We will cover these parts of the book (8^{th} edition):

2.6
3.1.1-3.1.3 (up to page 207)
3.2.1-3.2.4
3.3.1, 3.3.2

Matrices

- A **matrix** is a rectangular array of numbers.
- A matrix with m rows and n columns is called an **m×n matrix.**
- **Example:** $A = \begin{bmatrix} -1 & 1 \\ 2.5 & -0.3 \\ 8 & 0 \end{bmatrix}$ is a 3x2 matrix

A matrix with the same number of rows and columns is called **square**.

Two matrices are **equal** if they have the same number of rows and columns and the corresponding entries in every position are equal.

Matrices

- A general description of an m×n matrix $A = [a_{ij}]$:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

$$\begin{bmatrix} a_{1j} \\ a_{2j} \\ \cdot \\ \cdot \\ \cdot \\ a_{mj} \end{bmatrix}$$

j-th column of A

$$\begin{bmatrix} a_{i1}, & a_{i2}, & \dots, & a_{in} \end{bmatrix}$$

i-th row of A

Matrix Addition

- Let $A = [a_{ij}]$ and $B = [b_{ij}]$ be $m \times n$ matrices.
- The sum of A and B, denoted by A+B, is the $m \times n$ matrix that has $a_{ij} + b_{ij}$ as its (i, j)th element.
- In other words, $A+B = [a_{ij} + b_{ij}]$.

- **Example:**

$$\begin{bmatrix} -2 & 1 \\ 4 & 8 \\ -3 & 0 \end{bmatrix} + \begin{bmatrix} 5 & 9 \\ -3 & 6 \\ -4 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 10 \\ 1 & 14 \\ -7 & 1 \end{bmatrix}$$

Matrix Multiplication

- Let A be an m×k matrix and B be a k×n matrix.
- The **product** of A and B, denoted by AB, is the m×n matrix with (i, j)th entry equal to the sum of the products of the corresponding elements from the i-th row of A and the j-th column of B.

- In other words, if AB = $[c_{ij}]$, then

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{ik}b_{kj} = \sum_{t=1}^{k} a_{it}b_{tj}$$

Matrix Multiplication

- A more intuitive description of calculating C = AB:

$$A = \begin{bmatrix} \boxed{3 \quad 0 \quad 1} \\ -2 & -1 & 4 \\ 0 & 0 & 5 \\ -1 & 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} \boxed{\begin{matrix} 2 \\ 0 \\ 3 \end{matrix}} & \begin{matrix} 1 \\ -1 \\ 4 \end{matrix} \end{bmatrix}$$

- Take the first column of B

- Turn it counterclockwise by 90 degrees and superimpose it on the first row of A

- Multiply corresponding entries in A and B and add the products: 3x2 + 0x0 + 1x3 = 9

- Enter the result in the upper-left corner of C

Matrix Multiplication

- Now superimpose the first column of B on the second, third, …, m-th row of A to obtain the entries in the first column of C (same order).

- Then repeat this procedure with the second, third, …, n-th column of B, to obtain to obtain the remaining columns in C (same order).

- After completing this algorithm, the new matrix C contains the product AB.

Matrix Multiplication

- Let us calculate the complete matrix C:

$$A = \begin{bmatrix} 3 & 0 & 1 \\ -2 & -1 & 4 \\ 0 & 0 & 5 \\ -1 & 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 2 & 1 \\ 0 & -1 \\ 3 & 4 \end{bmatrix}$$

$$C = \begin{bmatrix} 9 & 7 \\ 8 & 15 \\ 15 & 20 \\ -2 & -2 \end{bmatrix}$$

Identity Matrices

- The **identity matrix of order n** is the n×n matrix $I_n = [\delta_{ij}]$, where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$:

$$A = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ . & . & & . \\ . & . & & . \\ . & . & & . \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Multiplying an mxn matrix A by an identity matrix of appropriate size does not change this matrix:

$AI_n = I_mA = A$

Powers and Transposes of Matrices

- The **power function** can be defined for **square** matrices. If A is an $n \times n$ matrix, we have:

- $A^0 = I_n$,
- $A^r = AAA\ldots A$ (r times the letter A)

- The **transpose** of an $m \times n$ matrix $A = [a_{ij}]$, denoted by A^t, is the $n \times m$ matrix obtained by interchanging the rows and columns of A.

- In other words, if $A^t = [b_{ij}]$, then $b_{ij} = a_{ji}$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$.

Powers and Transposes of Matrices

- Example
:

$$A = \begin{bmatrix} 2 & 1 \\ 0 & -1 \\ 3 & 4 \end{bmatrix} \qquad A^t = \begin{bmatrix} 2 & 0 & 3 \\ 1 & -1 & 4 \end{bmatrix}$$

A square matrix A is called **symmetric** if $A = A^t$.
Thus $A = [a_{ij}]$ is symmetric if $a_{ij} = a_{ji}$ for all
$i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$.

$$A = \begin{bmatrix} 5 & 1 & 3 \\ 1 & 2 & -9 \\ 3 & -9 & 4 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 3 & 1 \\ 1 & 3 & 1 \end{bmatrix}$$

A is symmetric, B is not.

Zero-One Matrices

- A matrix with entries that are either 0 or 1 is called a **zero-one matrix**. Zero-one matrices are often used like a “table” to represent discrete structures.
- We can define Boolean operations on the entries in zero-one matrices:

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
0	0	0
0	1	1
1	0	1
1	1	1

Zero-One Matrices

- Let $A = [a_{ij}]$ and $B = [b_{ij}]$ be $m \times n$ zero-one matrices.

- Then the **join** of A and B is the zero-one matrix with (i, j)th entry $a_{ij} \vee b_{ij}$. The join of A and B is denoted by $A \vee B$.

- The **meet** of A and B is the zero-one matrix with (i, j)th entry $a_{ij} \wedge b_{ij}$. The meet of A and B is denoted by $A \wedge B$.

Zero-One Matrices

- **Example:** $$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 0 & 0 \end{bmatrix}$$

Join: $$A \vee B = \begin{bmatrix} 1 \vee 0 & 1 \vee 1 \\ 0 \vee 1 & 1 \vee 1 \\ 1 \vee 0 & 0 \vee 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Meet: $$A \wedge B = \begin{bmatrix} 1 \wedge 0 & 1 \wedge 1 \\ 0 \wedge 1 & 1 \wedge 1 \\ 1 \wedge 0 & 0 \wedge 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

Zero-One Matrices

- Let $A = [a_{ij}]$ be an $m \times k$ zero-one matrix and $B = [b_{ij}]$ be a $k \times n$ zero-one matrix.
- Then the **Boolean product** of A and B, denoted by $A \circ B$, is the $m \times n$ matrix with (i, j)th entry $[c_{ij}]$, where
- $c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2i}) \vee \ldots \vee (a_{ik} \wedge b_{kj})$.
- Note that the actual Boolean product symbol has a dot in its center.
- Basically, Boolean multiplication works like the multiplication of matrices, but with computing $\wedge$ instead of the product and $\vee$ instead of the sum.

Zero-One Matrices

- **Example:**

$$A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

$$A \circ B = \begin{bmatrix} (1 \wedge 0) \vee (0 \wedge 0) & (1 \wedge 1) \vee (0 \wedge 1) \\ (1 \wedge 0) \vee (1 \wedge 0) & (1 \wedge 1) \vee (1 \wedge 1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Zero-One Matrices

- Let A be a square zero-one matrix and r be a positive integer.

- The **r-th Boolean power** of A is the Boolean product of r factors of A. The r-th Boolean power of A is denoted by $A^{[r]}$.

- $A^{[0]} = I_n$,
- $A^{[r]} = A \circ A \circ \ldots \circ A$ (r times the letter A)

- Algorithms

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, …, 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; $a_1, a_2, \ldots, a_n$: 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 zero 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

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; $a_1, a_2, \ldots, a_n$: integers)
- i := 1 {i is left endpoint of search interval}
- j := n {j is right endpoint of search interval}
- **while** (i < j)
- **begin**
- m := $\lfloor (i + j)/2 \rfloor$
- **if** x > a_m **then** i := m + 1
- **else** j := m
- **end**
- **if** x = a_i **then** location := i
- **else** location := 0
- **Return** location{location is the subscript of the term that equals x, or is zero if x is not found}

UMASS BOSTON

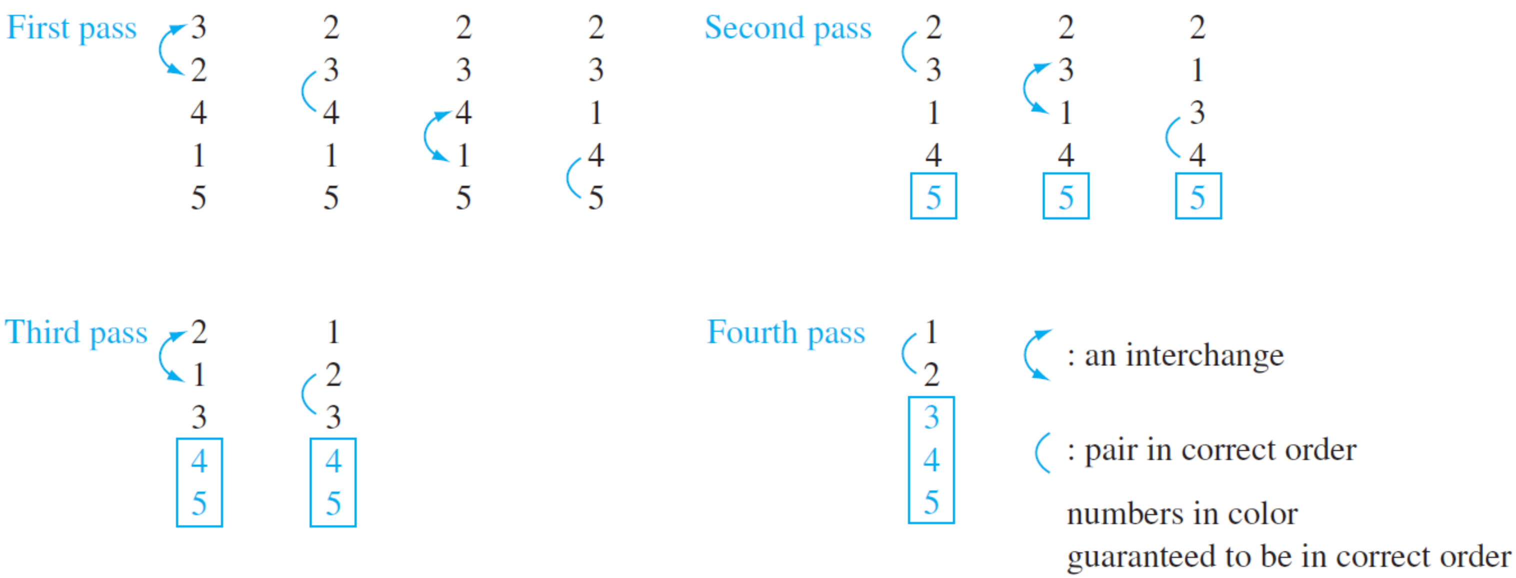
Algorithm Examples

- **procedure** bubblesort(a_1, a_2, …, a_n: real numbers, $n \geq 2$)
- **for** i := 1 to n-1
  - **for** j := 1 to n-1
    - **if** $a_j > a_{j+1}$ then interchange a_j and a_{j+1}
- {a_1, a_2, …, a_n is in increasing order}

Algorithm Examples

- Bubble sort:

- It puts a list into increasing order by successively comparing adjacent elements, interchanging them if they are in the wrong order. To carry out the bubble sort, we perform the basic operation, that is, interchanging a larger element with a smaller one following it, starting at the beginning of the list, for a full pass. We iterate this procedure until the sort is complete.



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 O(g(x)) if there are constants C and k such that

- $|f(x)| \leq C|g(x)|$

- whenever $x > k$.

- This is read as "f(x) is big-oh of g(x)"

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
- $f(x) \leq C \cdot g(x)$ whenever $x > k$.
- If we want to show that $f(x)$ is $O(g(x))$, we only need to find **one** pair (C, k) (which is never unique).

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
- $f(x) \leq C \cdot g(x)$ whenever $x > k$,
- because **C does not grow with 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 $O(x^2)$.
- 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 Growth of Functions

- Question: If f(x) is $O(x^2)$, is it also $O(x^3)$?

- **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 O(g(x)).

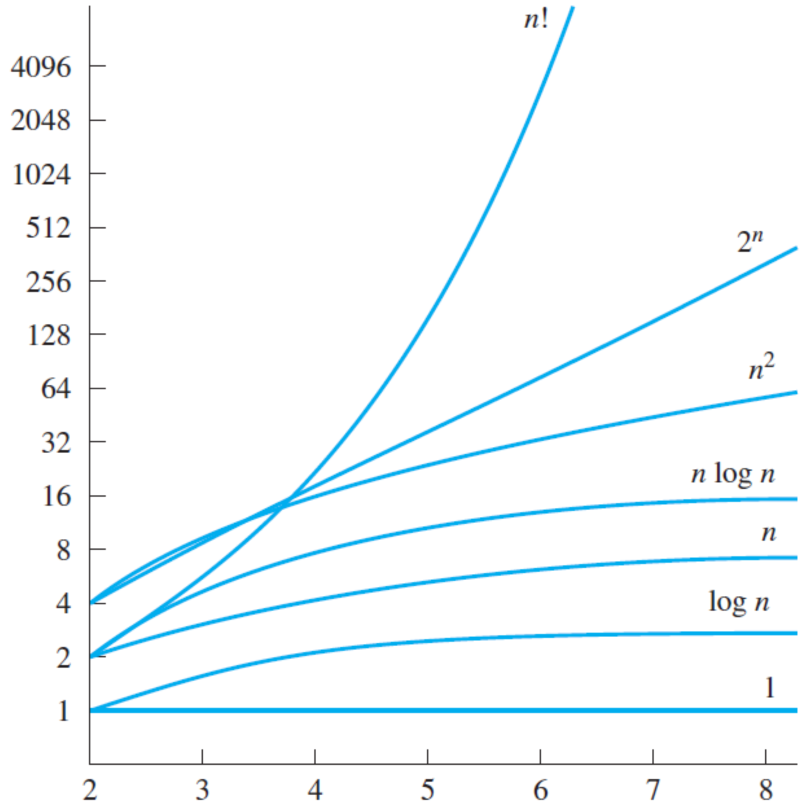
- Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_{n-1}, a_n$ are real numbers. Then $f(x)$ is $O(x^n)$

The Growth of Functions

- “Popular” functions g(n) are
- $n \log n$, 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 Combinations of Functions

- Suppose that $f_1(x)$ is $O\left(g_1(x)\right)$ and that $f_2(x)$ is $O\left(g_2(x)\right)$. Then $(f_1+f_2)(x)$ is $O(g(x))$, where $g(x) = (\max(|g_1(x)|, |g_2(x)|))$ for all x.

- Suppose that $f_1(x)$ is $O\left(g_1(x)\right)$ and that $f_2(x)$ is $O\left(g_2(x)\right)$. Then $(f_1 f_2)(x)$ is $O\left(g_1(x)g_2(x)\right)$.

Complexity of Algorithms

- 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 **time complexity** and **space complexity**, respectively.

Time Complexity

- The time complexity of an algorithm can be expressed in terms of the number of operations used by the algorithm when the input has a particular size.
- Time complexity is described in terms of the number of operations required instead of actual computer time because of the difference in time needed for different computers to perform basic operations.

Time 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.

- By the worst-case performance of an algorithm, we mean the largest number of operations needed to solve the given problem.

- 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:
- **procedure** linear_search(x: integer; a_1, a_2, …, a_n: integers)
- i := 1
- **while** (i ≤ n and x ≠ a_i)
-         i := i + 1
- **if** i ≤ n **then** location := i
- **else** location := 0
- **Return** location{location is the subscript of the term that equals x, or is zero if x is not found}

Complexity

- For n elements, the loop
- **while** $(i \leq n$ and $x \neq a_i)$
  $i := i + 1$
- is processed n times, requiring 2n comparisons.
- When it is entered for the (n+1)th time, only the comparison $i \leq n$ is executed and terminates the loop.
- Finally, the comparison

if $i \leq n$ **then** location := i

is executed, so all in all we have a worst-case time complexity of 2n + 2.

Reminder: Binary Search Algorithm

- **procedure** binary_search(x: integer; $a_1, a_2, \ldots, a_n$: integers)
- i := 1 {i is left endpoint of search interval}
- j := n {j is right endpoint of search interval}
- **while** (i < j)
- **begin**
- m := $\lfloor (i + j)/2 \rfloor$
- **if** $x > a_m$ **then** i := m + 1
- **else** j := m
- **end**
- **if** $x = a_i$ **then** location := i
- **else** location := 0
- **Return** location{location is the subscript of the term that equals x, or is zero if x is not found}

Complexity

- What is the time complexity of the binary search algorithm?

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

- 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 $2^k < n < 2^{k+1}$.

Complexity

- In the first cycle of the loop
- **while** $(i < j)$
- **begin**
- $m := \lfloor (i + j)/2 \rfloor$
- **if** $x > a_m$ **then** $i := m + 1$
- **else** $j := m$
- **end**

- the search interval is restricted to 2^{k-1} elements, using two comparisons.

Complexity

- In the second cycle, the search interval is restricted to 2^{k-2} elements, again using two comparisons.

- This is repeated until there is only one (2^0) element left in the search interval.

- At this point 2k comparisons have 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 $n = 2^{30}$.

Complexity

- 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	2.5×10^{41}
1,000,000	5×10^9	4.8×10^{41392}

Complexity

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

- So what is 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

- 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**.

- You will find out more about this in CS420.

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 + … + 1
- $= (n – 1)n/2 = 0.5n^2 – 0.5n$
- Time complexity is $O(n^2)$.

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 O(n).