Recurrence Relations

Section 8.2 - 8.3 in the textbook

Recurrence Relations

A **recurrence relation** for the sequence $\{a_n\}$ is an equation that expresses a_n is terms of one or more of the previous terms of the sequence, namely, $a_0, a_1, \dots, a_{n-1}$, for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer.

A sequence is called a **solution** of a recurrence relation if its terms satisfy the recurrence relation.

Recurrence Relations

In other words, a recurrence relation is like a recursively defined sequence, but **without specifying any initial values (initial conditions)**.

Therefore, the same recurrence relation can have (and usually has) **multiple solutions**.

If **both** the initial conditions and the recurrence relation are specified, then the sequence is **uniquely** determined.

Recurrence Relations

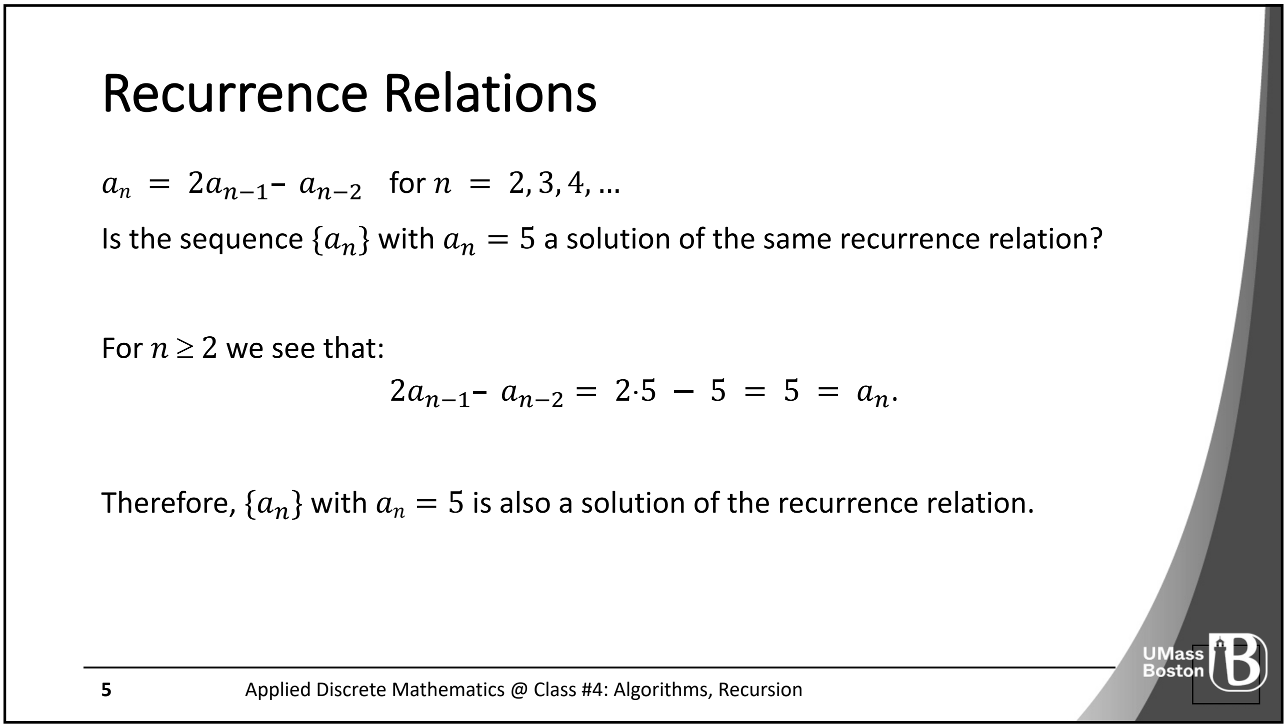
Example: Consider the recurrence relation $a_n = 2a_{n-1} - a_{n-2}$
for $n = 2, 3, 4, \ldots$

Is the sequence $\{a_n\}$ with $a_n = 3n$ a solution of this recurrence relation?

For $n \geq 2$ we see that

$$2a_{n-1} - a_{n-2} = 2(3(n-1)) - 3(n-2) = 3n = a_n.$$

Therefore, $\{a_n\}$ with $a_n = 3n$ is a solution of the recurrence relation.



Recurrence Relations

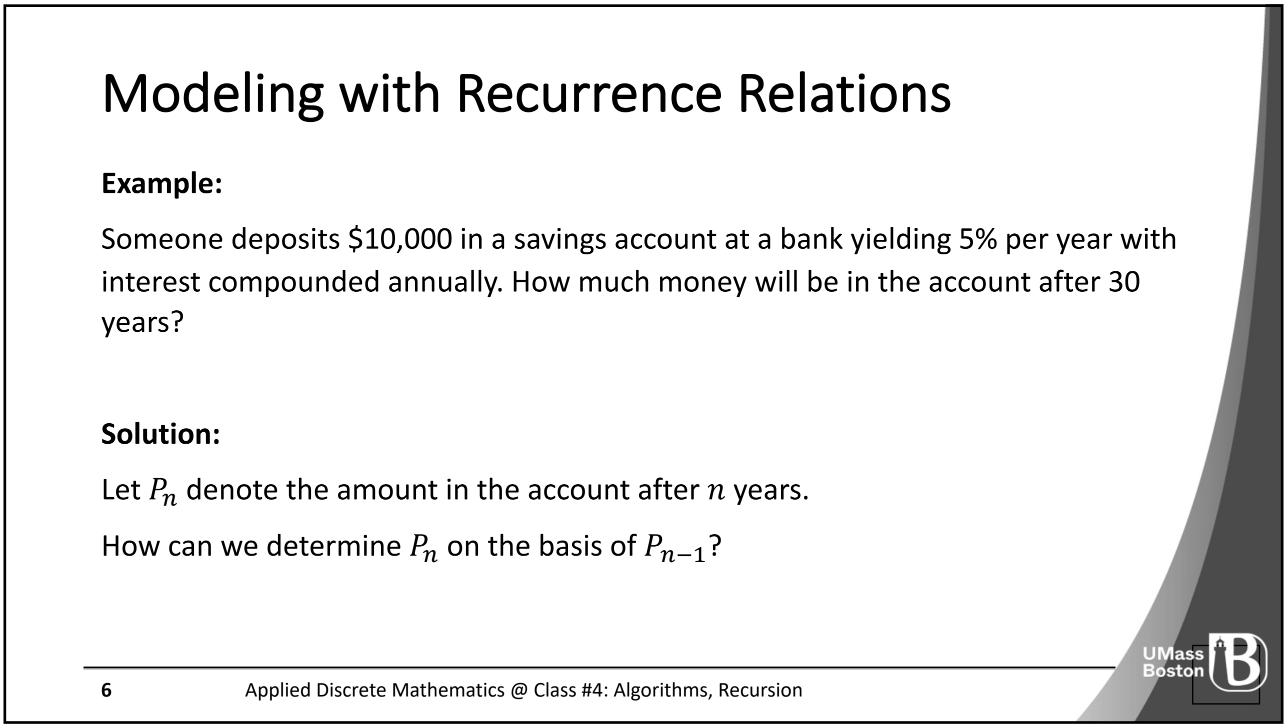
$a_n = 2a_{n-1} - a_{n-2}$ for $n = 2, 3, 4, \ldots$

Is the sequence $\{a_n\}$ with $a_n = 5$ a solution of the same recurrence relation?

For $n \geq 2$ we see that:

$$2a_{n-1} - a_{n-2} = 2 \cdot 5 - 5 = 5 = a_n.$$

Therefore, $\{a_n\}$ with $a_n = 5$ is also a solution of the recurrence relation.



Modeling with Recurrence Relations

Example:

Someone deposits $10,000 in a savings account at a bank yielding 5% per year with interest compounded annually. How much money will be in the account after 30 years?

Solution:

Let P_n denote the amount in the account after n years.

How can we determine P_n on the basis of P_{n-1}?

Modeling with Recurrence Relations

We can derive the following **recurrence relation**:

$$P_n = P_{n-1} + 0.05P_{n-1} = 1.05P_{n-1}.$$

The initial condition is $P_0 = 10{,}000$.

Then we have:

$P_1 = 1.05P_0$

$P_2 = 1.05P_1 = (1.05)^2P_0$

$P_3 = 1.05P_2 = (1.05)^3P_0$

…

$P_n = 1.05P_{n-1} = (1.05)^nP_0$

We now have a **formula** to calculate P_n for any natural number n and can avoid the iteration.

Modeling with Recurrence Relations

Let us use this formula to find P_{30} under the initial condition $P_0 = 10{,}000$:

$$P_{30} = (1.05)^{30}\times 10{,}000 = 43{,}219.42$$

After 30 years, the account contains $43,219.42.

Modeling with Recurrence Relations

Another example:

Let a_n denote the number of bit strings of length n that do not have two consecutive 0s ("valid strings"). Find a recurrence relation and give initial conditions for the sequence $\{a_n\}$.

Solution:

Idea: The number of valid strings equals the number of valid strings ending with a 0 plus the number of valid strings ending with a 1.

Modeling with Recurrence Relations

Let us assume that $n \geq 3$, so that the string contains at least 3 bits.

Let us further assume that we know the number a_{n-1} of valid strings of length $(n-1)$ and the number a_{n-2} of valid strings of length $(n-2)$.

Then how many valid strings of length n are there, if the string ends with a 1?

There are a_{n-1} such strings, namely the set of valid strings of length $(n - 1)$ with a 1 appended to them.

Note: Whenever we append a 1 to a valid string, that string remains valid.

Modeling with Recurrence Relations

Now we need to know: How many valid strings of length n are there, if the string ends with a **0**?

Valid strings of length n ending with a 0 **must have a 1 as their (n – 1)st bit** (otherwise they would end with 00 and would not be valid).

And what is the number of valid strings of length $(n - 1)$ that end with a 1?

We already know that there are a_{n-1} strings of length n that end with a 1.

Therefore, there are a_{n-2} strings of length $(n - 1)$ that end with a 1.

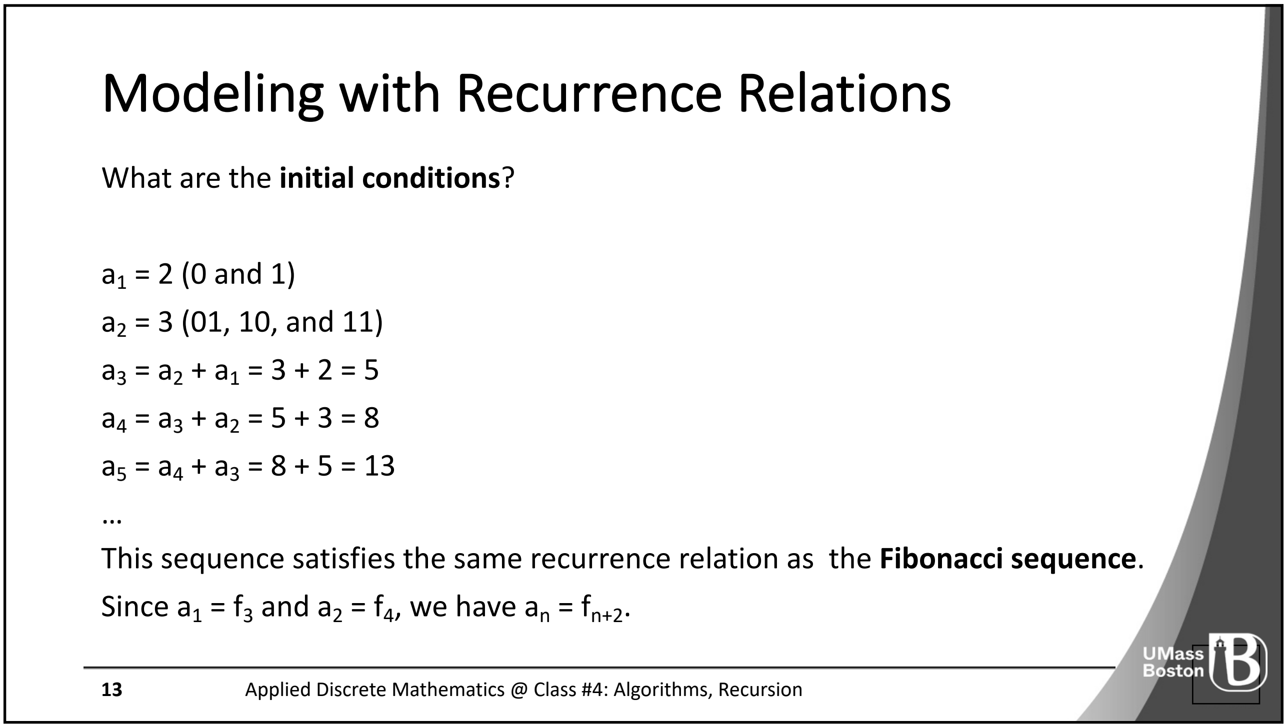
Modeling with Recurrence Relations

So there are a_{n-2} valid strings of length n that end with a 0 (all valid strings of length (n – 2) with 10 appended to them).

As we said before, the number of valid strings is the number of valid strings ending with a 0 plus the number of valid strings ending with a 1.

That gives us the following **recurrence relation**:

$$a_n = a_{n-1} + a_{n-2}$$



Modeling with Recurrence Relations

What are the **initial conditions**?

$a_1 = 2$ (0 and 1)

$a_2 = 3$ (01, 10, and 11)

$a_3 = a_2 + a_1 = 3 + 2 = 5$

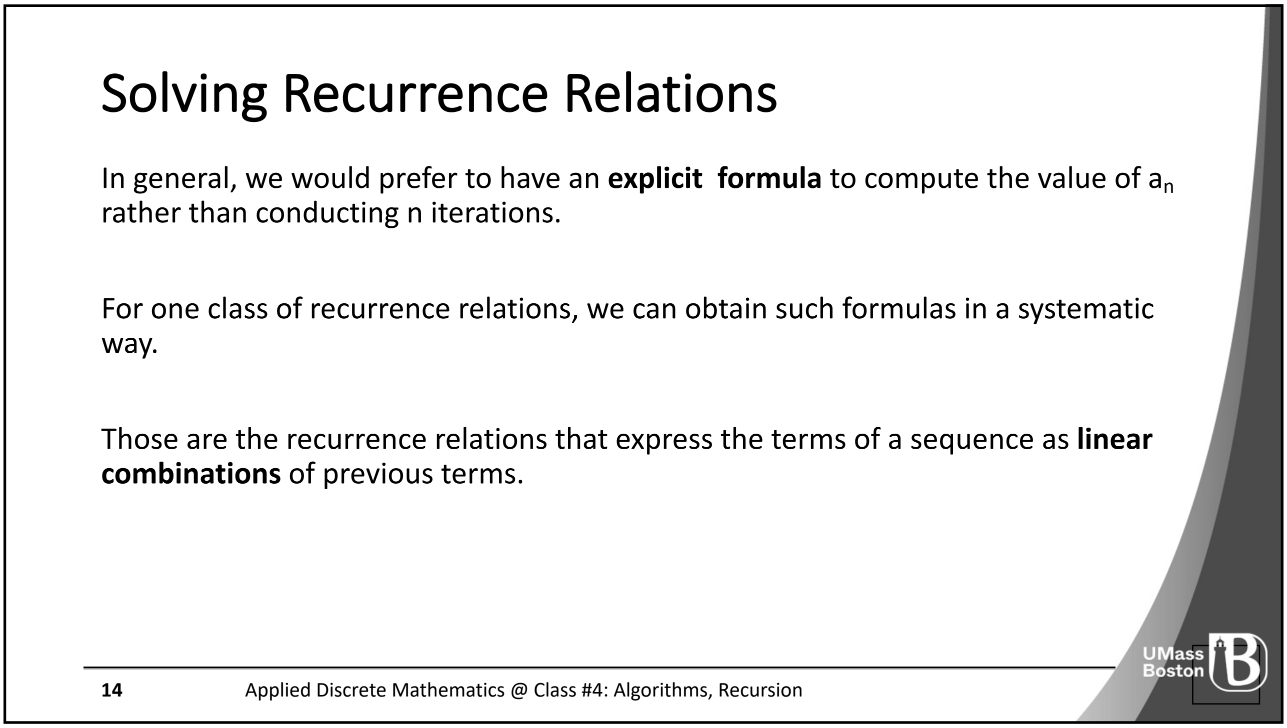
$a_4 = a_3 + a_2 = 5 + 3 = 8$

$a_5 = a_4 + a_3 = 8 + 5 = 13$

…

This sequence satisfies the same recurrence relation as the **Fibonacci sequence**.

Since $a_1 = f_3$ and $a_2 = f_4$, we have $a_n = f_{n+2}$.



Solving Recurrence Relations

In general, we would prefer to have an **explicit formula** to compute the value of a_n rather than conducting n iterations.

For one class of recurrence relations, we can obtain such formulas in a systematic way.

Those are the recurrence relations that express the terms of a sequence as **linear combinations** of previous terms.

Solving Recurrence Relations

Definition: A linear homogeneous recurrence relation of degree k with constant coefficients is a recurrence relation of the form:

$$a_n = c_1a_{n-1} + c_2a_{n-2} + \dots + c_ka_{n-k},$$

Where $c_1, c_2, \dots, c_k$ are real numbers, and $c_k \neq 0$.

A sequence satisfying such a recurrence relation is uniquely determined by the recurrence relation and the k initial conditions

$$a_0 = C_0,\ a_1 = C_1,\ a_2 = C_2,\ \dots,\ a_{k-1} = C_{k-1}.$$

Solving Recurrence Relations

Examples:

The recurrence relation $P_n = (1.05)P_{n-1}$
is a linear homogeneous recurrence relation of **degree one**.

The recurrence relation $f_n = f_{n-1} + f_{n-2}$
is a linear homogeneous recurrence relation of **degree two**.

The recurrence relation $a_n = a_{n-5}$
is a linear homogeneous recurrence relation of **degree five**.

Solving Recurrence Relations

Basically, when solving such recurrence relations, we try to find solutions of the form $\boldsymbol{a_n = r^n}$, where r is a constant, $a_n = r^n$ is a solution of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k}$ if and only if

$r^n = c_1 r^{n-1} + c_2 r^{n-2} + \cdots + c_k r^{n-k}$.

Divide this equation by r^{n-k} and subtract the right-hand side from the left:

$$r^k - c_1 r^{k-1} + c_2 r^{k-2} + \cdots + c_k = 0$$

This is called the **characteristic equation** of the recurrence relation.

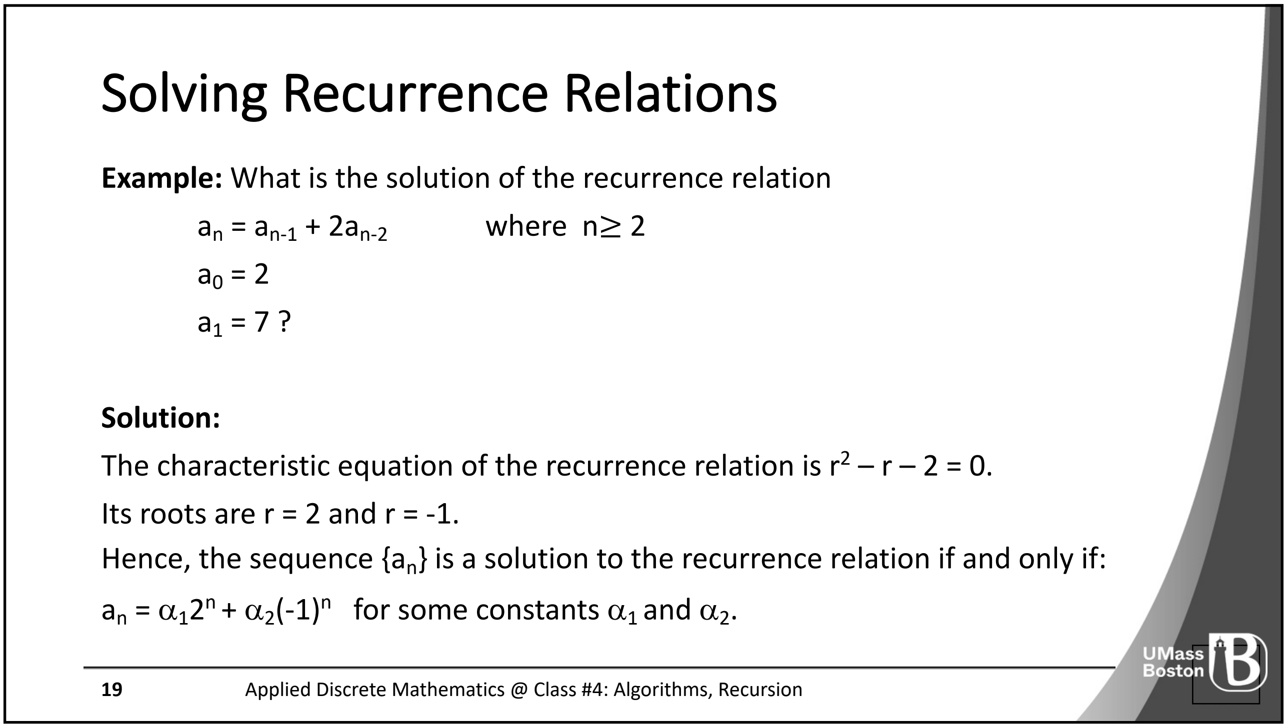
Solving Recurrence Relations

The solutions of this equation are called the **characteristic roots** of the recurrence relation.

Let us consider linear homogeneous recurrence relations of **degree two**.

Theorem: Let c_1 and c_2 be real numbers. Suppose that $r^2 - c_1 r - c_2 = 0$ has two distinct roots r_1 and r_2. Then the sequence $\{an\}$ is a solution of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2} \leftrightarrow a_n = \alpha_1 r_1^n + \alpha_2 r_2^n$ for $n = 0, 1, 2, \ldots$, where α_1 and α_2 are constants.

The proof is shown on pp. 542-543 of the textbook



Solving Recurrence Relations

Example: What is the solution of the recurrence relation

$a_n = a_{n-1} + 2a_{n-2}$ where $n \geq 2$

$a_0 = 2$

$a_1 = 7$?

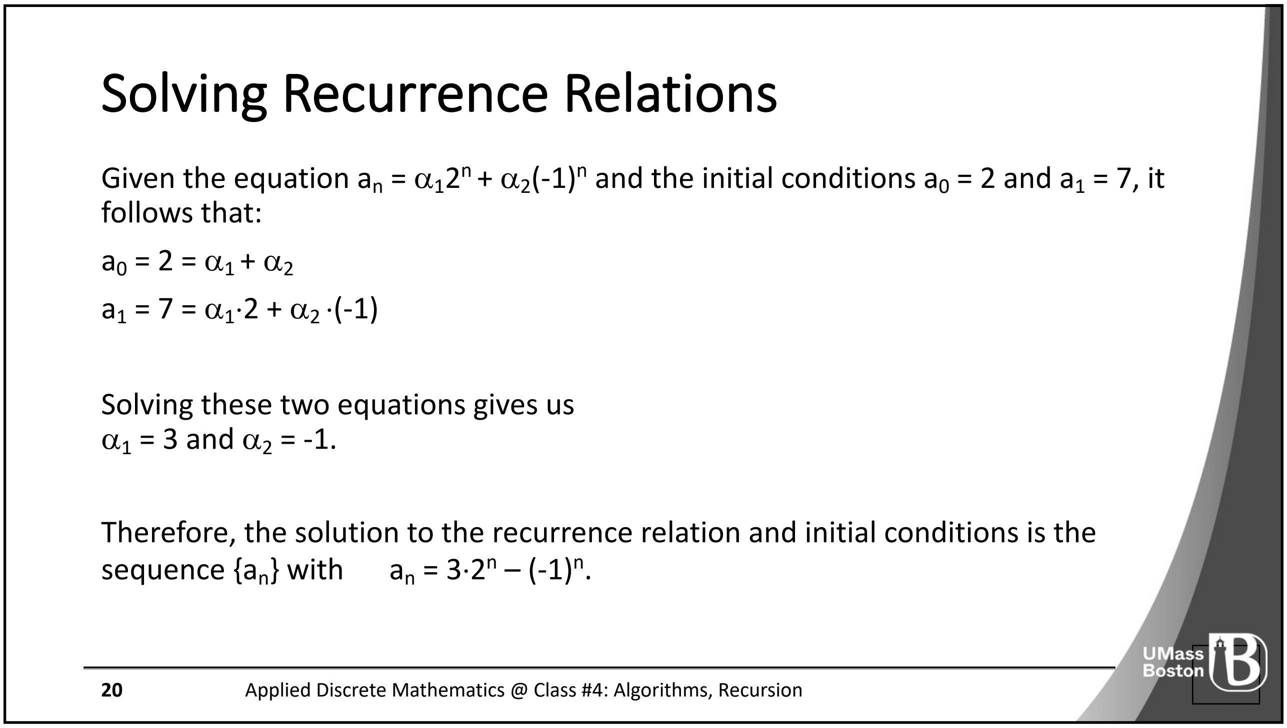
Solution:

The characteristic equation of the recurrence relation is $r^2 - r - 2 = 0$.

Its roots are $r = 2$ and $r = -1$.

Hence, the sequence $\{a_n\}$ is a solution to the recurrence relation if and only if:

$a_n = \alpha_1 2^n + \alpha_2(-1)^n$ for some constants α_1 and α_2.



Solving Recurrence Relations

Given the equation $a_n = \alpha_1 2^n + \alpha_2(-1)^n$ and the initial conditions $a_0 = 2$ and $a_1 = 7$, it follows that:

$a_0 = 2 = \alpha_1 + \alpha_2$

$a_1 = 7 = \alpha_1 \cdot 2 + \alpha_2 \cdot (-1)$

Solving these two equations gives us
$\alpha_1 = 3$ and $\alpha_2 = -1$.

Therefore, the solution to the recurrence relation and initial conditions is the sequence $\{a_n\}$ with $a_n = 3 \cdot 2^n - (-1)^n$.

Solving Recurrence Relations

Another Example: Give an explicit formula for the Fibonacci numbers.

Solution:

The Fibonacci numbers satisfy the recurrence relation $f_n = f_{n-1} + f_{n-2}$ with initial conditions $f_0 = 0$ and $f_1 = 1$.

The characteristic equation is $r^2 - r - 1 = 0$.

Its roots are $r_1 = \frac{1+\sqrt{5}}{2}, r_2 = \frac{1-\sqrt{5}}{2}$

Solving Recurrence Relations

Therefore, the Fibonacci numbers are given by:

$$f_n = \alpha_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + \alpha_2 \left(\frac{1-\sqrt{5}}{2}\right)^n \text{ for some } \alpha_1 \text{ and } \alpha_2$$

We can determine values for these constants so that the sequence meets the conditions $f_0 = 0$ and $f_1 = 1$:

$$f_0 = \alpha_1 + \alpha_2 = 0$$

$$f_1 = \alpha_1 \left(\frac{1+\sqrt{5}}{2}\right) + \alpha_2 \left(\frac{1-\sqrt{5}}{2}\right) = 1$$

Solving Recurrence Relations

The unique solution to this system of two equations and two variables is

$$\alpha_1 = \frac{1}{\sqrt{5}}, \alpha_2 = -\frac{1}{\sqrt{5}}$$

So finally, we obtained an explicit formula for the Fibonacci numbers:

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

Solving Recurrence Relations

But what happens if the characteristic equation has only one root?

How can we then match our equation with the initial conditions a_0 and a_1 ?

Theorem: Let c_1 and c_2 be real numbers with $c_2 \neq 0$. Suppose that $r^2 - c_1 r - c_2 = 0$ has only one roots r_0. The sequence $\{a_n\}$ is a solution of the recurrence relation $a_n = c_1 a_{n-1} + c_2 a_{n-2} \leftrightarrow a_n = \alpha_1 r_0^n + \alpha_2 n r_0^n$ for $n = 0, 1, 2, \ldots$, where α_1 and α_2 are constants

Solving Recurrence Relations

Example: What is the solution of the recurrence relation $a_n = 6a_{n-1} - 9a_{n-2}$ with $a_0 = 1$ and $a_1 = 6$?

Solution:

The only root of $r^2 - 6r + 9 = 0$ is $r_0 = 3$. Hence, the solution to the recurrence relation is $a_n = \alpha_1 3^n + \alpha_2 n3^n$ for some constants α_1 and α_2.

To match the initial condition, we need:

$$a_0 = 1 = \alpha_1$$
$$a_1 = 6 = \alpha_1 \cdot 3 + \alpha_2 \cdot 3$$

Solving these equations yields $\alpha_1 = 1$ and $\alpha_2 = 1$.

Consequently, the overall solution is given by $a_n = 3^n + n3^n$.

Divide-and-Conquer Relations

Some algorithms take a problem and **successively divide** it into one or more smaller problems until there is a **trivial solution** to them.

For example, the **binary search** algorithm recursively divides the input into two halves and eliminates the irrelevant half until only one relevant element remained.

This technique is called **“divide and conquer”.**

We can use **recurrence relations** to analyze the complexity of such algorithms.

Divide-and-Conquer Relations

Suppose that an algorithm divides a problem (input) of size **n** into **a** subproblems, where each subproblem is of size $\frac{n}{b}$. Assume that $g(n)$ operations are performed for such a division of a problem.

Then, if $\boldsymbol{f(n)}$ represents the number of operations required to solve the problem, it follows that f satisfies the recurrence relation

$$\boldsymbol{f(n) = af(\tfrac{n}{b}) + g(n)}.$$

This is called a **divide-and-conquer recurrence relation.**

Divide-and-Conquer Relations

Example: The binary search algorithm reduces the search for an element in a search sequence of size $\boldsymbol{n}$ to the binary search for this element in a search sequence of size $\boldsymbol{\frac{n}{2}}$ (if n is even).

Two comparisons are needed to perform this reduction.

Hence, if $\boldsymbol{f(n)}$ is the number of comparisons required to search for an element in a search sequence of size n, then $\boldsymbol{f(n) = f\left(\frac{n}{2}\right) + 2}$ if n is even.

Divide-and-Conquer Relations

Usually, we do not try to solve such divide-and conquer relations, but we use them to derive a **big-O estimate** for the complexity of an algorithm.

Theorem: Let f be an increasing function that satisfies the recurrence relation

$$f(n) = af(\frac{n}{b}) + cn^d$$

whenever $n = b^k$, where k is a positive integer, a, c, and d are real numbers with a ≥ 1, and b is an integer greater than 1. Then $f(n)$ is

$O(n^d)$,	if $a < b^d$,
$O(n^d \log n)$	if $a = b^d$,
$O(n^{\log_b a})$	if $a > b^d$

Divide-and-Conquer Relations

Example:

For binary search, we have: $f(n) = f(\frac{n}{2}) + 2$, so $a = 1, b = 2$, and $d = 0$ (d = 0 because here, $g(n)$ does not depend on n).

Consequently, $a = b^d$, and therefore, $f(n)$ is $O(n^d \log n) = O(\log n)$.

The binary search algorithm has logarithmic time complexity.

Algorithms

Chapter 3 in the textbook

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…

Algorithms

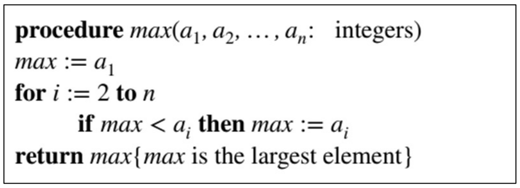
Properties of algorithms:

- **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.

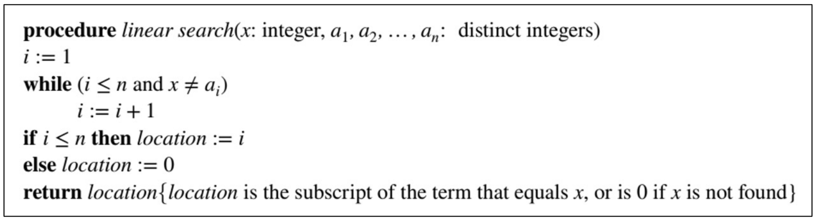
Example: an algorithm that finds the maximum element in a finite sequence



```
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}
```

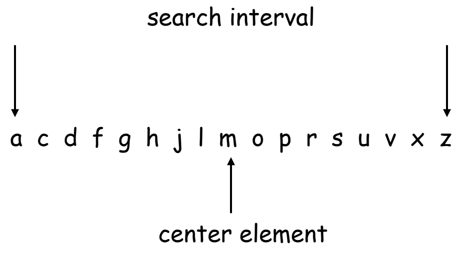
Algorithm Examples

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'



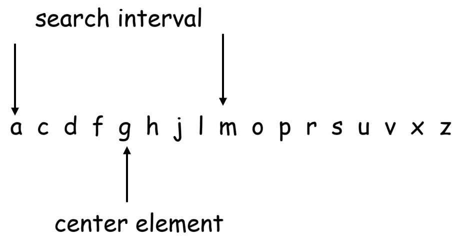
search interval

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

center element

Algorithm Examples

binary search for the letter 'j'



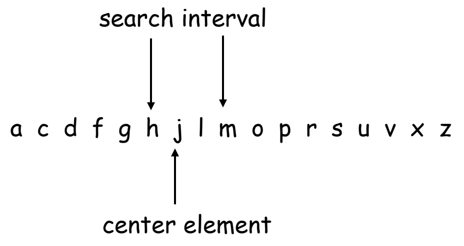
search interval

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'



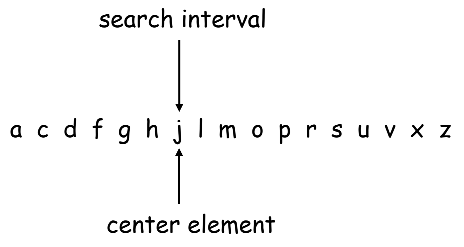
search interval

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

center element

Algorithm Examples

binary search for the letter 'j'



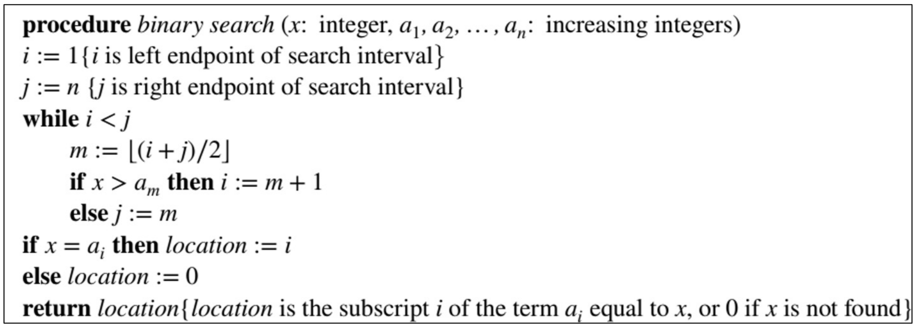
search interval

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?

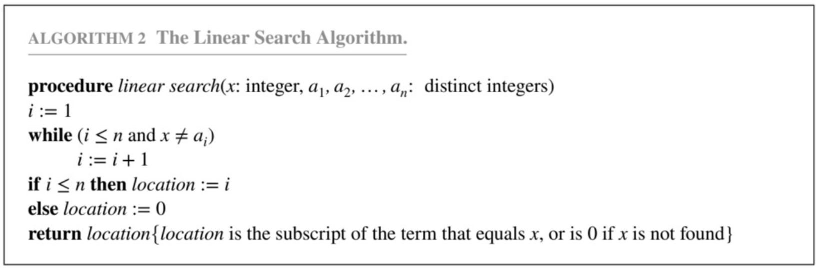
We will determine the **worst-case** number of comparisons as a function of the number n of terms in the sequence.

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:

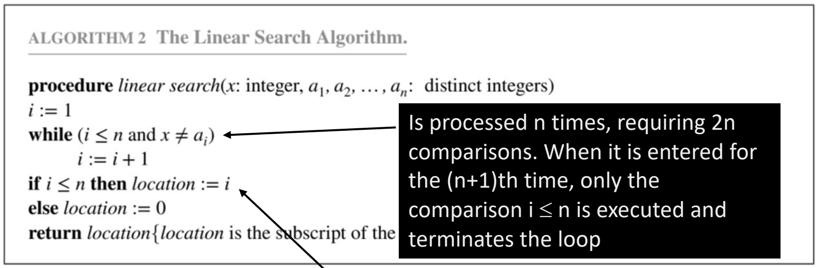


ALGORITHM 2 The Linear Search Algorithm.

```
procedure linear search($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \le n$ 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 term that equals $x$, or is 0 if $x$ is not found}
```

Algorithm Examples

Here is the linear search algorithm again:



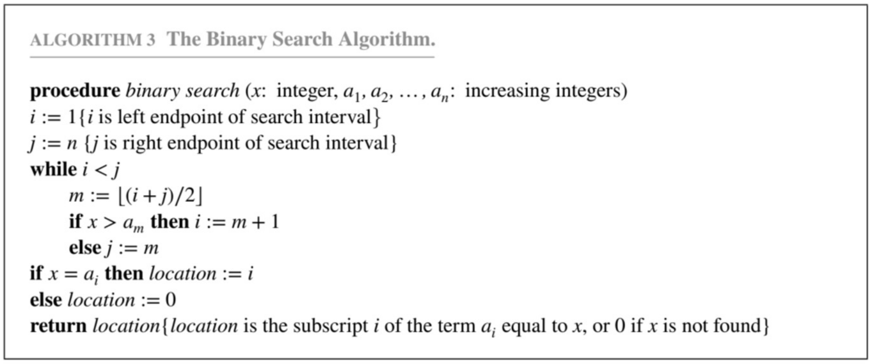
ALGORITHM 2 The Linear Search Algorithm.

```
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 comparison i ≤ n is executed and terminates the loop

Finally, this comparison is excuted, so in the worst-case, the time complexity is (2n+2)

Reminder: Binary Search Algorithm



ALGORITHM 3 The Binary Search Algorithm.

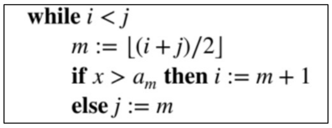
```
procedure binary search ($x$: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1${$i$ is left endpoint of search interval}
$j := n$ {$j$ is right endpoint of search interval}
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
if $x = a_i$ 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

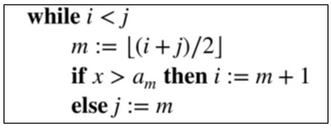
- Let us assume there are $n = 2^k$ elements in the list, which means that $k = \log n$.
- If n is not a power of 2, it can be considered part of a larger list, where $2^k < n < 2^{k+1}$.
- In the first cycle of the loop



```
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: $2k + 2 = 2\lceil \log n \rceil + 2$.

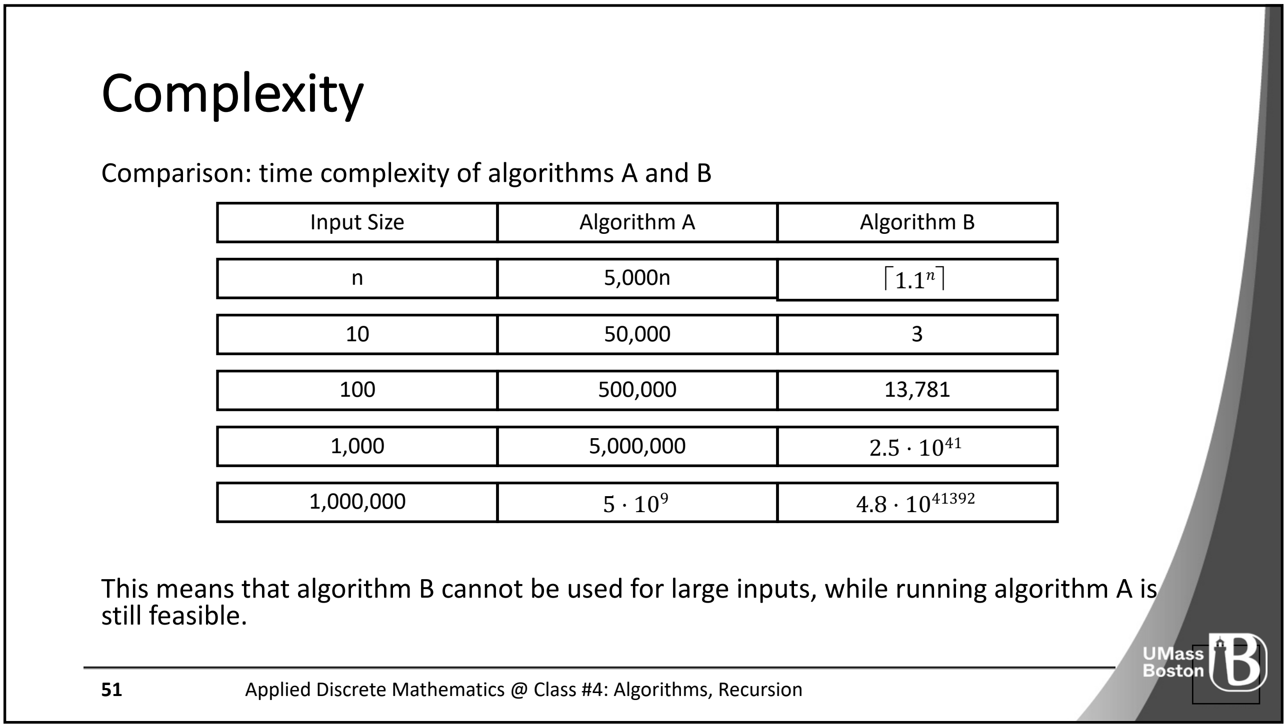
Complexity

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.

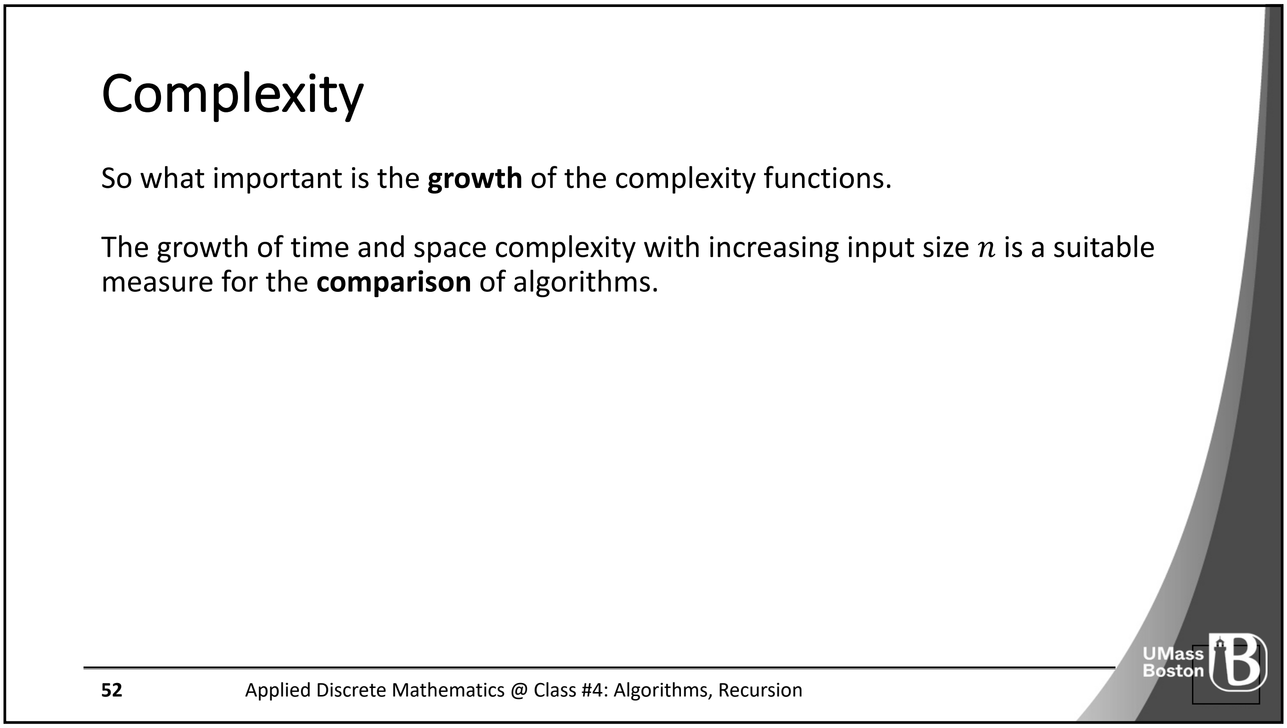


Complexity

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

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

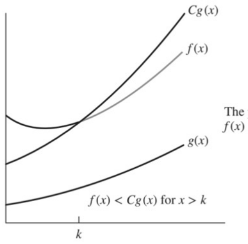
The Growth of Functions

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 $x \leq k$.

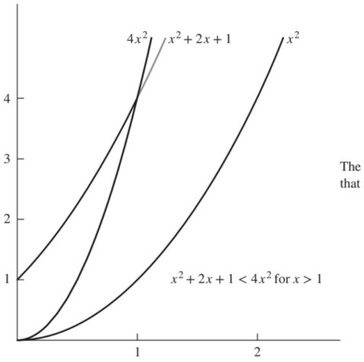
The Growth of Functions

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 $\boldsymbol{O}(g(x))$

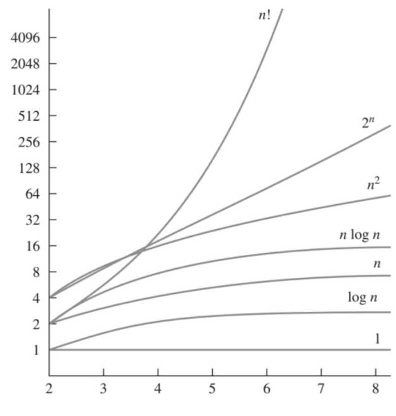
The Growth of Functions

"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($a_1, a_2, \ldots, a_n$: 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:

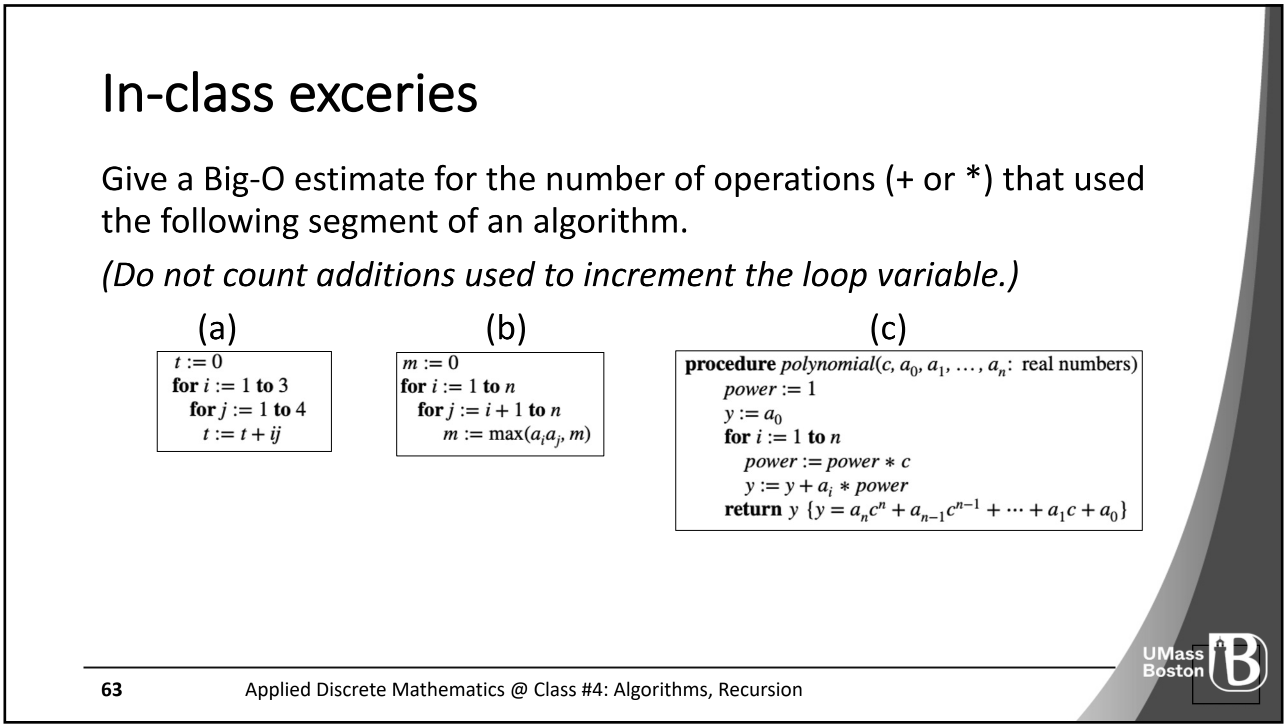
```
**procedure** max_diff($a_1, a_2, \ldots, a_n$: integers)
min := $a_1$
max := $a_1$
**for** i := 2 to n
    **if** $a_i$ < min **then** min := $a_i$
    **else** if $a_i$ > max **then** max := $a_i$

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.)

(a)

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

(b)

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

(c)

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