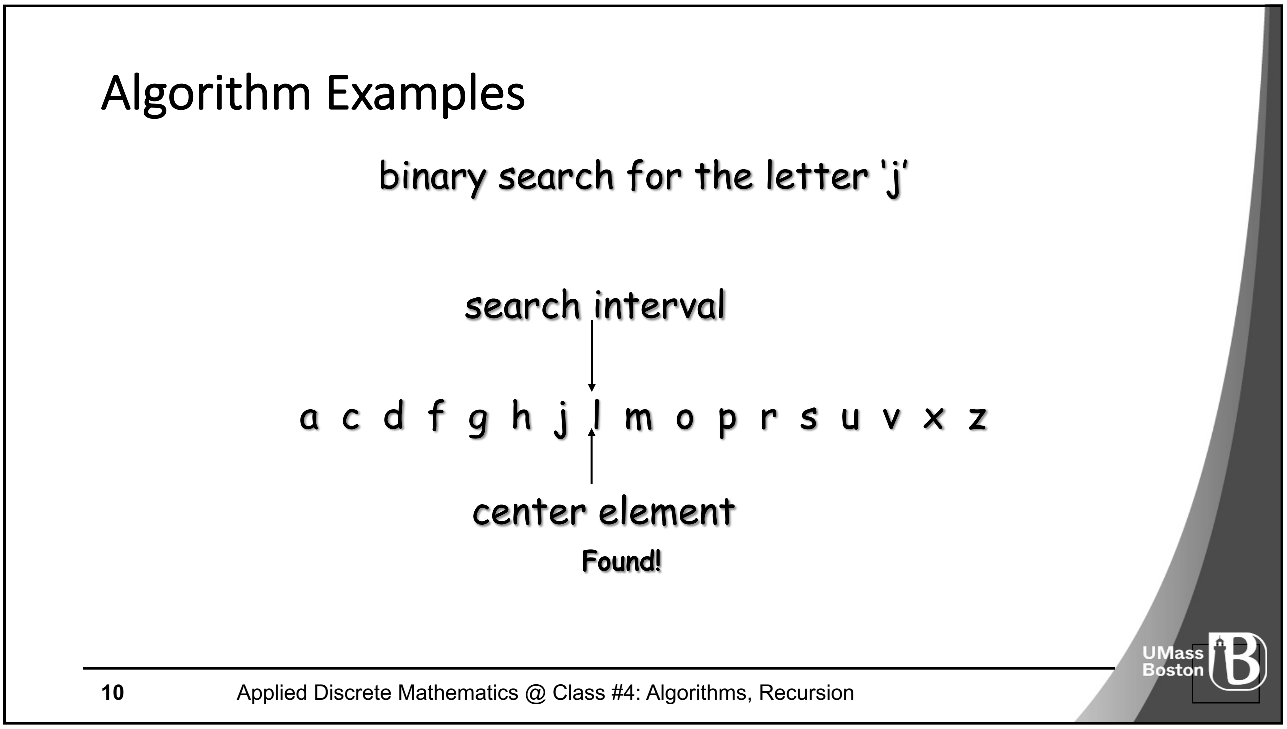
Algorithm Examples

binary search for the letter ‘j’

search interval

a c d f g h j l m o p r s u v x z

center element



Algorithm Examples

binary search for the letter ‘j’

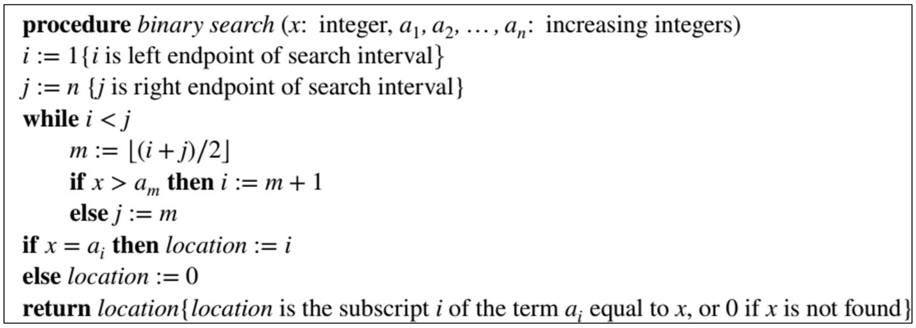
search interval

a c d f g h j l m o p r s u v x z

center element

Found!

Algorithm Examples



```
procedure binary search ($x$: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {$i$ is left endpoint of search interval}
$j := n$ {$j$ is right endpoint of search interval}
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
if $x = a_i$ then location := $i$
else location := 0
return location {location is the subscript $i$ of the term $a_i$ equal to $x$, or 0 if $x$ is not found}
```

Algorithm Examples

Obviously, on sorted sequences, binary search is more efficient than linear search.

How can we analyze the efficiency of algorithms?

We can measure the

- **time** (number of elementary computations) and
- **space** (number of memory cells) that the algorithm requires.

These measures are called **computational complexity** and **space complexity**, respectively.

Complexity

What is the time complexity of the linear search algorithm?

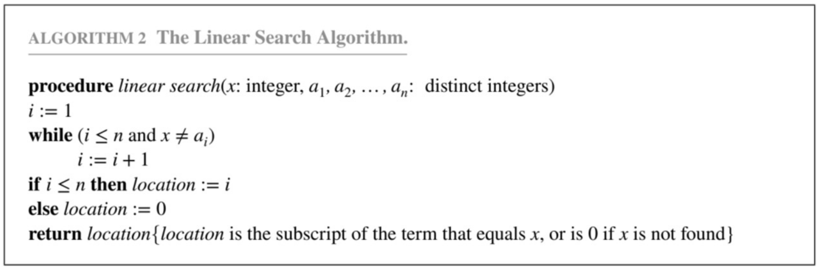
We will determine the **worst-case** number of comparisons as a function of the number n of terms in the sequence.

The worst case for the linear algorithm occurs when the element to be located is not included in the sequence.

In that case, every item in the sequence is compared to the element to be located.

Algorithm Examples

Here is the linear search algorithm again:

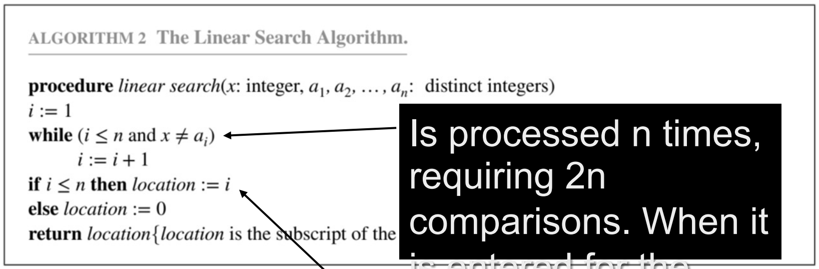


ALGORITHM 2 The Linear Search Algorithm.

```
procedure linear search($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
        $i := i + 1$
if $i \leq n$ then location := $i$
else location := 0
return location{location is the subscript of the term that equals $x$, or is 0 if $x$ is not found}
```

Algorithm Examples

Here is the linear search algorithm again:



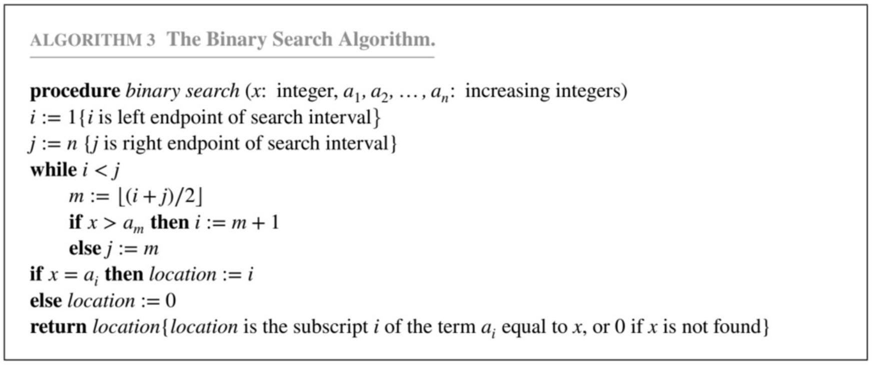
ALGORITHM 2 The Linear Search Algorithm.

```
procedure linear search($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \le n$ and $x \ne a_i$)
    $i := i + 1$
if $i \le n$ then location := $i$
else location := 0
return location{location is the subscript of the
```

Is processed n times, requiring 2n comparisons. When it is entered for the (n+1)th time, only the executed and terminates the loop

Finally, this comparison is excuted, so in the worst-case, the time

Reminder: Binary Search Algorithm



ALGORITHM 3 The Binary Search Algorithm.

```
procedure binary search ($x$: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1${$i$ is left endpoint of search interval}
$j := n$ {$j$ is right endpoint of search interval}
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
if $x = a_i$ then location := $i$
else location := 0
return location{location is the subscript $i$ of the term $a_i$ equal to $x$, or 0 if $x$ is not found}
```

What is the time complexity of the algorithm?

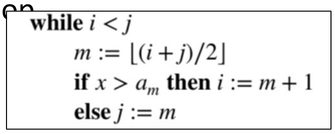
Again, we will determine the **worst-case** number of comparisons as a function of the number n of terms in the sequence.

Complexity-Binary Search Algorithm

Let us assume there are $n = 2^k$ elements in the list, which means that $k = \log n$.

If n is not a power of 2, it can be considered part of a larger list, where $2^k < n < 2^{k+1}$.

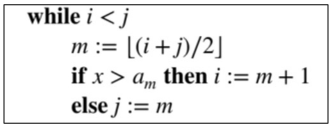
In the first cycle of the loop



```
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
```

- ✓The search interval is restricted to 2^{k-1} elements,
- ✓Using 2 comparisons

Complexity-Binary Search Algorithm



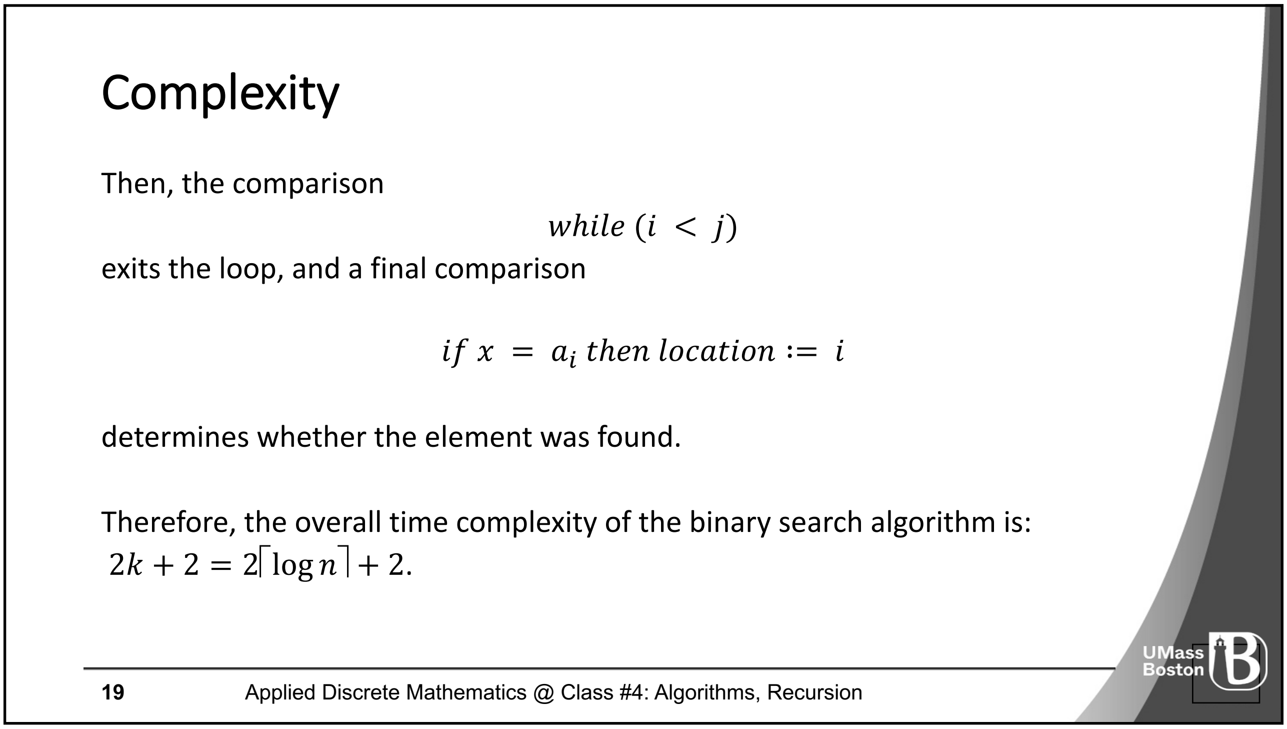
```
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
```

In the second cycle of the loop

- ✓The search interval is restricted to 2^{k-2} elements,
- ✓Using 2 comparisons (again)

… this is repeated until only 1 (2^0) element left in the search interval

- ✓At this point, 2k comparisons has been conducted



Complexity

Then, the comparison

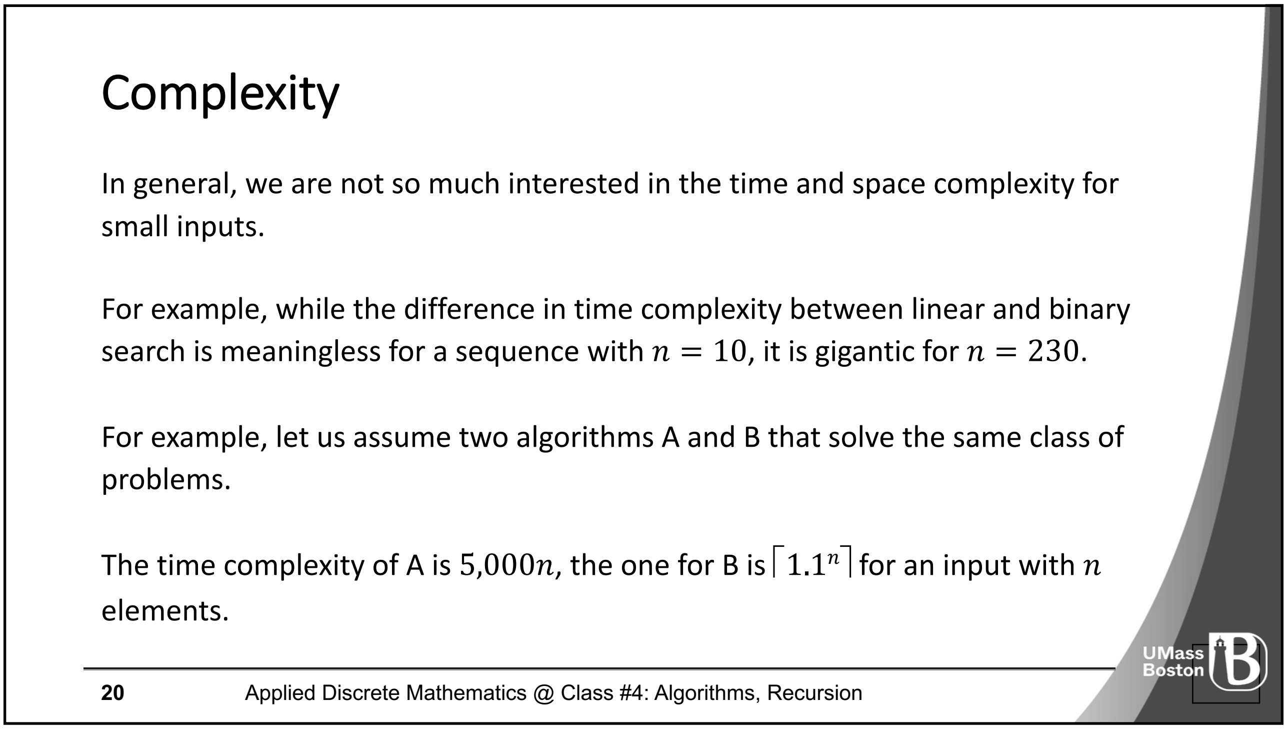
$$while\ (i < j)$$

exits the loop, and a final comparison

$$if\ x = a_i\ then\ location := i$$

determines whether the element was found.

Therefore, the overall time complexity of the binary search algorithm is:
$2k + 2 = 2\lceil \log n \rceil + 2.$



Complexity

In general, we are not so much interested in the time and space complexity for small inputs.

For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with $n = 10$, it is gigantic for $n = 230$.

For example, let us assume two algorithms A and B that solve the same class of problems.

The time complexity of A is $5{,}000n$, the one for B is $\lceil 1.1^n \rceil$ for an input with n elements.

Complexity

Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	5,000n	$\lceil 1.1^{n} \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^{9}$	$4.8 \cdot 10^{41392}$

This means that algorithm B cannot be used for large inputs, while running algorithm A is still feasible.

Complexity

So what important is the **growth** of the complexity functions.

The growth of time and space complexity with increasing input size n is a suitable measure for the **comparison** of algorithms.

The Growth of Functions

The growth of functions is usually described using the **big-O notation**.

Definition: Let f and g be functions from the integers or the real numbers to the real numbers. We say that $f(x)$ is $\boldsymbol{O}(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|, \text{ whenever } x > k.$$

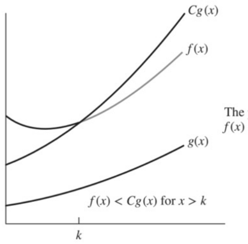
The Growth of Functions

When we analyze the growth of **complexity functions**, $f(x)$ and $g(x)$ are always positive.

Therefore, we can simplify the **big-O** requirement to

$$f(x) \leq C \cdot g(x) \text{ whenever } x > k.$$

If we want to show that $f(x)$ is $\boldsymbol{O}(g(x))$, we only need to find **one** pair (C, k) (which is never unique).



The part of the graph of $f(x)$ that satisfies $f(x) < Cg(x)$ is shown in color.

FIGURE 2 **The function $f(x)$ is $O(g(x))$.**

The Growth of Functions

The idea behind the **big-O** notation is to establish an **upper boundary** for the growth of a function $f(x)$ for large x.

This boundary is specified by a function $g(x)$ that is usually much **simpler** than $f(x)$.

We accept the constant C in the requirement

$$f(x) \leq C \cdot g(x) \text{ whenever } x > k,$$

because $\boldsymbol{C}$ **does not grow with** $\boldsymbol{x}$

We are only interested in large x, so it is OK if $f(x) > C \cdot g(x)$ for $x \leq k$.

The Growth of Functions

Example: Show that $f(x) = x^2 + 2x + 1$ is $\boldsymbol{O}(x^2)$.

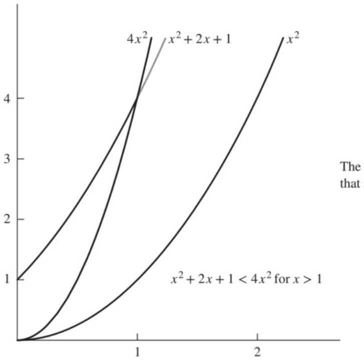
For $x > 1$ we have:

$$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2$$
$$\Rightarrow x^2 + 2x + 1 \leq 4x^2$$

Therefore, for $C = 4$ and $k = 1$:

$f(x) \leq Cx^2$ whenever $x > k$.

$\Rightarrow f(x) \; is \; O(x^2)$.



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in color.

The Growth of Functions

Question: If $f(x)$ is $\boldsymbol{O}(x^2)$, is it also $\boldsymbol{O}(x^3)$?

Yes. x^3 grows faster than x^2, so x^3 grows also faster than $f(x)$.

Therefore, we always have to find the **smallest** simple function $g(x)$ for which $f(x)$ is $\boldsymbol{O}(g(x))$

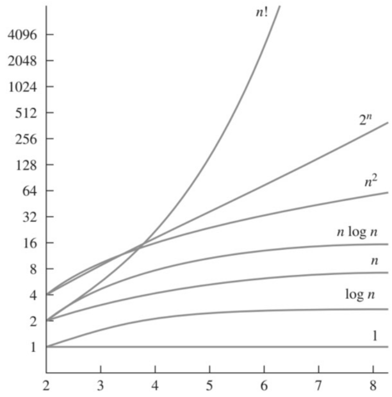
The Growth of Functions

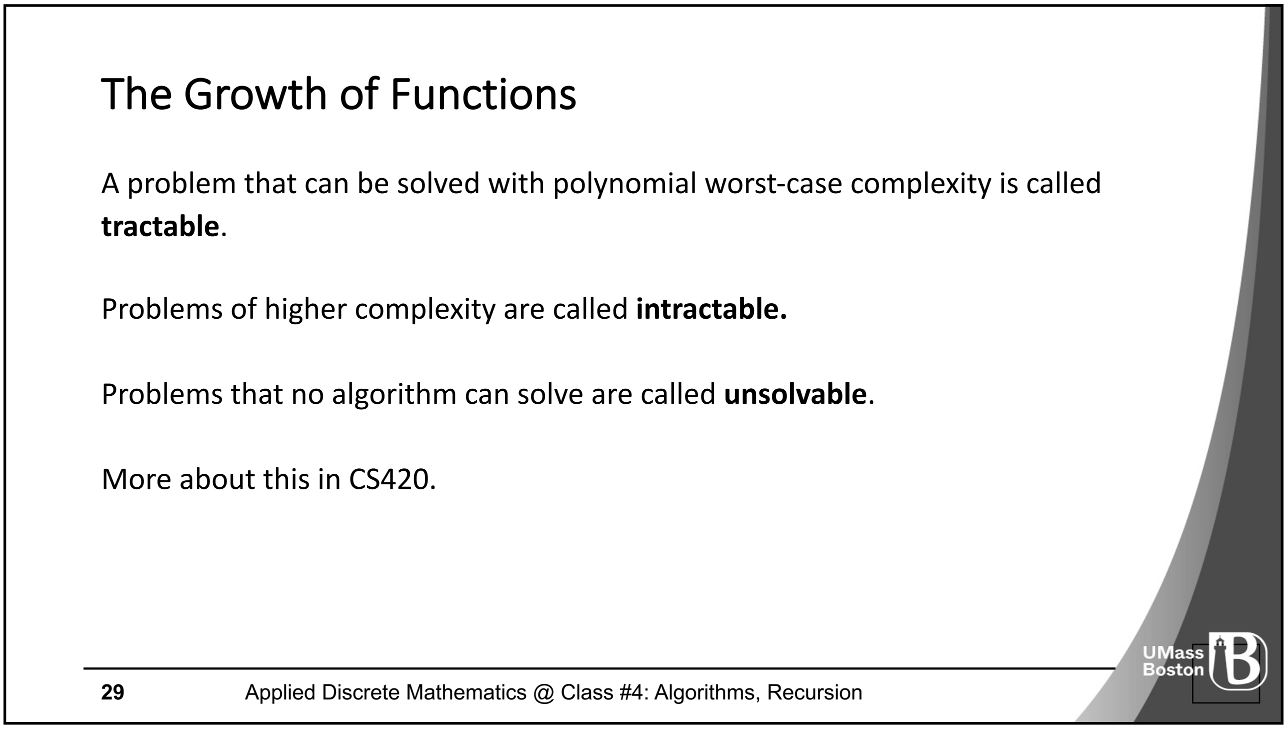
"Popular" functions g(n) are:

- $nlogn, 1, 2^n, n^2, n!, n, n^3, \log n$

Listed from slowest to fastest growth:

- 1
- log n
- n
- n log n
- n^2
- n^3
- 2^n
- n!





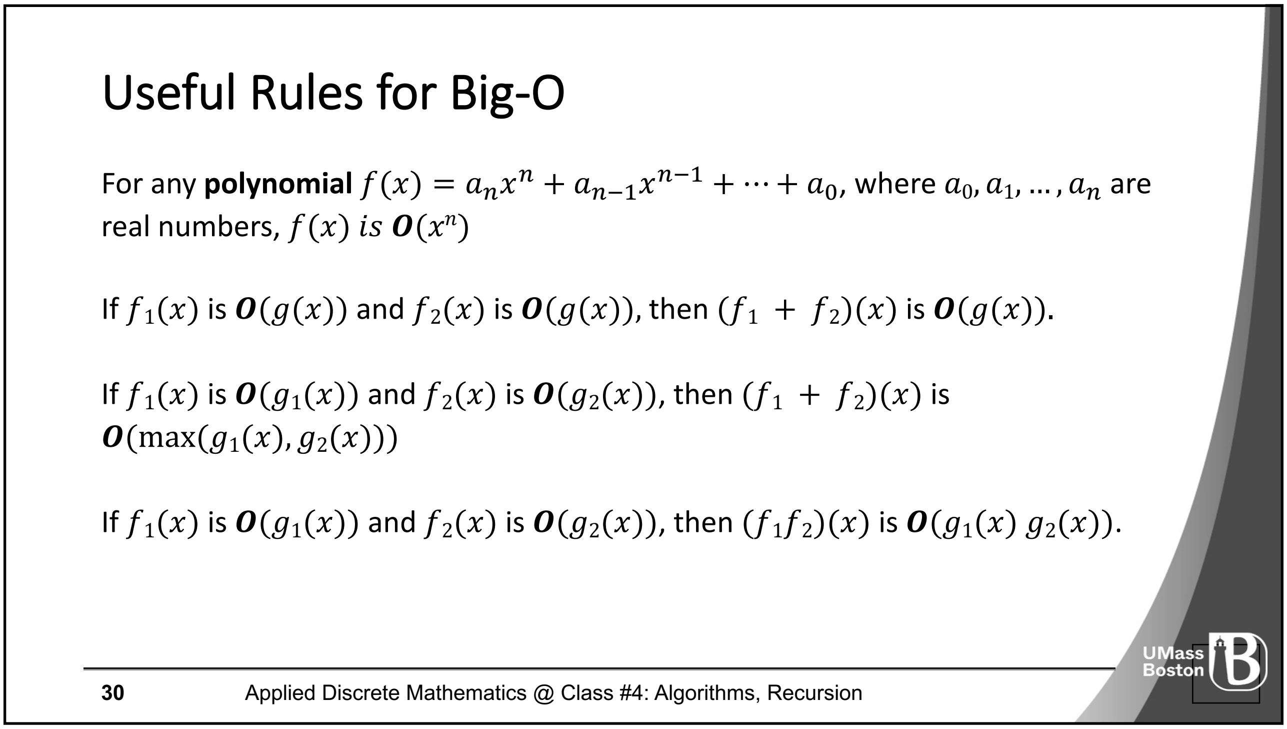
The Growth of Functions

A problem that can be solved with polynomial worst-case complexity is called **tractable**.

Problems of higher complexity are called **intractable.**

Problems that no algorithm can solve are called **unsolvable**.

More about this in CS420.



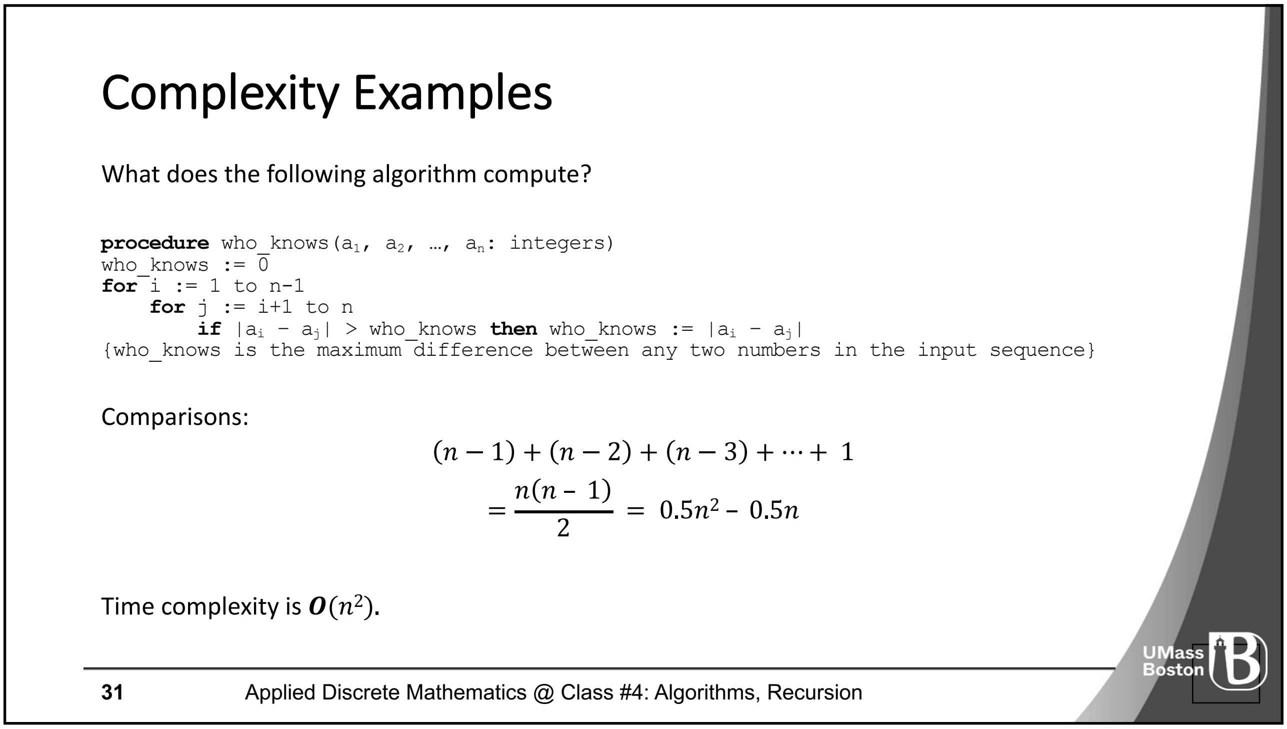
Useful Rules for Big-O

For any **polynomial** $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$, where $a_0, a_1, \ldots, a_n$ are real numbers, $f(x)\ is\ \boldsymbol{O}(x^n)$

If $f_1(x)$ is $\boldsymbol{O}(g(x))$ and $f_2(x)$ is $\boldsymbol{O}(g(x))$, then $(f_1 + f_2)(x)$ is $\boldsymbol{O}(g(x))$.

If $f_1(x)$ is $\boldsymbol{O}(g_1(x))$ and $f_2(x)$ is $\boldsymbol{O}(g_2(x))$, then $(f_1 + f_2)(x)$ is $\boldsymbol{O}(\max(g_1(x), g_2(x)))$

If $f_1(x)$ is $\boldsymbol{O}(g_1(x))$ and $f_2(x)$ is $\boldsymbol{O}(g_2(x))$, then $(f_1 f_2)(x)$ is $\boldsymbol{O}(g_1(x)\ g_2(x))$.



Complexity Examples

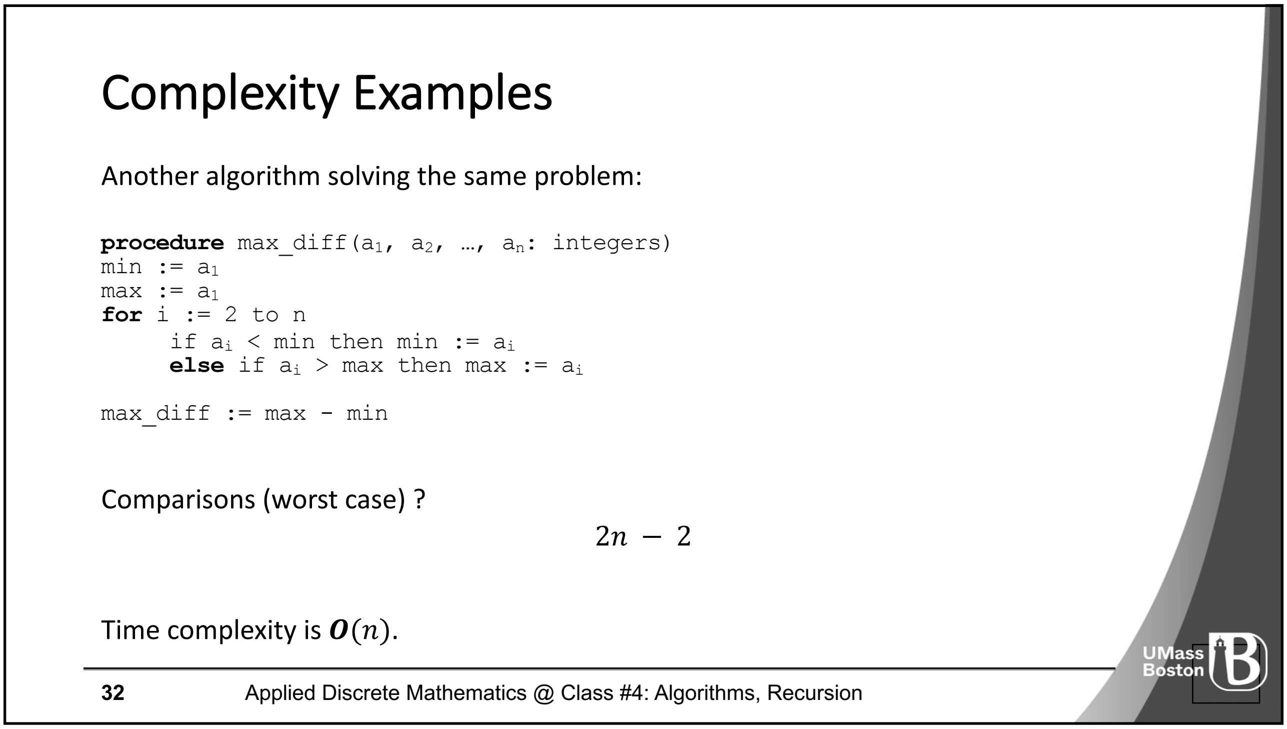
What does the following algorithm compute?

```
procedure who_knows($a_1$, $a_2$, …, $a_n$: integers)
who_knows := 0
for i := 1 to n-1
    for j := i+1 to n
        if $|a_i - a_j|$ > who_knows then who_knows := $|a_i - a_j|$
{who_knows is the maximum difference between any two numbers in the input sequence}
```

Comparisons:

$$(n-1)+(n-2)+(n-3)+\cdots+ 1$$
$$=\frac{n(n-1)}{2} = 0.5n^2 - 0.5n$$

Time complexity is $\boldsymbol{O}(n^2)$.



Complexity Examples

Another algorithm solving the same problem:

```
procedure max_diff($a_1$, $a_2$, …, $a_n$: integers)
min := $a_1$
max := $a_1$
for i := 2 to n
     if $a_i$ < min then min := $a_i$
     else if $a_i$ > max then max := $a_i$

max_diff := max - min
```

Comparisons (worst case) ?

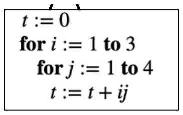
$$2n - 2$$

Time complexity is $\boldsymbol{O}(n)$.

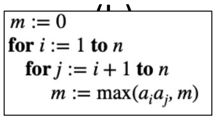
In-class exceries

Give a Big-O estimate for the number of operations (+ or *) that used the following segment of an algorithm.

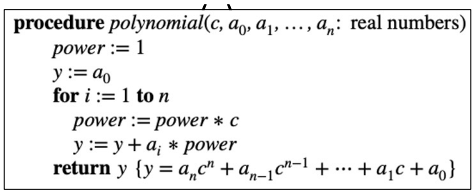
(Do not count additions used to increment the loop variable.)



```
t := 0
for i := 1 to 3
  for j := 1 to 4
    t := t + ij
```



```
m := 0
for i := 1 to n
  for j := i + 1 to n
     m := max($a_ia_j$, m)
```



```
procedure polynomial(c, $a_0, a_1, \dots, a_n$: real numbers)
    power := 1
    y := $a_0$
    for i := 1 to n
       power := power ∗ c
       y := y + $a_i$ ∗ power
    return y {$y = a_nc^n + a_{n-1}c^{n-1} + \cdots + a_1c + a_0$}
```

Induction

Chapter 5 in the textbook

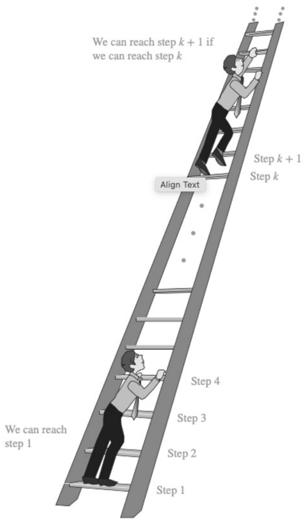
Mathematical Induction

Suppose we have an infinite ladder, and we want to know whether wen can reach every step on this ladder.

1. We can reach the first rung of the ladder.
2. If we can reach a particular rung of the ladder, then we can reach to the next rung.

Mathematical induction is an extremely important proof technique that can be used to prove assertions of this type.

mathematical induction is used extensively to prove results about a large variety of discrete objects



Induction

The **principle of mathematical induction** is a useful tool for proving that a certain predicate is true for **all natural numbers**.

It cannot be used to discover theorems, but only to prove them.

Induction

If we have a propositional function P(n), and we want to prove that P(n) is true for any natural number n, we do the following:

1. Show that P(0) is true.
   (basis step)

2. Show that if P(n) then P(n + 1) for any $n \in \mathbb{N}$.
   (inductive step)

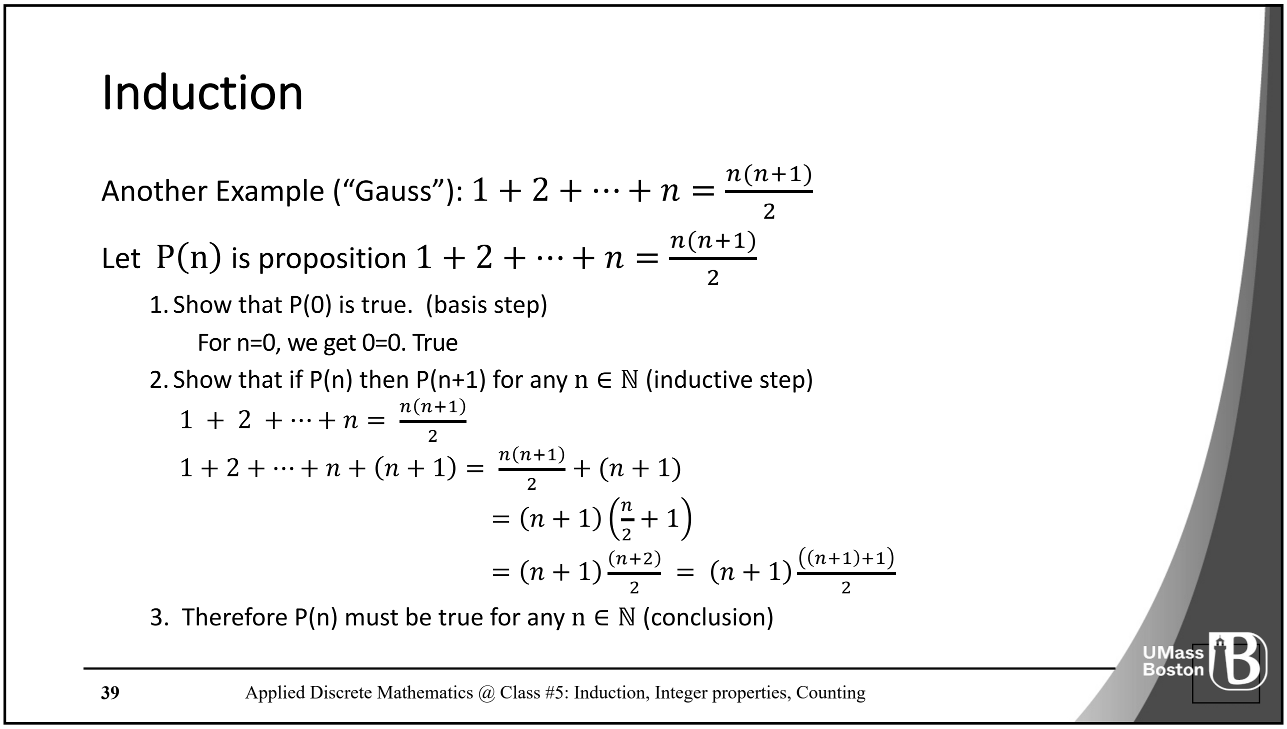
3. Then P(n) must be true for any $n \in \mathbb{N}$.
   (conclusion)

Induction

Example: Show that $n < 2^n$ for all positive integers n.

Let P(n) be the proposition "$n < 2^n$."

- Show that P(1) is true. (basis step)
  - P(1) is true, because $1 < 2^1 = 2$.
- Show that if P(n) is true, then P(n + 1) is true. (inductive step)
  - Assume that $n < 2^n$ is true.
  - We need to show that P(n + 1) is true, i.e. $n + 1 < 2^{n+1}$
  - We start from $n < 2^n$:
    - $n + 1 < 2^n + 1 \leq 2^n + 2^n = 2^{n+1}$
  - Therefore, if $n < 2^n$ then $n + 1 < 2^{n+1}$
- Then P(n) must be true for any positive integer. (conclusion)
  - $n < 2^n$ is true for any positive integer.
- End of proof.



Induction

Another Example ("Gauss"): $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

Let $\mathrm{P(n)}$ is proposition $1 + 2 + \cdots + n = \frac{n(n+1)}{2}$

1. Show that P(0) is true. (basis step)

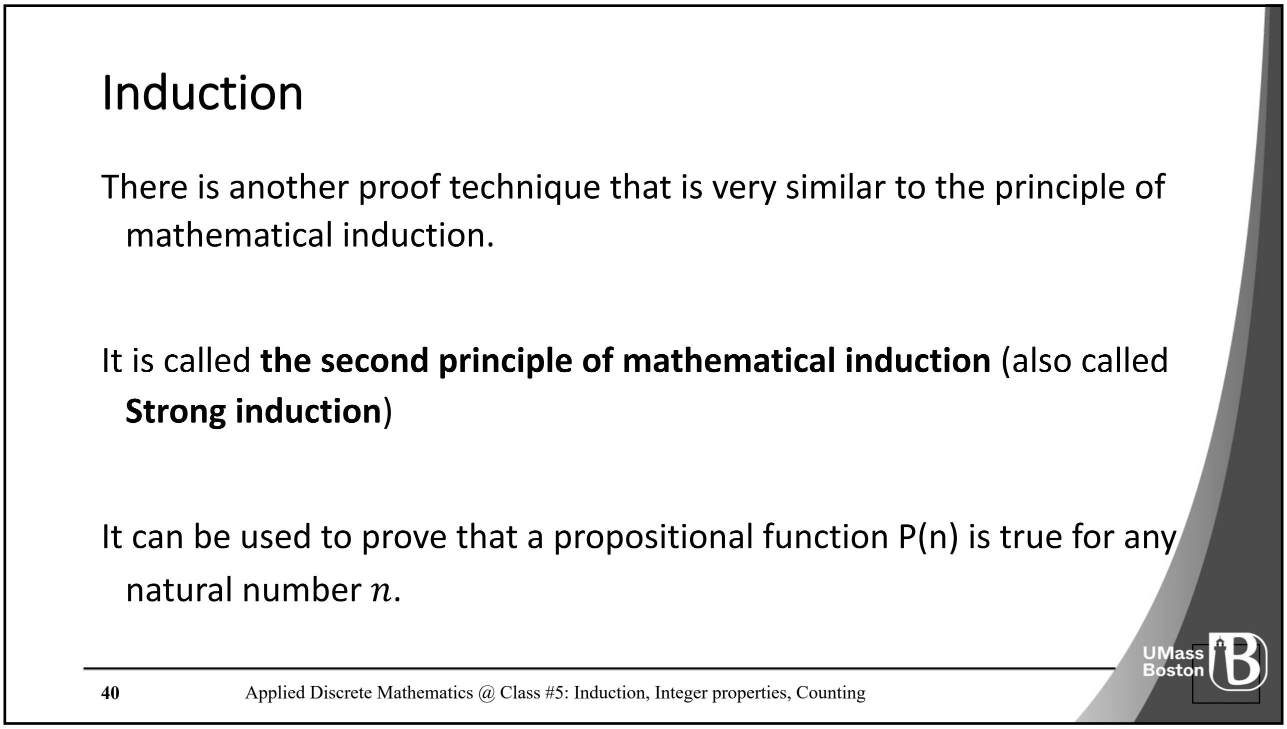
   For n=0, we get 0=0. True

2. Show that if P(n) then P(n+1) for any $\mathrm{n} \in \mathbb{N}$ (inductive step)

$$1 \ + \ 2 \ + \cdots + n = \frac{n(n+1)}{2}$$

$$\begin{aligned} 1 + 2 + \cdots + n + (n+1) &= \frac{n(n+1)}{2} + (n+1) \\ &= (n+1)\left(\frac{n}{2} + 1\right) \\ &= (n+1)\frac{(n+2)}{2} \ = \ (n+1)\frac{((n+1)+1)}{2} \end{aligned}$$

3. Therefore P(n) must be true for any $\mathrm{n} \in \mathbb{N}$ (conclusion)



Induction

There is another proof technique that is very similar to the principle of mathematical induction.

It is called **the second principle of mathematical induction** (also called **Strong induction**)

It can be used to prove that a propositional function P(n) is true for any natural number n.

Induction

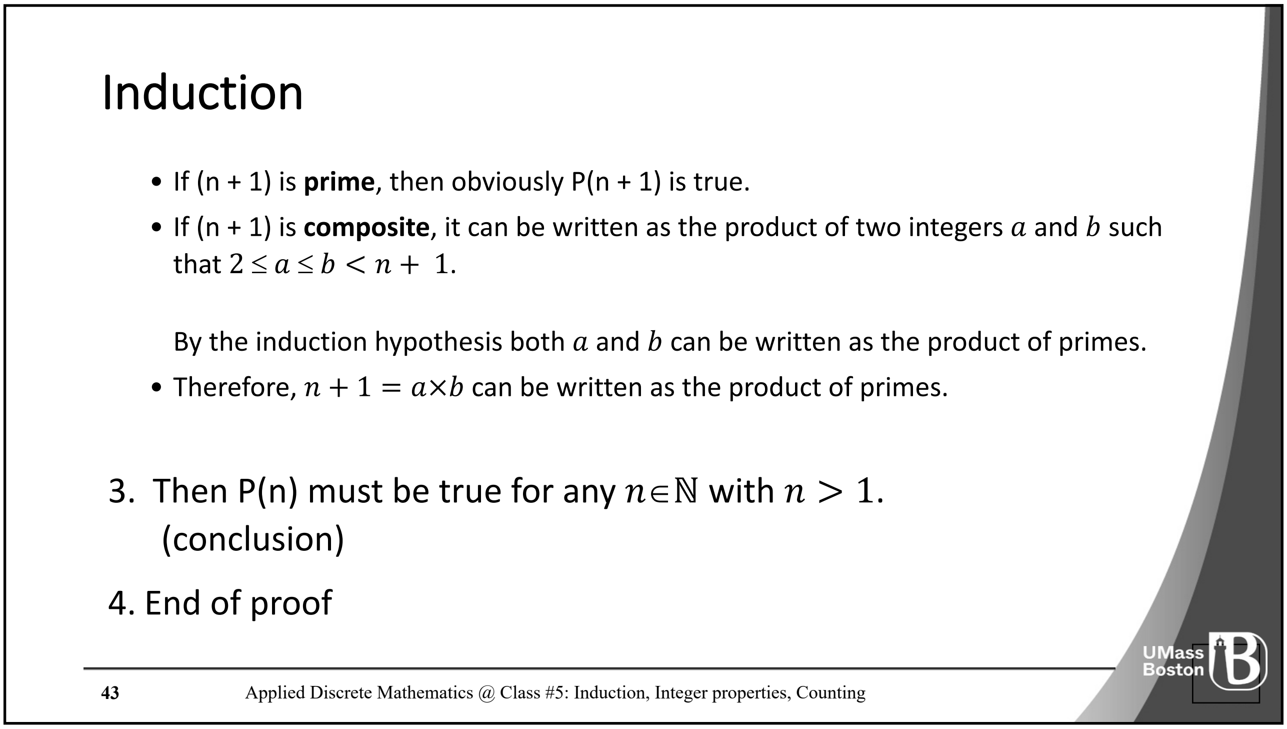
The second principle of mathematical induction:

1. Show that P(0) is true.
   (basis step)
2. Show that if P(0) and P(1) and … and P(n), then P(n + 1) for any $n \in \mathbb{N}$.
   (inductive step)
3. Then P(n) must be true for any $n \in \mathbb{N}$.
   (conclusion)

Induction

Example: Show that every integer greater than 1 can be written as the product of primes.

1. Show that P(2) is true.
   (basis step) Obviously, 2 is the product of one prime: itself.
2. Show that if P(2) and P(3) and … and P(n), then P(n + 1) for any $n \in \mathbb{N}$.
   (inductive step)
   There are two possible cases:
   - If (n + 1) is **prime**, then obviously P(n + 1) is true.
   - If (n + 1) is **composite**, it can be written as the product of two integers a and b such that $2 \le a \le b < n + 1$.



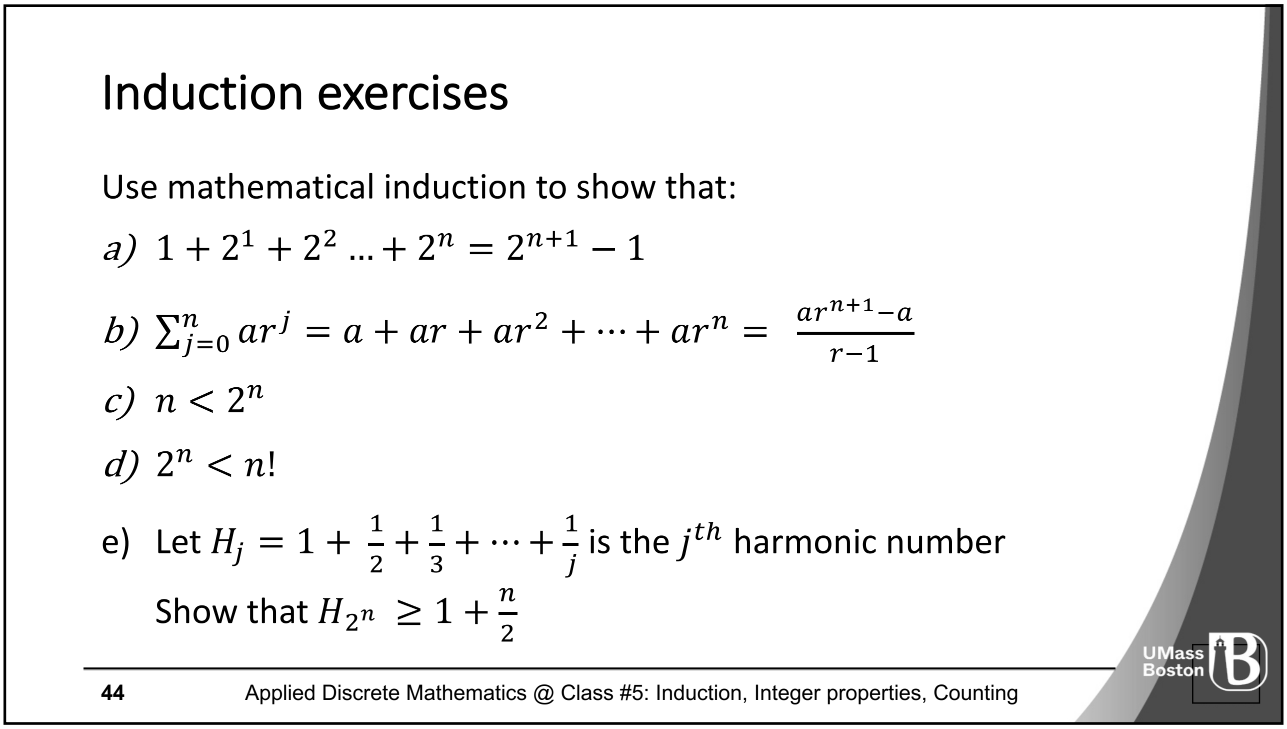
Induction

- If (n + 1) is **prime**, then obviously P(n + 1) is true.
- If (n + 1) is **composite**, it can be written as the product of two integers a and b such that $2 \leq a \leq b < n + 1$.

  By the induction hypothesis both a and b can be written as the product of primes.
- Therefore, $n + 1 = a \times b$ can be written as the product of primes.

3. Then P(n) must be true for any $n \in \mathbb{N}$ with $n > 1$. (conclusion)

4. End of proof



Induction exercises

Use mathematical induction to show that:

a) $1 + 2^1 + 2^2 \ldots + 2^n = 2^{n+1} - 1$

b) $\sum_{j=0}^{n} ar^j = a + ar + ar^2 + \cdots + ar^n = \frac{ar^{n+1} - a}{r - 1}$

c) $n < 2^n$

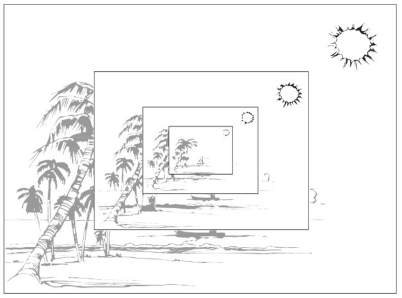
d) $2^n < n!$

e) Let $H_j = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{j}$ is the j^{th} harmonic number

Show that $H_{2^n} \geq 1 + \frac{n}{2}$

Recursive Definitions

Recursion is a principle closely related to mathematical induction.



In a **recursive definition**, an object is defined in terms of itself.

We can recursively define **sequences**, **functions** and **sets**.

Recursively Defined Sequences

Example:

The sequence $\{a_n\}$ of powers of 2 is given by $a_n = 2^n$ for $n = 0, 1, 2, \ldots$

The same sequence can also be defined recursively:

$a_0 = 1$

$a_{n+1} = 2 \cdot a_n$ for $n = 0, 1, 2, \ldots$

Obviously, induction and recursion are similar principles.

Recursively Defined Functions

We can use the following method to define a function with the **natural numbers** as its domain:

1. Specify the value of the function at zero.
2. Give a rule for finding its value at any integer from its values at smaller integers.

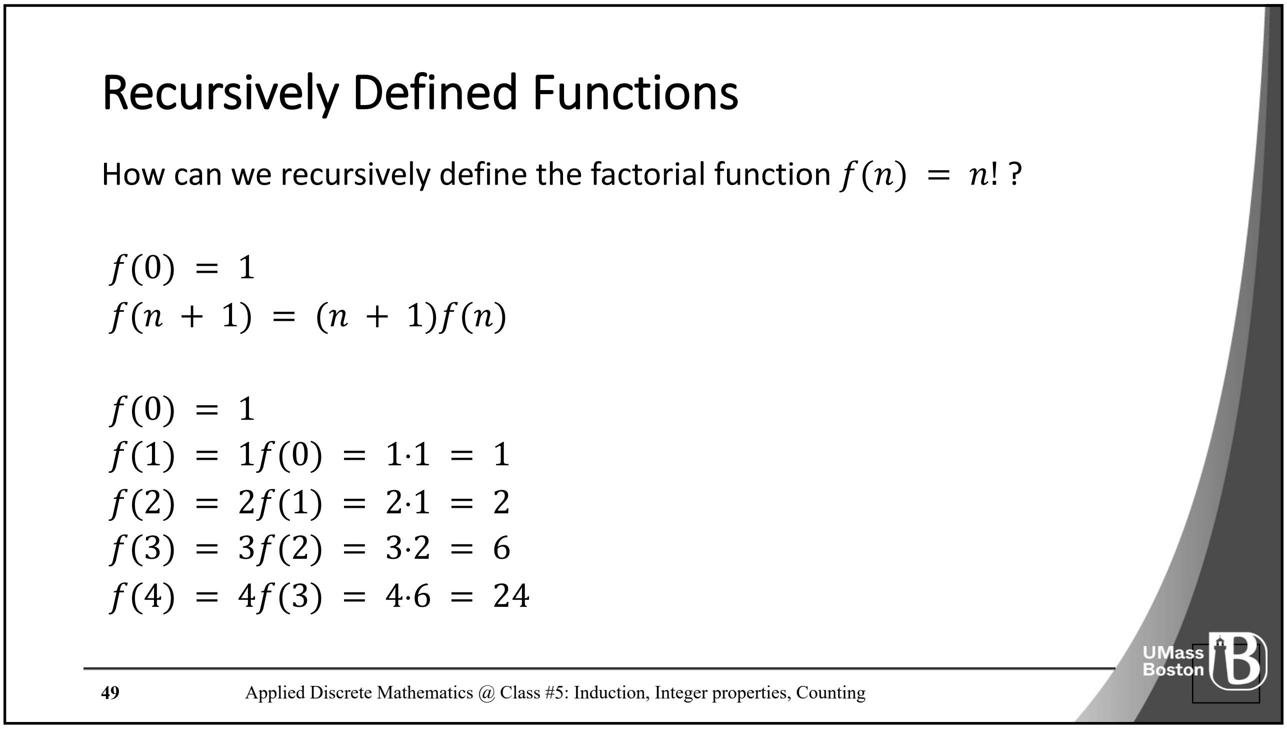
Such a definition is called **recursive** or **inductive definition**.

Recursively Defined Functions

Example:

$$f(0) = 3$$
$$f(n + 1) = 2f(n) + 3$$

$$f(0) = 3$$
$$f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9$$
$$f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21$$
$$f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45$$
$$f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93$$

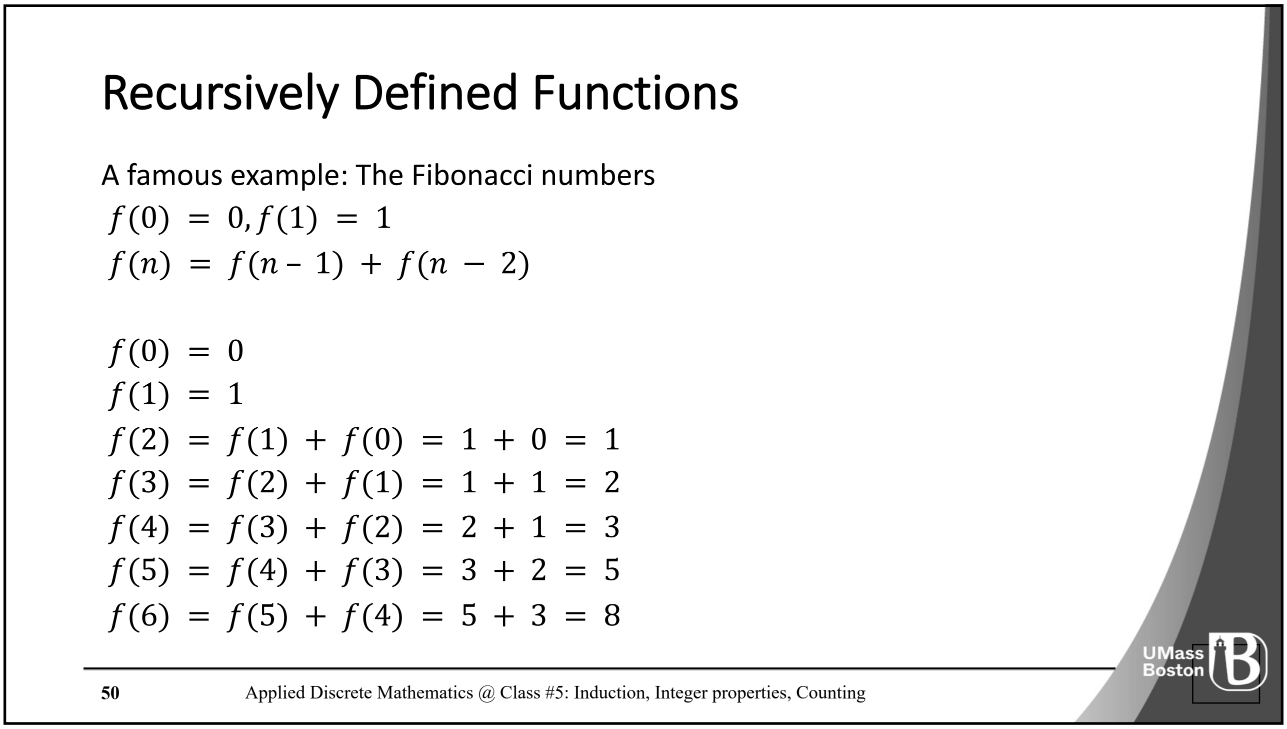


Recursively Defined Functions

How can we recursively define the factorial function $f(n) = n!$?

$f(0) = 1$
$f(n + 1) = (n + 1)f(n)$

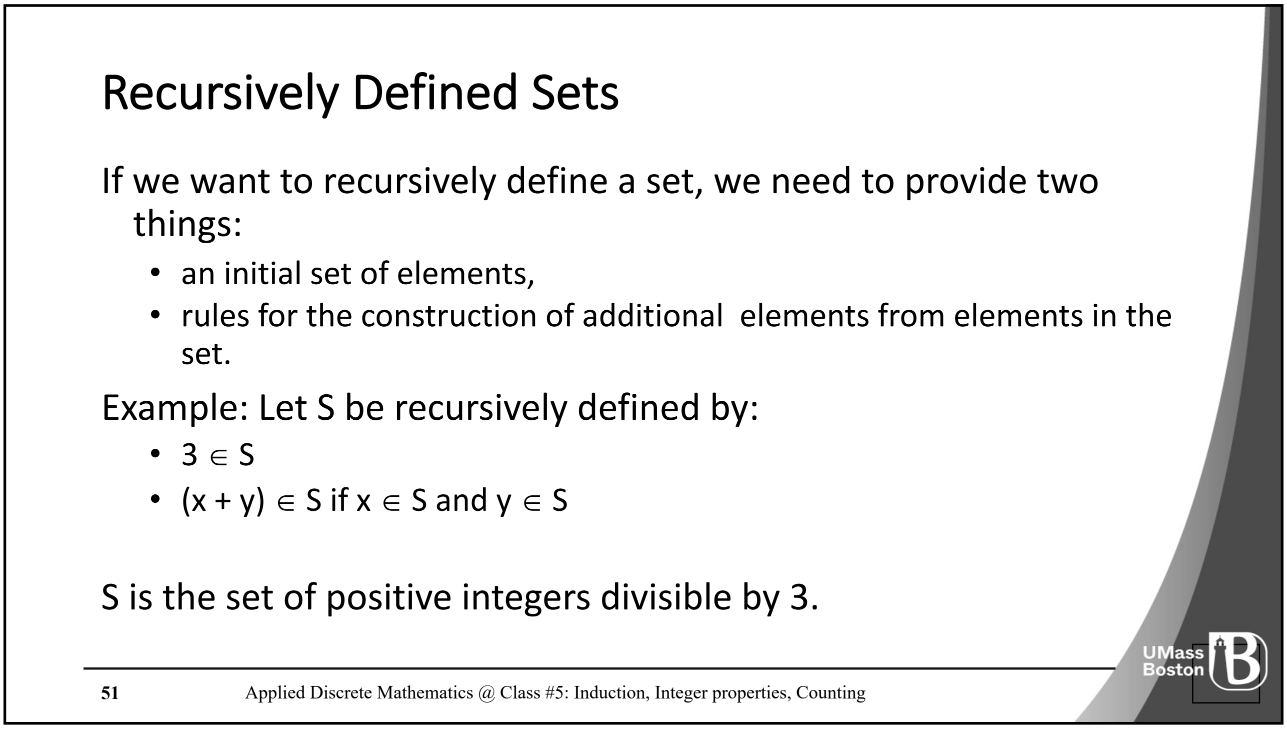
$f(0) = 1$
$f(1) = 1f(0) = 1 \cdot 1 = 1$
$f(2) = 2f(1) = 2 \cdot 1 = 2$
$f(3) = 3f(2) = 3 \cdot 2 = 6$
$f(4) = 4f(3) = 4 \cdot 6 = 24$



Recursively Defined Functions

A famous example: The Fibonacci numbers
$f(0) = 0, f(1) = 1$
$f(n) = f(n - 1) + f(n - 2)$

$f(0) = 0$
$f(1) = 1$
$f(2) = f(1) + f(0) = 1 + 0 = 1$
$f(3) = f(2) + f(1) = 1 + 1 = 2$
$f(4) = f(3) + f(2) = 2 + 1 = 3$
$f(5) = f(4) + f(3) = 3 + 2 = 5$
$f(6) = f(5) + f(4) = 5 + 3 = 8$



Recursively Defined Sets

If we want to recursively define a set, we need to provide two things:

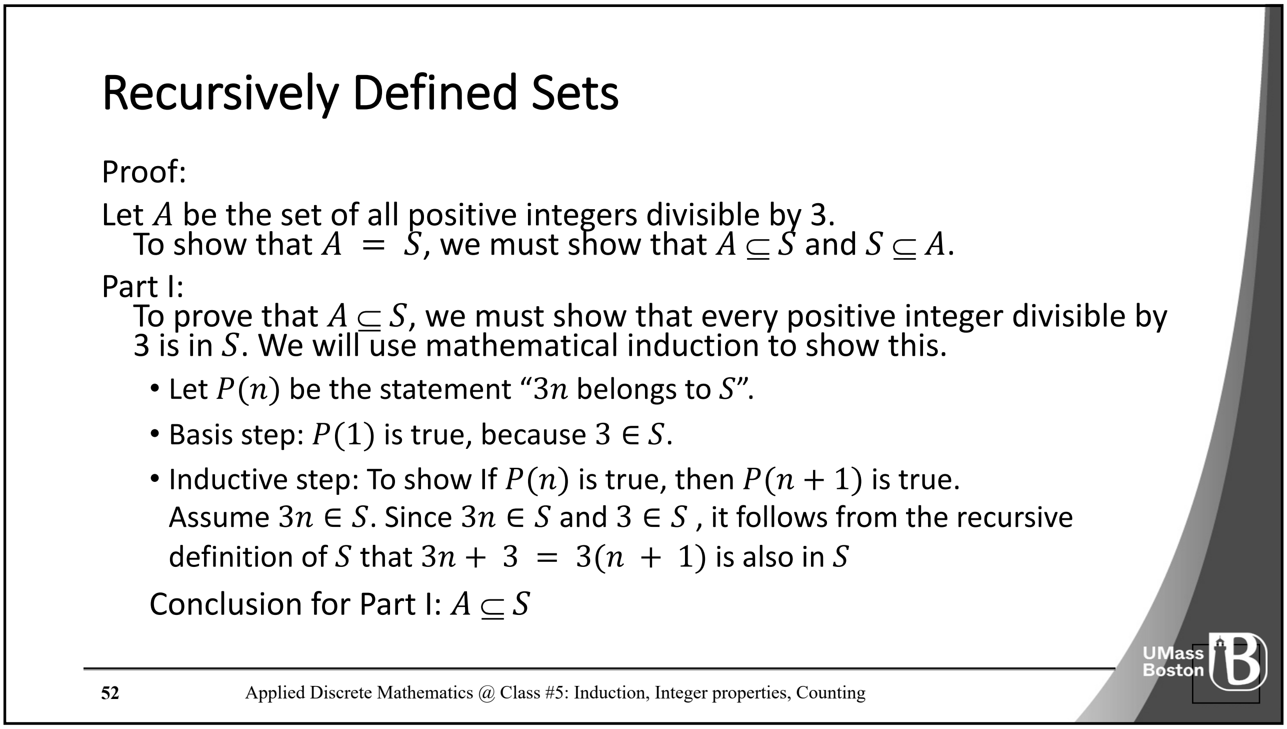
- an initial set of elements,
- rules for the construction of additional elements from elements in the set.

Example: Let S be recursively defined by:

- $3 \in S$
- $(x + y) \in S$ if $x \in S$ and $y \in S$

S is the set of positive integers divisible by 3.

51



Recursively Defined Sets

Proof:

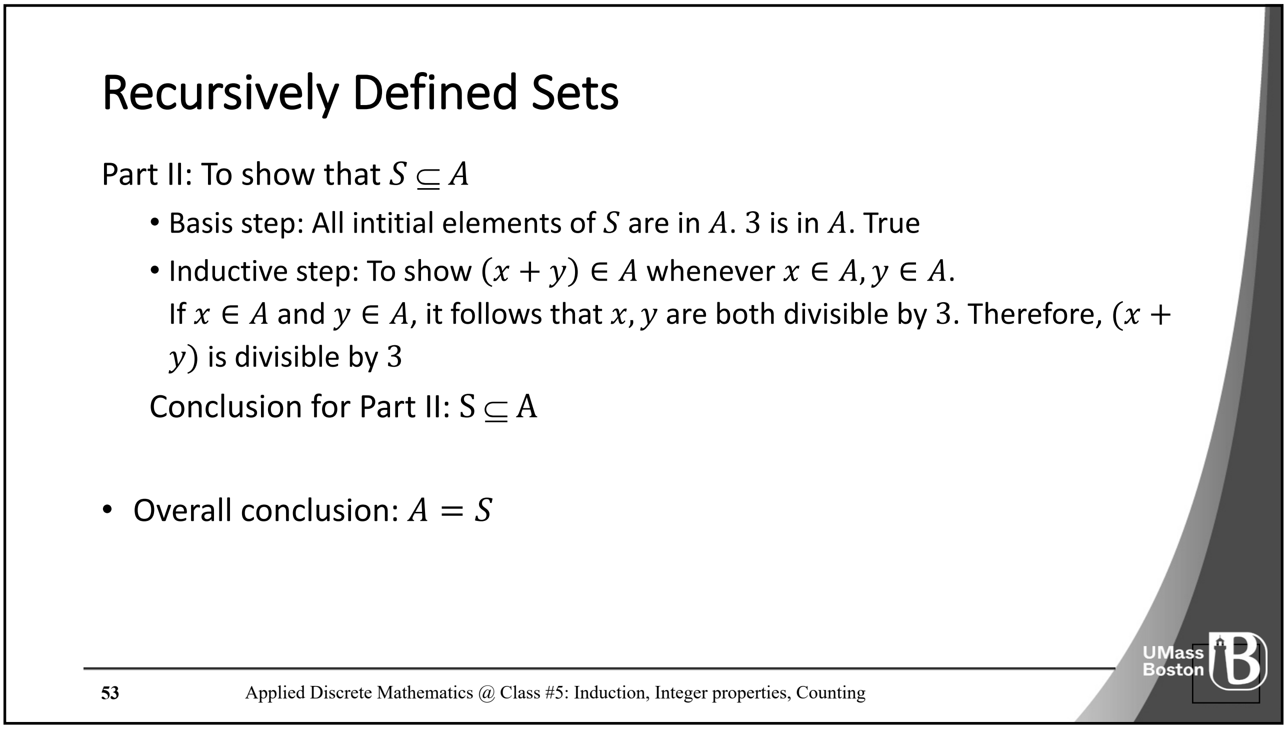
Let A be the set of all positive integers divisible by 3.
To show that $A = S$, we must show that $A \subseteq S$ and $S \subseteq A$.

Part I:
To prove that $A \subseteq S$, we must show that every positive integer divisible by 3 is in S. We will use mathematical induction to show this.

- Let $P(n)$ be the statement “$3n$ belongs to S”.
- Basis step: $P(1)$ is true, because $3 \in S$.
- Inductive step: To show If $P(n)$ is true, then $P(n+1)$ is true.
  Assume $3n \in S$. Since $3n \in S$ and $3 \in S$, it follows from the recursive definition of S that $3n + 3 = 3(n + 1)$ is also in S

Conclusion for Part I: $A \subseteq S$

52



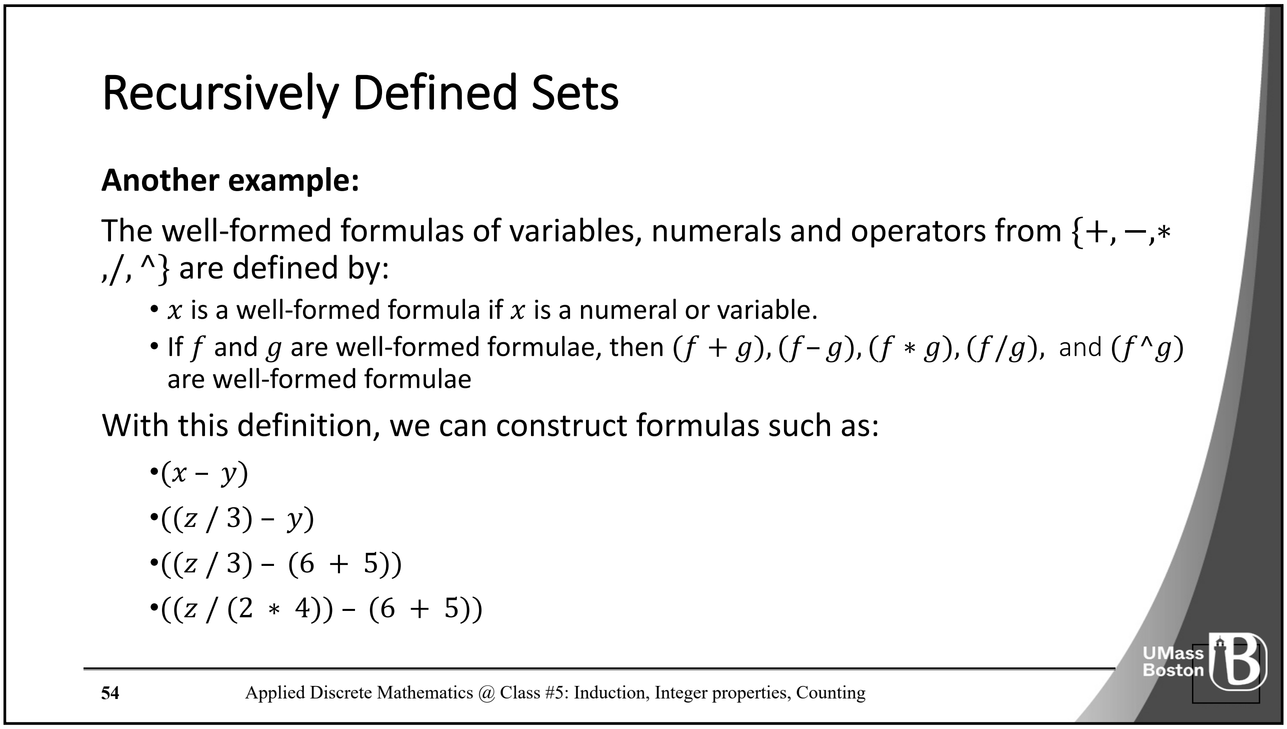
Recursively Defined Sets

Part II: To show that $S \subseteq A$

- Basis step: All intitial elements of S are in A. 3 is in A. True
- Inductive step: To show $(x + y) \in A$ whenever $x \in A, y \in A$.
  If $x \in A$ and $y \in A$, it follows that x, y are both divisible by 3. Therefore, $(x + y)$ is divisible by 3

  Conclusion for Part II: $\mathrm{S} \subseteq \mathrm{A}$

- Overall conclusion: $A = S$



Recursively Defined Sets

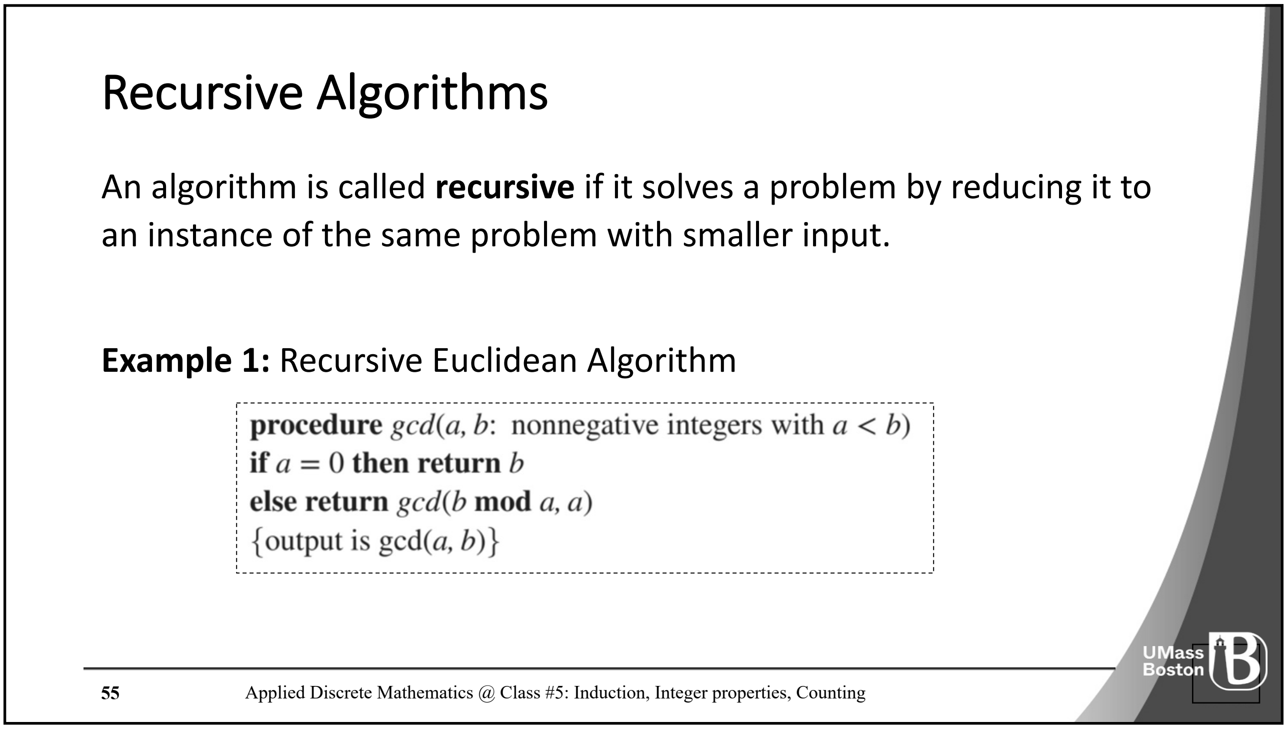
Another example:

The well-formed formulas of variables, numerals and operators from $\{+, -, *, /, \hat{}\}$ are defined by:

- x is a well-formed formula if x is a numeral or variable.
- If f and g are well-formed formulae, then $(f + g), (f - g), (f * g), (f / g),$ and $(f \hat{} g)$ are well-formed formulae

With this definition, we can construct formulas such as:

- $(x - y)$
- $((z / 3) - y)$
- $((z / 3) - (6 + 5))$
- $((z / (2 * 4)) - (6 + 5))$

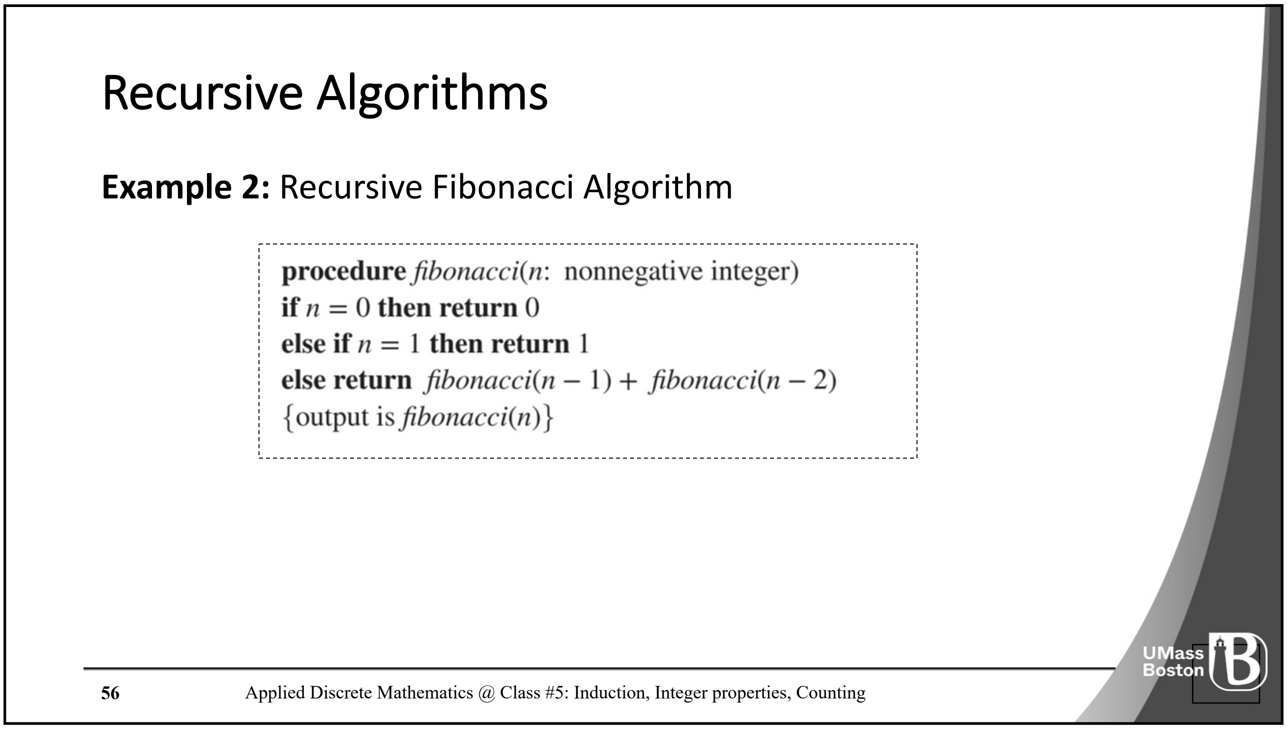


Recursive Algorithms

An algorithm is called **recursive** if it solves a problem by reducing it to an instance of the same problem with smaller input.

Example 1: Recursive Euclidean Algorithm

```
procedure gcd($a, b$: nonnegative integers with $a < b$)
if $a = 0$ then return $b$
else return gcd($b$ mod $a, a$)
{output is gcd($a, b$)}
```



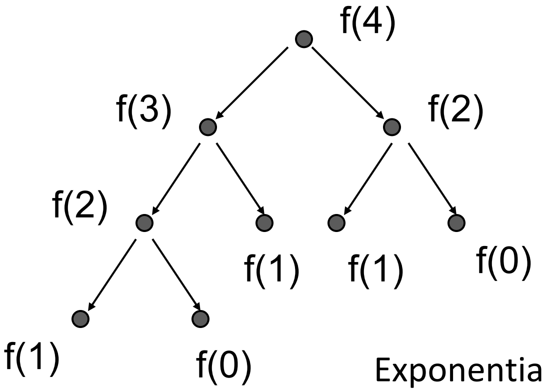
Recursive Algorithms

Example 2: Recursive Fibonacci Algorithm

```
procedure fibonacci($n$: nonnegative integer)
if $n = 0$ then return 0
else if $n = 1$ then return 1
else return fibonacci($n - 1$) + fibonacci($n - 2$)
{output is fibonacci($n$)}
```

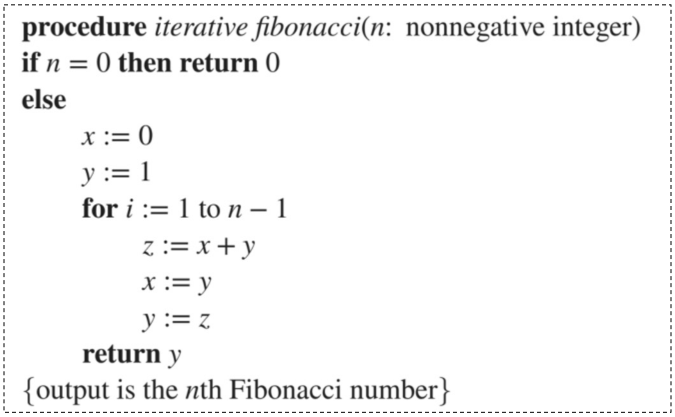
Recursive Algorithms

Recursive Fibonacci Evaluation:



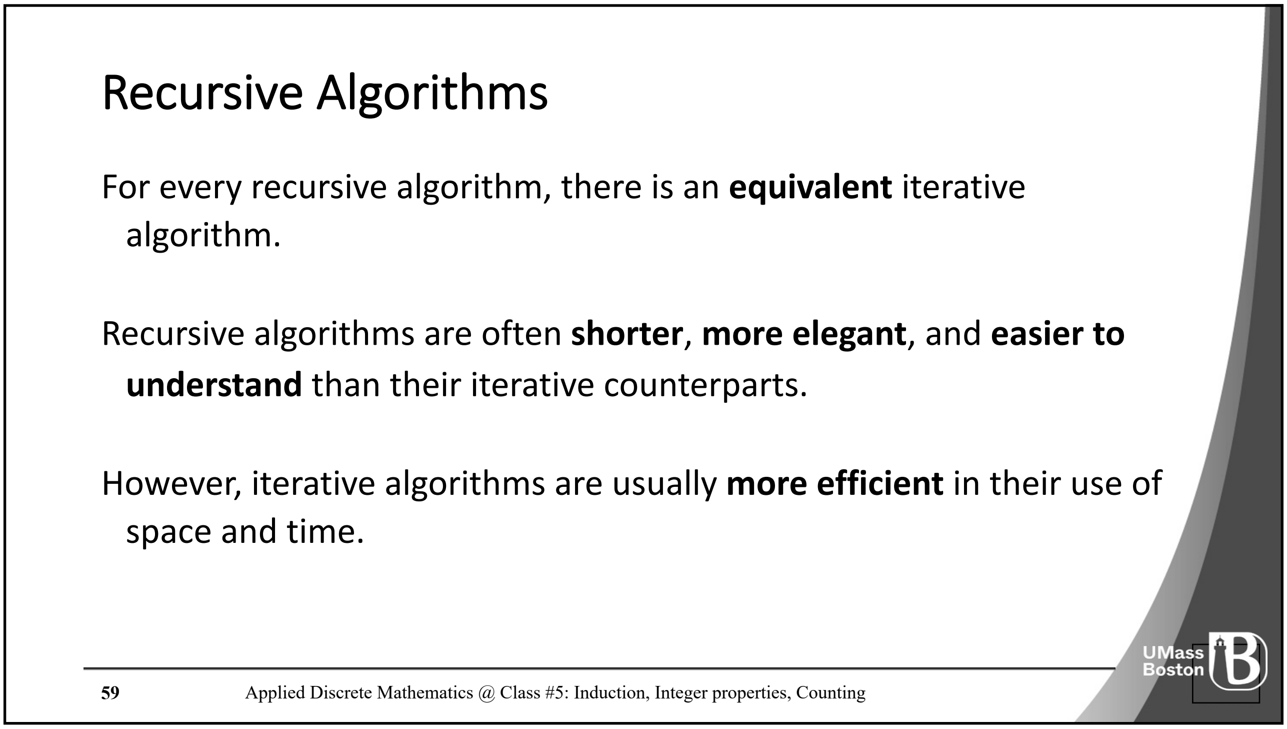
Exponential complexity!

Recursive Algorithms



```
procedure iterative fibonacci(n: nonnegative integer)
if n = 0 then return 0
else
    x := 0
    y := 1
    for i := 1 to n − 1
        z := x + y
        x := y
        y := z
    return y
{output is the nth Fibonacci number}
```

Linear complexity $O(n)$



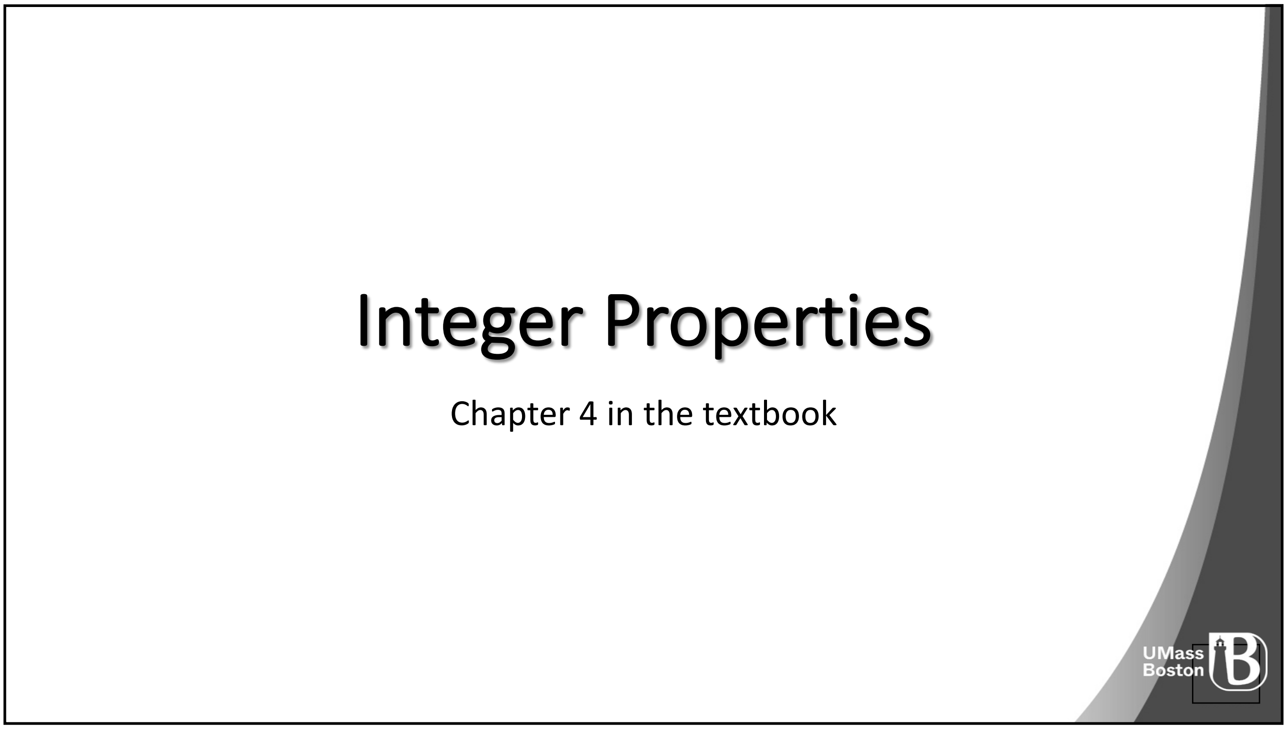
Recursive Algorithms

For every recursive algorithm, there is an **equivalent** iterative algorithm.

Recursive algorithms are often **shorter**, **more elegant**, and **easier to understand** than their iterative counterparts.

However, iterative algorithms are usually **more efficient** in their use of space and time.

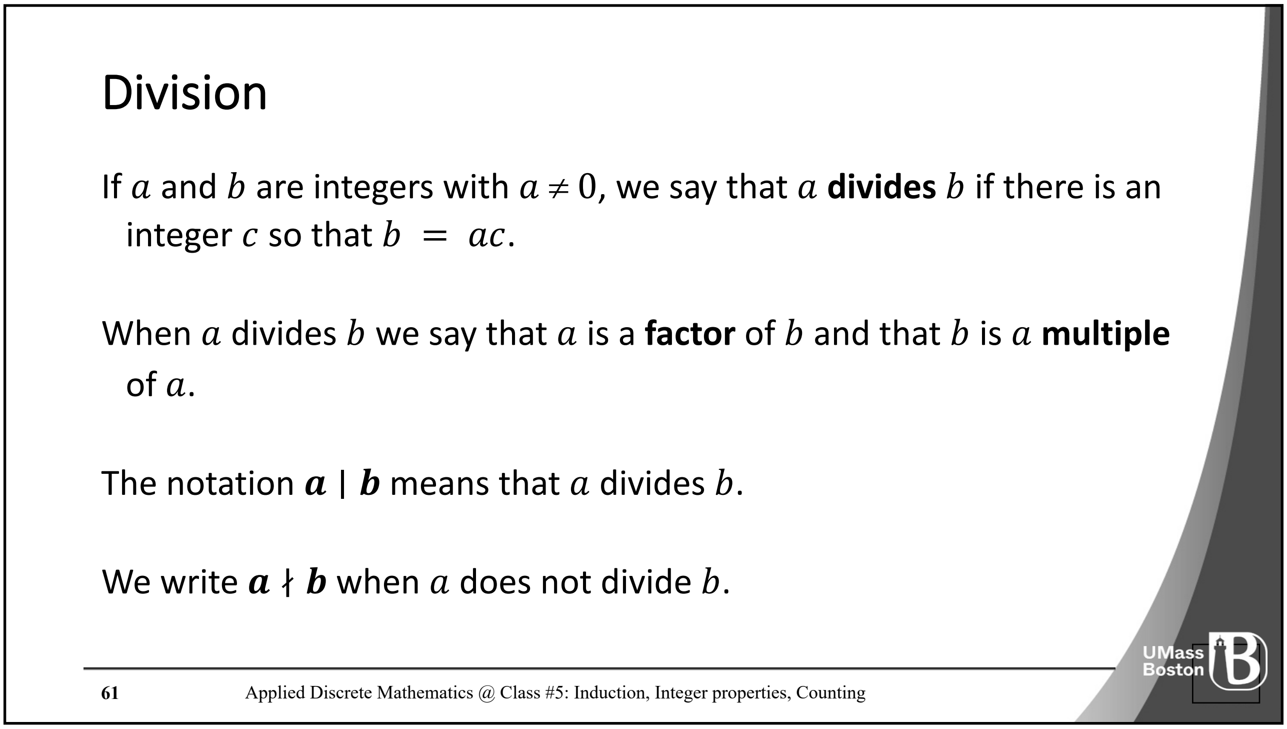
59



Integer Properties

Chapter 4 in the textbook

60



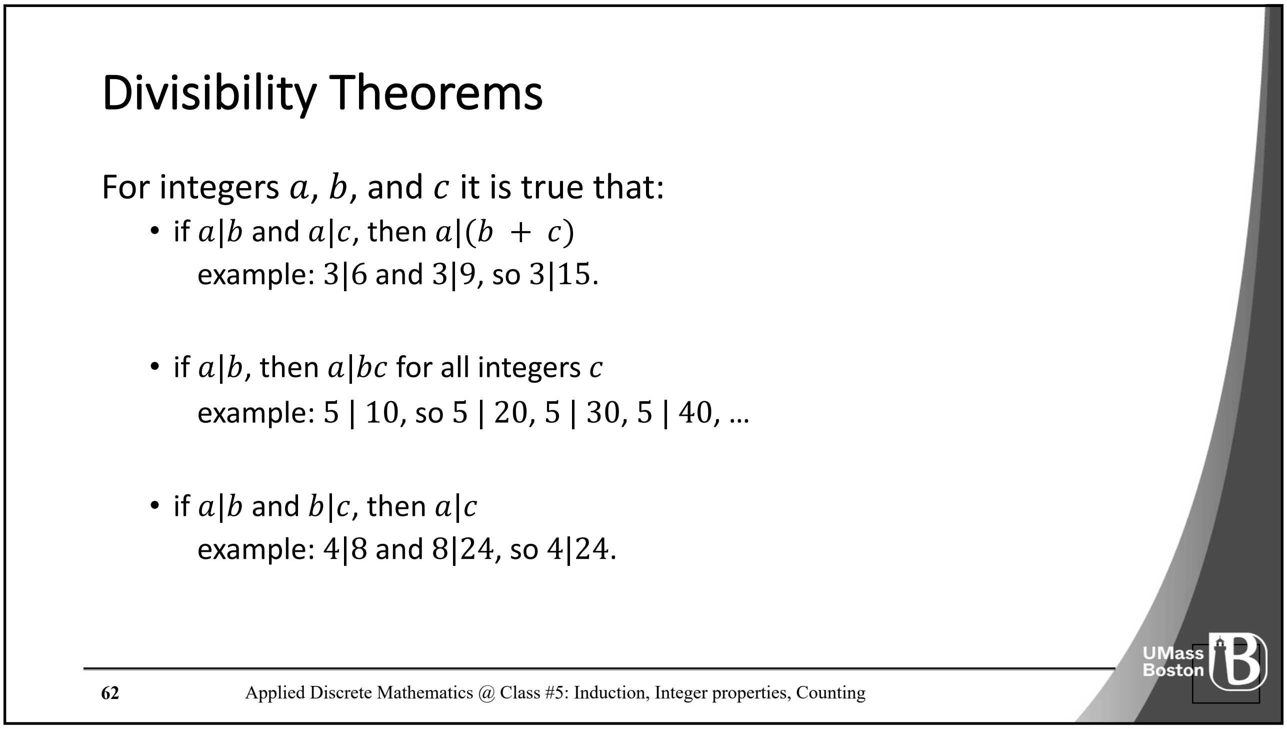
Division

If a and b are integers with $a \neq 0$, we say that a **divides** b if there is an integer c so that $b = ac$.

When a divides b we say that a is a **factor** of b and that b is a **multiple** of a.

The notation $\boldsymbol{a} \mid \boldsymbol{b}$ means that a divides b.

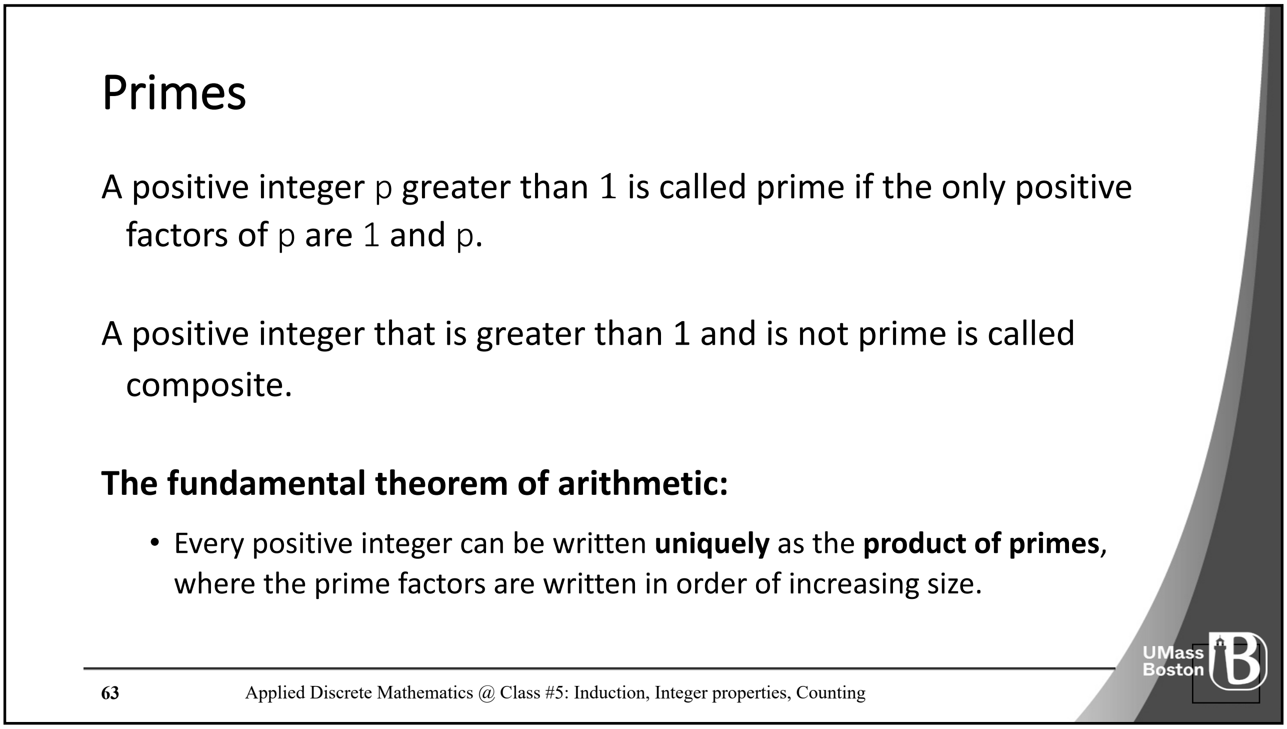
We write $\boldsymbol{a} \nmid \boldsymbol{b}$ when a does not divide b.



Divisibility Theorems

For integers a, b, and c it is true that:

- if $a|b$ and $a|c$, then $a|(b + c)$
  example: 3|6 and 3|9, so 3|15.
- if $a|b$, then $a|bc$ for all integers c
  example: 5 | 10, so 5 | 20, 5 | 30, 5 | 40, …
- if $a|b$ and $b|c$, then $a|c$
  example: 4|8 and 8|24, so 4|24.



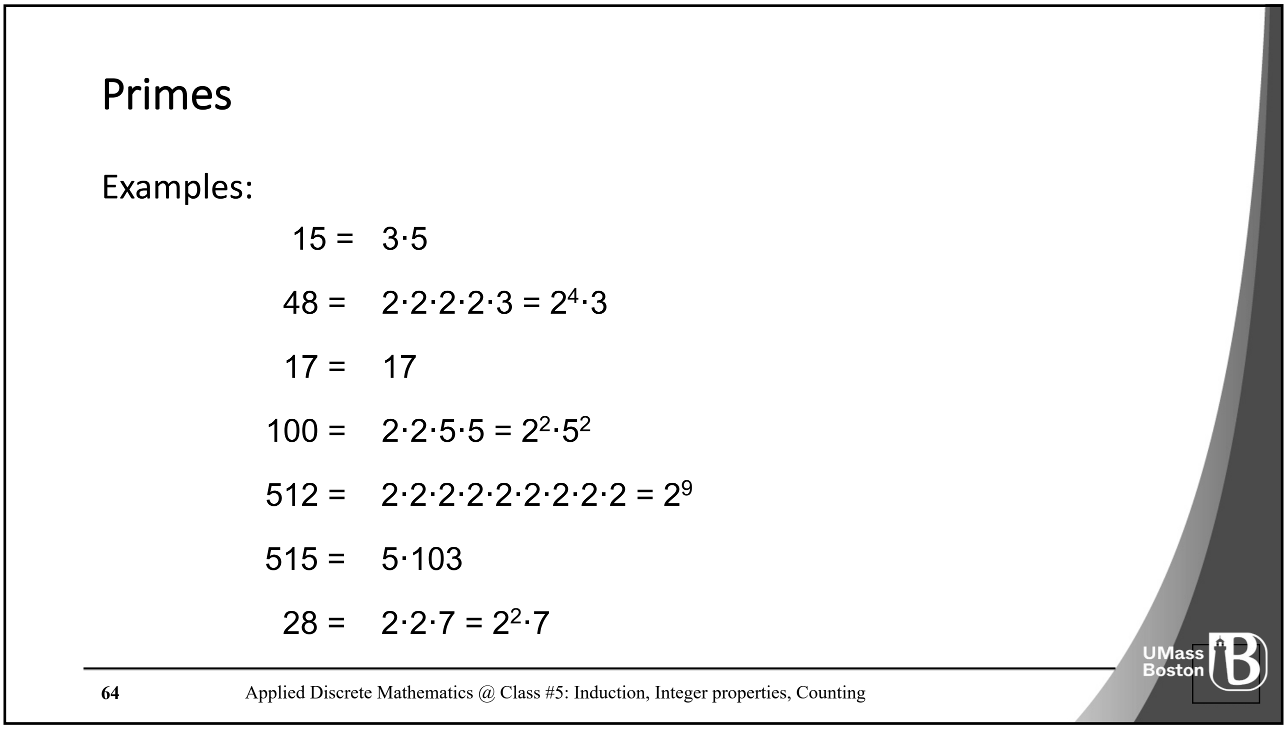
Primes

A positive integer p greater than 1 is called prime if the only positive factors of p are 1 and p.

A positive integer that is greater than 1 and is not prime is called composite.

The fundamental theorem of arithmetic:

- Every positive integer can be written **uniquely** as the **product of primes**, where the prime factors are written in order of increasing size.



Primes

Examples:

$15 = 3 \cdot 5$

$48 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 2^4 \cdot 3$

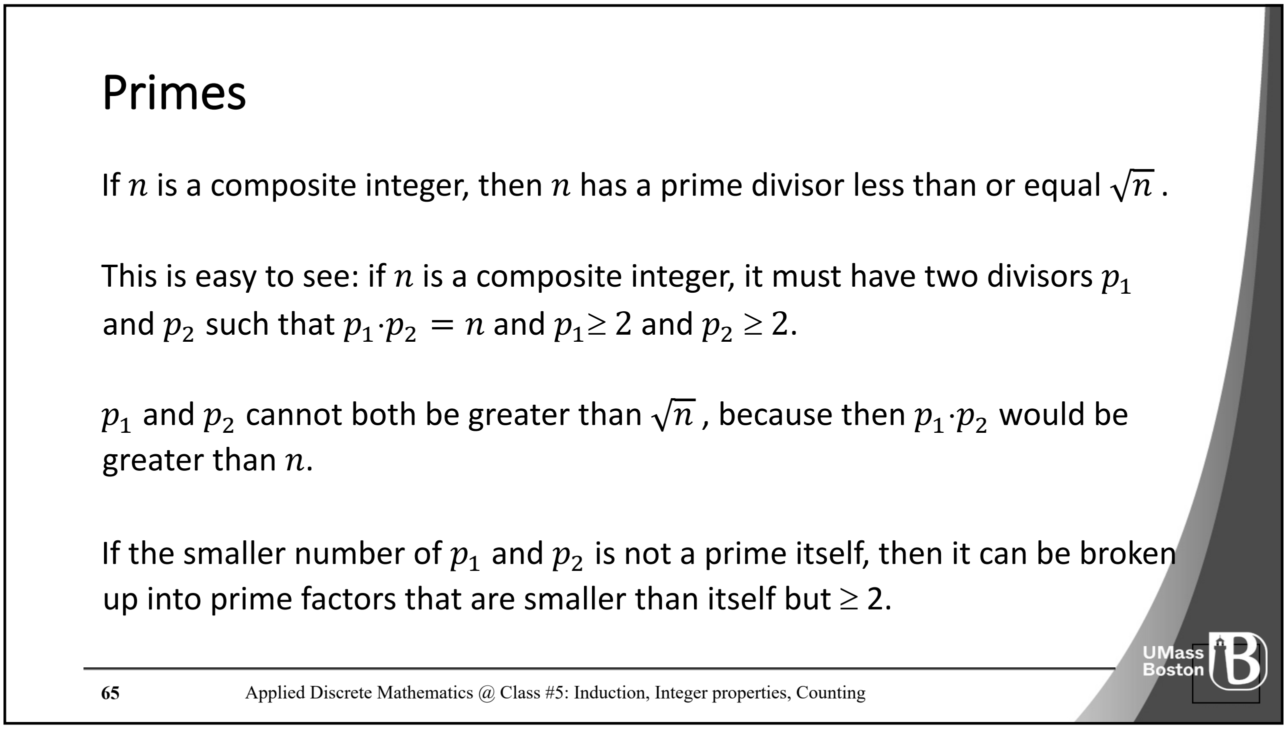
$17 = 17$

$100 = 2 \cdot 2 \cdot 5 \cdot 5 = 2^2 \cdot 5^2$

$512 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 2^9$

$515 = 5 \cdot 103$

$28 = 2 \cdot 2 \cdot 7 = 2^2 \cdot 7$



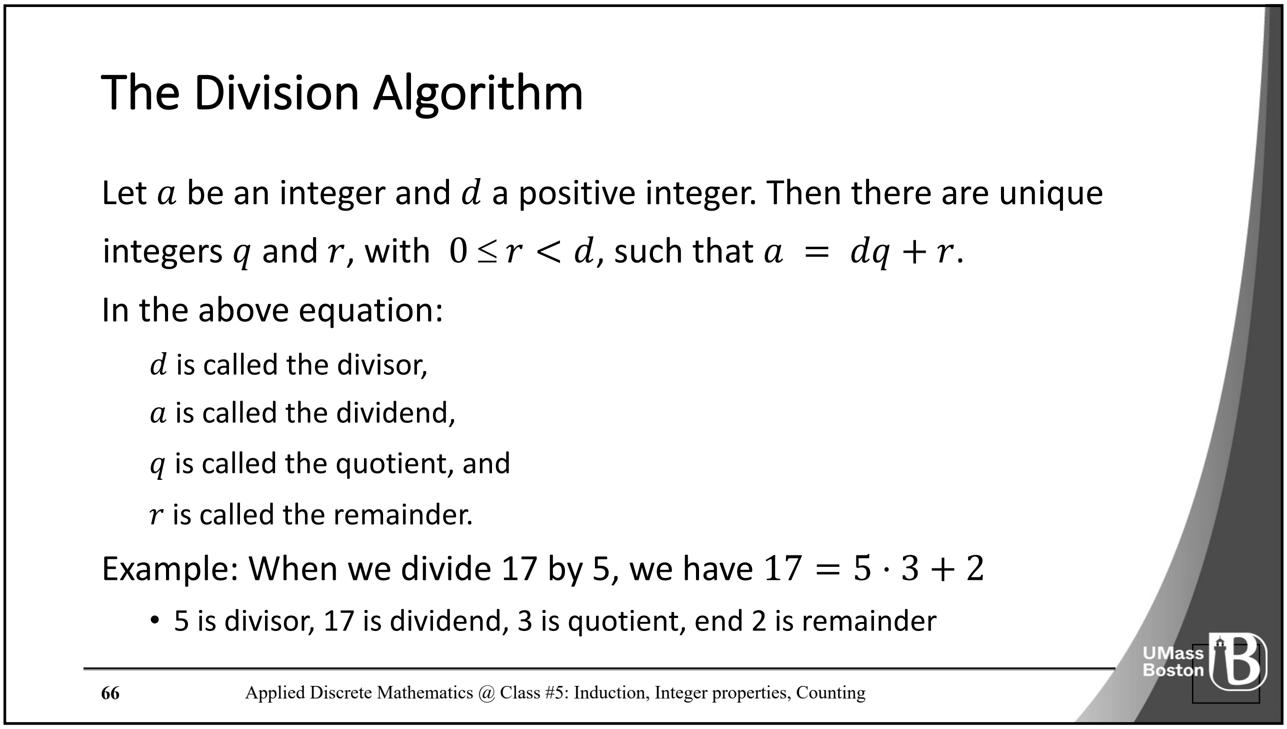
Primes

If n is a composite integer, then n has a prime divisor less than or equal $\sqrt{n}$.

This is easy to see: if n is a composite integer, it must have two divisors p_1 and p_2 such that $p_1 \cdot p_2 = n$ and $p_1 \geq 2$ and $p_2 \geq 2$.

p_1 and p_2 cannot both be greater than $\sqrt{n}$, because then $p_1 \cdot p_2$ would be greater than n.

If the smaller number of p_1 and p_2 is not a prime itself, then it can be broken up into prime factors that are smaller than itself but ≥ 2.



The Division Algorithm

Let a be an integer and d a positive integer. Then there are unique integers q and r, with $0 \leq r < d$, such that $a = dq + r$.

In the above equation:

d is called the divisor,

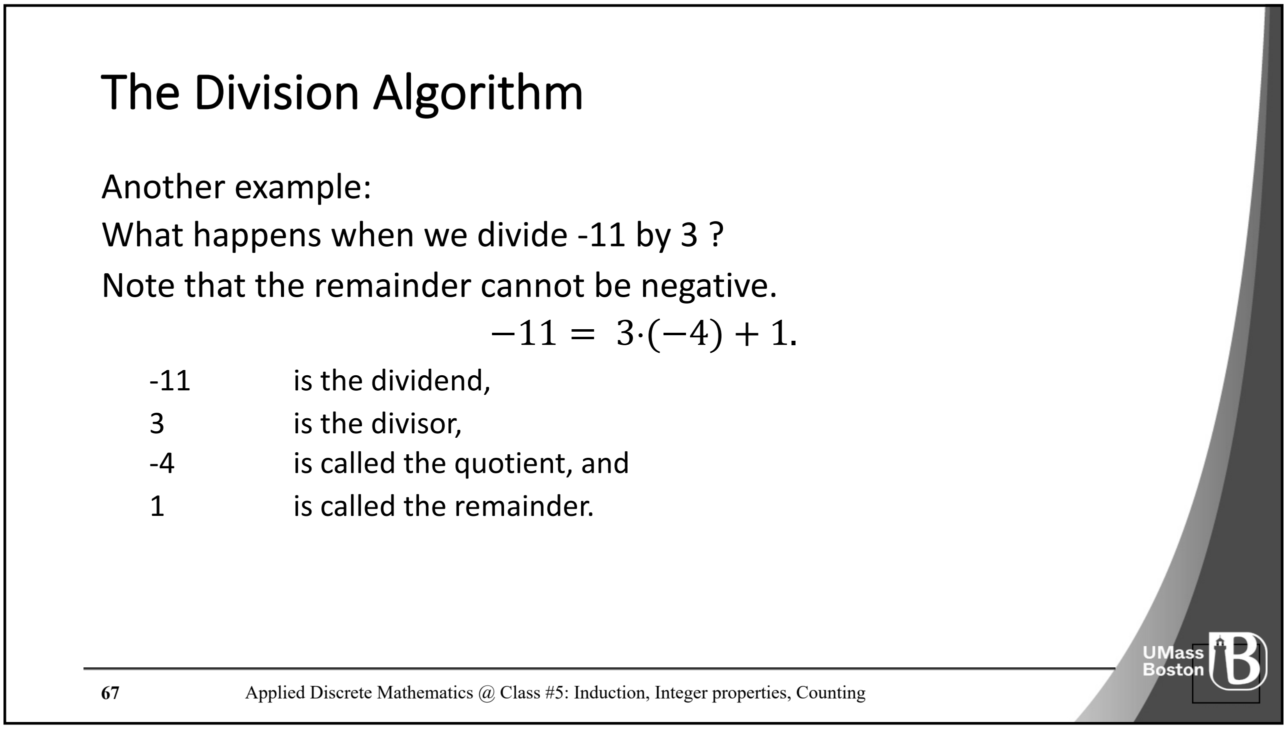
a is called the dividend,

q is called the quotient, and

r is called the remainder.

Example: When we divide 17 by 5, we have $17 = 5 \cdot 3 + 2$

- 5 is divisor, 17 is dividend, 3 is quotient, end 2 is remainder



The Division Algorithm

Another example:

What happens when we divide -11 by 3 ?

Note that the remainder cannot be negative.

$$-11 = 3\cdot(-4) + 1.$$

-11 is the dividend,

3 is the divisor,

-4 is called the quotient, and

1 is called the remainder.