

# Discrete Mathematics for Computer Science

## Lecture 6: Complexity of Algorithms

Dr. Ming Tang

Department of Computer Science and Engineering  
Southern University of Science and Technology (SUSTech)  
Email: tangm3@sustech.edu.cn



## Countable Sets: Example 3

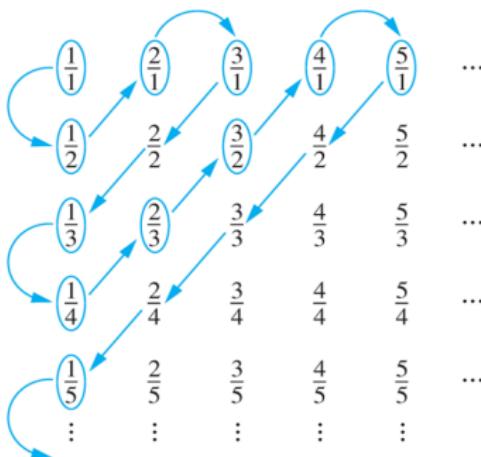
**Theorem:** The set of positive rational numbers is countable.

Hint: prove by showing that the set of positive rational numbers **can be listed in a sequence**: specifying the **initial term** and **rule**

**Solution:**

Constructing the list: first list  $p/q$  with  $p + q = 2$ , next list  $p/q$  with  $p + q = 3$ , and so on.

$1, 1/2, 2, 3, 1/3, 1/4, 2/3, \dots$



# Uncountable Sets

**Theorem:** The set  $\mathcal{P}(\mathbb{N})$  is uncountable.

**Theorem:** The power set of any countably infinite set is uncountable.

**Proof by contradiction:**

Assume that  $\mathcal{P}(\mathbb{N})$  is countable. This implies that the elements of this set can be listed as  $S_0, S_1, S_2, \dots$ , where  $S_i \subseteq \mathbb{N}$ , and each  $S_i$  can be represented uniquely by the bit string  $b_{i0}b_{i1}b_{i2}\dots$ , where  $b_{ij} = 1$  if  $j \in S_i$  and  $b_{ij} = 0$  if  $j \notin S_i$ .

- $S_0 = b_{00}b_{01}b_{02}b_{03}\dots$
  - $S_1 = b_{10}b_{11}b_{12}b_{13}\dots$
  - $S_2 = b_{20}b_{21}b_{22}b_{23}\dots$
  - ⋮
- all  $b_{ij} \in \{0, 1\}$ .

Form a new set called  $R = b_0b_1b_2b_3\dots$ , where  $b_i = 0$  if  $b_{ii} = 1$ , and  $b_i = 1$  if  $b_{ii} = 0$ .  $R$  is different from each set in the list. Each bit string is unique, and  $R$  and  $S_i$  differ in the  $i$ -th bit for all  $i$ .

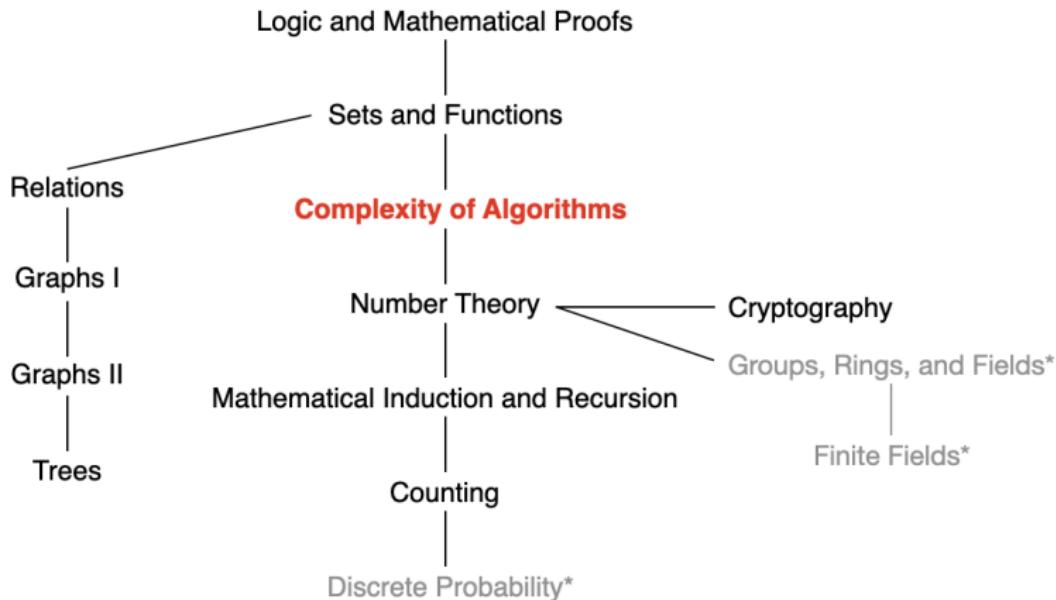
# Computable vs Uncomputable

**Definition:** We say that a function is **computable** if there is a computer program in some programming language that finds the values of this function. If a function is not computable, we say it is **uncomputable**.

**Theorem:** There are functions that are not computable.

- The set of all programs is countable.
- There are uncountably many different functions from a particular countably infinite set to itself (Exercise 38).

# This Lecture



The growth of functions, complexity of algorithm,  
P and NP problem, ....

# Algorithm

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

## ALGORITHM 5 The Insertion Sort.

```
procedure insertion sort( $a_1, a_2, \dots, a_n$ : real numbers with  $n \geq 2$ )
for  $j := 2$  to  $n$ 
     $i := 1$ 
    while  $a_j > a_i$ 
         $i := i + 1$ 
     $m := a_j$ 
    for  $k := 0$  to  $j - i - 1$ 
         $a_{j-k} := a_{j-k-1}$ 
     $a_i := m$ 
{ $a_1, \dots, a_n$  is in increasing order}
```

## ALGORITHM 4 The Bubble Sort.

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

How to compare algorithms with the same functionality?

- Time complexity: number of machine operations (e.g., additions)
- Space complexity: amount of memory needed

Before we get into details, the growth of functions ...

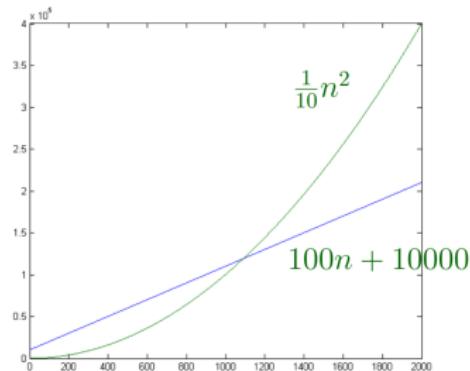
# The Growth of Functions

Which function is “bigger”,  $\frac{1}{10}n^2$  or  $100n + 10000$ ?

It depends on the value of  $n$ .

In Computer Science, we are usually interested in what happens when our problem **input size gets large**.

Notice that when  $n$  is “large enough”,  $\frac{1}{10}n^2$  gets much bigger than  $100n + 10000$  and stays larger.



# The Growth of Functions

- Big-O notation, e.g.,  $O(n^2)$
- Big-Omega notation, e.g.,  $\Omega(n^2)$
- Big-Theta notation, e.g.,  $\Theta(n^2)$

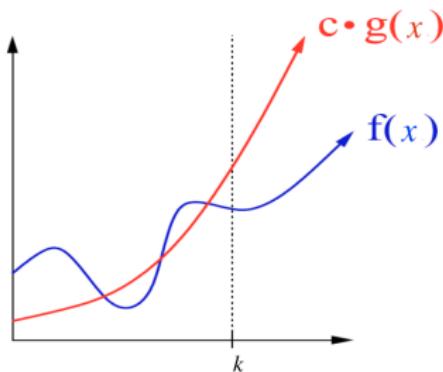
# Big-O Notation

Let  $f$  and  $g$  be functions from the set of integers or the set of real numbers to the set of 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  $g(x)$ .”]



## Big-O Notation: Example

Show that  $f(x) = x^2 + 2x + 1$  is  $O(x^2)$ .

**Proof:** We can readily estimate the size of  $f(x)$  when  $x > 1$ :

$$0 \leq x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2.$$

This is because when  $x > 1$ ,  $x < x^2$  and  $1 < x^2$ . Thus, let  $C = 4$ ,  $k = 1$ :

$$|f(x)| \leq C|x^2|, \text{ whenever } x > k.$$

Hence,  $f(x) = O(x^2)$ .

Note that there are **multiple ways** for proving this. Alternatively, we can estimate the size of  $f(x)$  when  $x > 2$ :

$$0 \leq x^2 + 2x + 1 \leq x^2 + x^2 + x^2 = 3x^2.$$

It follows that  $C = 3$ ,  $k = 2$ . ...

# Big-O Notation: Example

**Examples:** The following formulas are all  $O(x^2)$ :

- $4x^2$
- $8x^2 + 2x - 3$
- $x^2/5 + \sqrt{x} - \log(x)$

Observe that in the relationship “ $f(x)$  is  $O(x^2)$ ,”  $x^2$  can be replaced by any function with “**larger values**” than  $x^2$ . For example,

- $f(x)$  is  $O(x^3)$
- $f(x)$  is  $O(x^2 + x + 7)$ , ...

When  $f(x)$  is  $O(g(x))$ , and  $h(x)$  is a function that has **larger absolute values** than  $g(x)$  does for **sufficiently large values of  $x$** , it follows that

$f(x)$  is  $O(h(x))$ .

# Big-O Estimates for Polynomials

Let  $f(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$ , where  $a_0, a_1, \dots, a_n$  are real numbers. Then,  $f(x) = O(x^n)$ .

## Proof:

Assuming  $x > 1$ , we have

$$\begin{aligned}|f(x)| &= |a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0| \\&\leq |a_n|x^n| + |a_{n-1}|x^{n-1}| + \dots + |a_1|x| + |a_0| \\&= x^n(|a_n| + |a_{n-1}|/x + \dots + |a_1|/x^{n-1} + |a_0|/x^n) \\&\leq x^n(|a_n| + |a_{n-1}| + \dots + |a_1| + |a_0|).\end{aligned}$$

The leading term  $a_nx^n$  of a polynomial **dominates** its growth.

# Big-O Estimates for Some Functions

$$1 + 2 + \dots + n = O(n^2)$$

$$n! = O(n^n)$$

$$\log n! = O(n \log n)$$

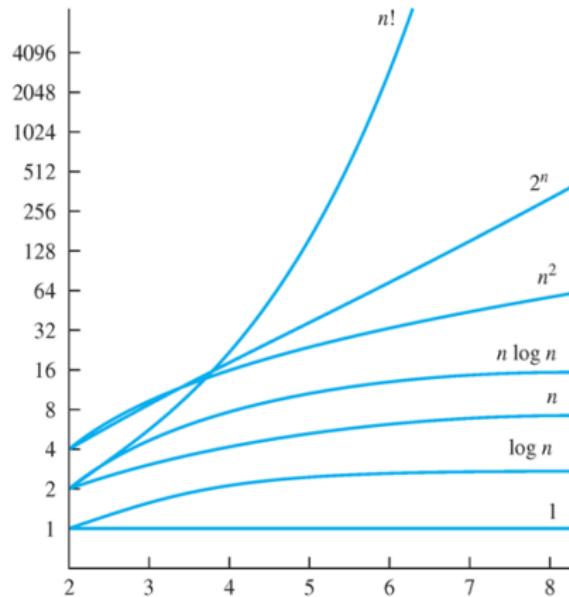
$$\log_a n = O(n) \text{ for an integer } a \geq 2$$

$$n^a = O(n^b) \text{ for integers } a \leq b$$

$$n^a = O(2^n) \text{ for an integer } a$$

In our textbook, p211-212

# Big-O Estimates for Some Functions



In our textbook, p211-212

# Combinations of Functions

If  $f_1(x)$  is  $O(g_1(x))$  and  $f_2(x)$  is  $O(g_2(x))$ , then

$$(f_1 + f_2)(x) = O(\max(|g_1(x)|, |g_2(x)|)).$$

**Proof:**

By definition, there exist constants  $C_1, C_2, k_1, k_2$  such that

$$|f_1(x)| \leq C_1|g_1(x)| \text{ when } x > k_1 \text{ and}$$

$$|f_2(x)| \leq C_2|g_2(x)| \text{ when } x > k_2. \text{ Then}$$

$$\begin{aligned} |(f_1 + f_2)(x)| &= |f_1(x) + f_2(x)| \\ &\leq |f_1(x)| + |f_2(x)| \\ &\leq C_1|g_1(x)| + C_2|g_2(x)| \\ &\leq C_1|g(x)| + C_2|g(x)| \\ &= (C_1 + C_2)|g(x)| \\ &= C|g(x)|, \end{aligned}$$

where  $g(x) = \max(|g_1(x)|, |g_2(x)|)$  and  $C = C_1 + C_2$ .

$$k = \max\{k_1, k_2\}.$$

# Combinations of Functions

If  $f_1(x)$  is  $O(g_1(x))$  and  $f_2(x)$  is  $O(g_2(x))$  then  $(f_1 f_2)(x) = O(g_1(x)g_2(x))$ .

**Proof:**

When  $k > \max(k_1, k_2)$ ,

$$\begin{aligned}|(f_1 f_2)(x)| &= |f_1(x)||f_2(x)| \\&\leq C_1|g_1(x)|C_2|g_2(x)| \\&\leq C_1 C_2|(g_1 g_2)(x)| \\&\leq C|(g_1 g_2)(x)|,\end{aligned}$$

where  $C = C_1 C_2$ .

# Big-Omega Notation

Let  $f$  and  $g$  be functions from the set of integers or the set of real numbers to the set of real numbers. We say that  $f(x)$  is  $\Omega(g(x))$  if there are positive constants  $C$  and  $k$  such that

$$|f(x)| \geq C|g(x)|$$



whenever  $x > k$ . [This is read as “ $f(x)$  is big-Omega of  $g(x)$ .”]

Big-O gives an **upper bound** on the growth of a function, while Big- $\Omega$  gives a **lower bound**.

Big- $\Omega$  tells us that a function grows **at least** as fast as another.

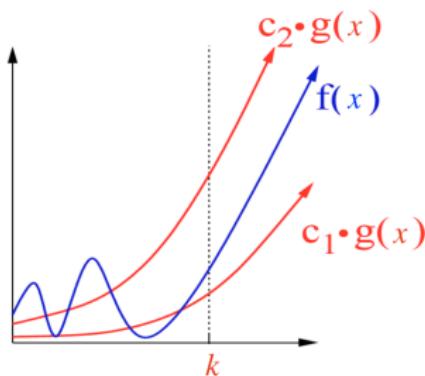
**Note:**  $f(x)$  is  $\Omega(g(x))$  if and only if  $g(x)$  is  $O(f(x))$ .

# Big-Theta Notation (Big-O & Big-Omega)

Let  $f$  and  $g$  be functions from the set of integers or the set of real numbers to the set of real numbers. We say that  $f(x)$  is  $\Theta(g(x))$  if

- $f(x)$  is  $O(g(x))$  and
- $f(x)$  is  $\Omega(g(x))$ .

When  $f(x)$  is  $\Theta(g(x))$ , we say that  $f(x)$  is big-Theta of  $g(x)$ , that  $f(x)$  is of order  $g(x)$ , and that  $f(x)$  and  $g(x)$  are of the same order.



# Big-Theta Notation

**Theorem:** Let  $f(x) = a_nx^n + a_{n-1}x^{n-1} + \dots + a_1x + a_0$ , where  $a_0, a_1, \dots, a_n$  are real numbers with  $a_n \neq 0$ . Then  $f(x)$  is of order  $x^n$ .

- $f(x) = O(x^n)$
- $f(x) = \Omega(x^n)$

# Big-Theta Notation: Examples

$$3n^2 + 4n = \Theta(n) ?$$

No

$$3n^2 + 4n = \Theta(n^2) ?$$

Yes

$$3n^2 + 4n = \Theta(n^3) ?$$

No, but  $O(n^3)$

$$n/5 + 10n \log n = \Theta(n^2) ?$$

No, but  $O(n^2)$

$$n^2/5 + 10n \log n = \Theta(n \log n) ?$$

No

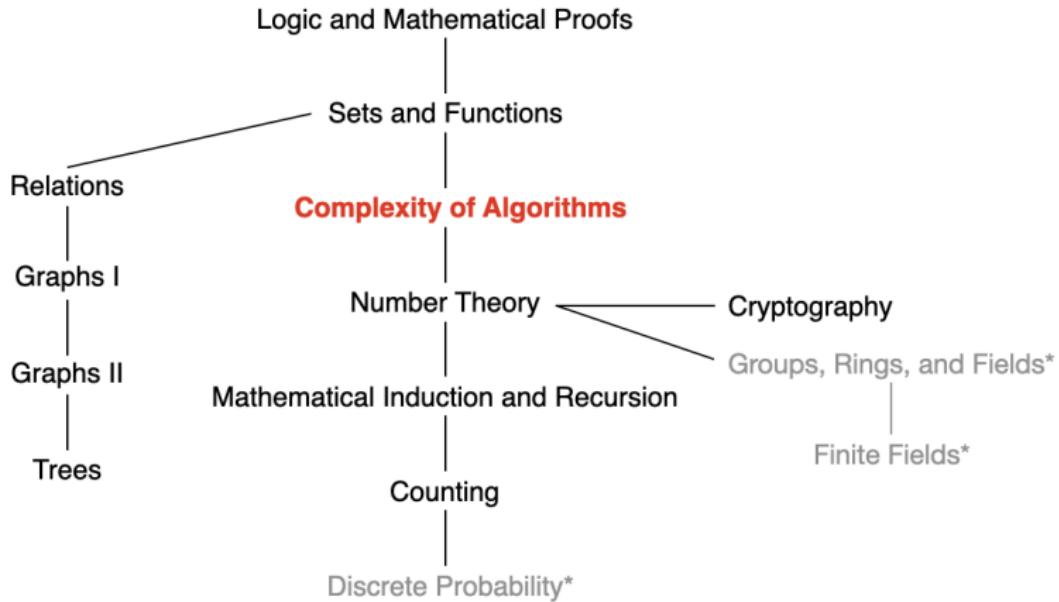
$$n^2/5 + 10n \log n = \Theta(n^2) ?$$

Yes

# The Growth of Functions

- Big-O notation, e.g.,  $O(n^2)$ 
  - ▶ Upper bound
- Big-Omega notation, e.g.,  $\Omega(n^2)$ 
  - ▶ Lower bound
- Big-Theta notation, e.g.,  $\Theta(n^2)$ 
  - ▶ Of the same order

# This Lecture



The growth of functions, **complexity of algorithm**,  
P and NP, ....

# Algorithms

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

A **computational problem** is a specification of the desired input-output relationship.

**Example** (Computational Problem and Algorithm):

- Computational Problem: Input  $n$  numbers  $a_1, a_2, \dots, a_n$ ; Output the sum of the  $n$  numbers.
- Algorithm: the following procedures

Step 1: set  $S = 0$

Step 2: for  $i = 1$  to  $n$ , replace  $S$  by  $S + a_i$

Step 3: output  $S$

# Instance

An **instance** of a problem is a realization of **all the inputs** needed to compute a solution to the problem.

**Example:** 8, 3, 6, 7, 1, 2, 9

A **correct algorithm** halts with the **correct output** for every input instance.  
We can then say that the algorithm solves the problem.

# Time and Space Complexity

- Time complexity: The number of machine operations (**addition, multiplication, comparison, replacement**, etc) needed in an algorithm.
- Space complexity: the amount of memory needed.

## Example (Algorithm)

Step 1: set  $S = 0$

Step 2: for  $i = 1$  to  $n$ , replace  $S$  by  $S + a_i$

Step 3: output  $S$

## Time Complexity:

- Steps 1 and 3 take one operation.
- Step 2 takes  $2n$  operations.

Therefore, altogether this algorithm takes  $1 + 2n + 1$  operations. The time complexity is  $O(n)$ .

# Horner's Algorithm and Its Complexity

**Example:** Consider the evaluation of  $f(x) = 1 + 2x + 3x^2 + 4x^3$ . Direct computation takes 3 additions and 6 multiplications.

**Can we do better?**

Another way is  $f(x) = 1 + x(2 + x(3 + 4x))$ , which takes 3 additions and 3 multiplications.

**Horner's algorithm** for computing

$f(x) = a_0 + a_1x + \dots + a_{n-1}x^{n-1} + a_nx^n = a_0 + x(a_1 + \dots + x(a_{n-1} + a_nx))$   
at a particular  $x$ :

**Step 1:** set  $S = a_n$

**Step 2:** for  $i = 1$  to  $n$ , replace  $S$  by  $a_{n-i} + Sx$

**Step 3:** output  $S$

The number of operations needed in this algorithm is  $1 + 3n + 1 = 3n + 2$ .  
So the time complexity of this algorithm is  $O(n)$ .

**Note:** Operations: addition, multiplication, comparison, replacement, etc.

# Time Complexity: Example

Determine the time complexity of the following algorithm:

```
for i := 1 to n
    for j := 1 to n
        a := 2 * n + i * j;
    end for
end for
```

- Computing  $a := 2 \times n + i \times j$  takes 4 operations (two multiplications, one addition, and one replacement).
- For each  $i$ , it takes  $4n$  operations to complete the second loop.
- Thus, this algorithm takes  $n \times 4n = 4n^2$  operations to complete the two loops. The time complexity of this algorithm is  $O(n^2)$ .



Southern University  
of Science and  
Technology

# Time Complexity: Example

Determine the time complexity of the following algorithm:

```
S := 0
for i := 1 to n
    for j := 1 to i
        S := S + i * j;
    end for
end for
```

- Computing  $S := S + i \times j$  takes 3 operations.
- For each  $i$ , completing the second loop takes  $3i$  operations.
- Thus, this algorithm takes

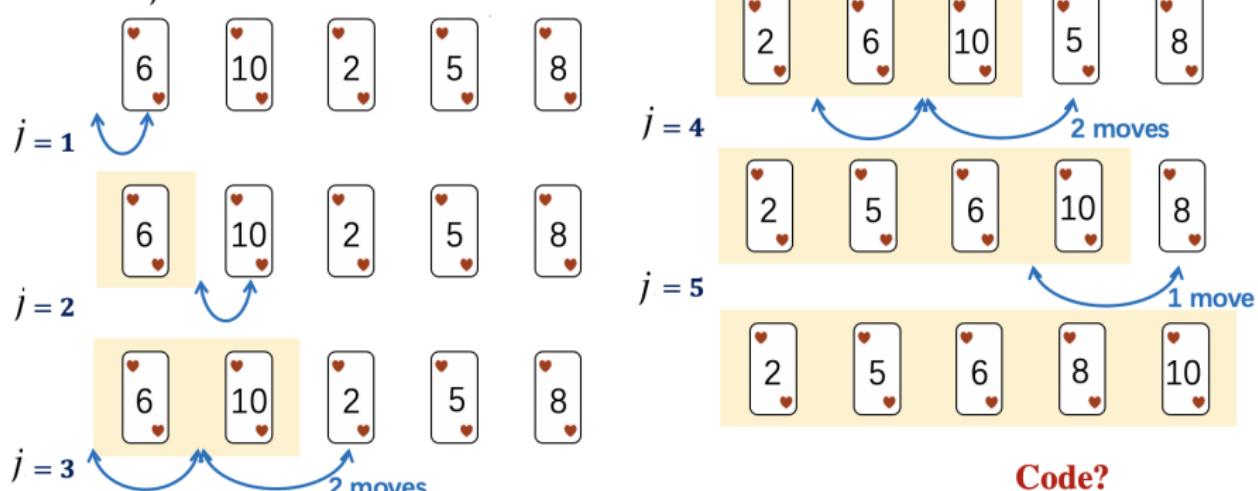
$$1 + \sum_{i=1}^n 3i = 1 + 3 \frac{n(n+1)}{2}$$

So the complexity of this algorithm is  $O(n^2)$ .



# Time Complexity: Example - Insertion Sort

In iteration  $j$ , we move the  $j$ -th element left until its correct place is found among the first  $j$  elements.



# Time Complexity: Example - Insertion Sort

**Input:**  $A[1 \dots n]$  is an array of numbers

for  $j := 2$  to  $n$

$key = A[j];$

$i = j - 1;$

    while  $i \geq 1$  and  $A[i] > key$  do

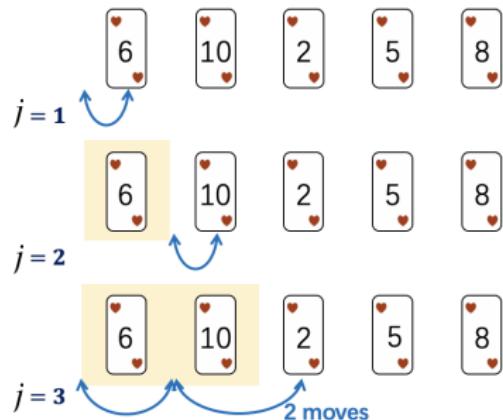
$A[i + 1] = A[i];$

$i --;$

    end while

$A[i + 1] = key;$

end for



The time complexity depends on the input array  $A[1, \dots, n]$ .

**Consider only the number of comparisons**

## Three Cases of Analysis: Best-Case

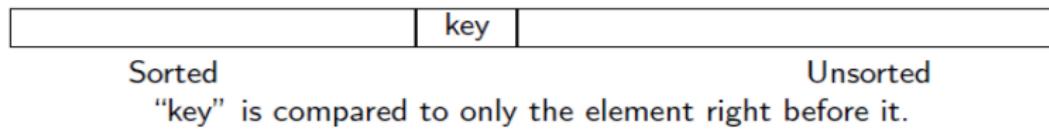
**Best-Case Complexity:** The **smallest** number of operations needed to solve the given problem using this algorithm on **input of specified size**.

**Example:** (Insertion Sort)

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A[n]$$

The number of comparisons needed is

$$\underbrace{1 + 1 + 1 + \cdots + 1}_{n-1} = n - 1 = \Theta(n)$$



## Three Cases of Analysis: Worst-Case

**Worst-Case Complexity:** The **largest** number of operations needed to solve the given problem using this algorithm on **input of specified size**.

**Example:** (Insertion Sort)

$$A[1] \geq A[2] \geq A[3] \geq \dots \geq A[n]$$

The number of comparisons needed is

$$1 + 2 + 3 + \dots + (n - 1) = \frac{n(n-1)}{2} = \Theta(n^2)$$

	key	
--	-----	--

Sorted

Unsorted

"key" is compared to everything element before it.

## Three Cases of Analysis: Average-Case

**Average-Case Complexity:** The **average number of operations** used to solve the problem **over all possible inputs** of a given size is found in this type of analysis.

**Example:** (Insertion Sort)

$\Theta(n^2)$  assuming that each of the  $n!$  instances are **equally likely**

	key	
--	-----	--

Sorted

Unsorted

On average, “key” is compared to half of the elements before it.

- For a particular instance, compute the number of comparisons
- Since we assume equal probability, take the average

Average-case complexity is usually difficult to compute.



Southern University  
of Science and  
Technology

# Some Thoughts on Algorithm Design

Algorithm Design is mainly about designing algorithms that have small Big- $O$  running time.

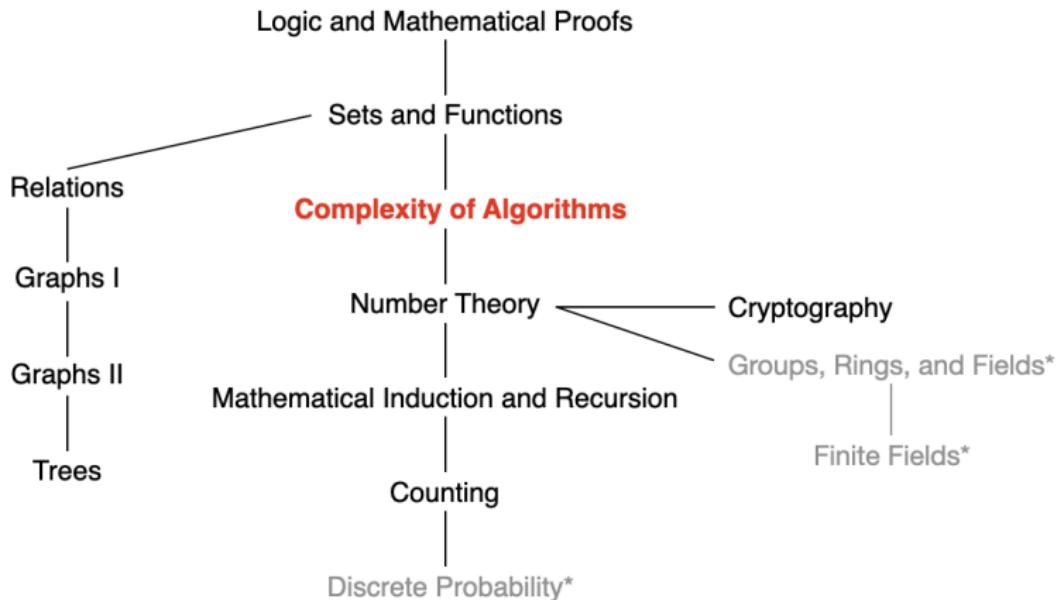
Being able to do good algorithm design lets you identify the hard parts of your problem and deal with them effectively.

Too often, programmers try to solve problems using brute force techniques and end up with slow complicated code!

- The most straightforward manner based on the statement of the problem and the definitions of terms

A few hours of abstract thought devoted to algorithm design could speed up the solution substantially and simplified it!

# This Lecture



The growth of functions, complexity of algorithm,  
P and NP problem, ....

# Dealing with Hard Problems

What happens if you **cannot** find an efficient algorithm for a given problem?

Blame yourself.



I couldn't find a polynomial-time algorithm.  
I guess I am too dumb.

# Dealing with Hard Problems

What happens if you **cannot** find an efficient algorithm for a given problem?

Show that **no**-efficient algorithm exists.



I couldn't find a polynomial-time algorithm,  
because **no** such algorithm exists.

# Dealing with Hard Problems

Showing that a problem **has** an efficient algorithm is, **relatively easy**:

- Design such an algorithm.

Proving that **no** efficient algorithm exists for a particular problem is **difficult**:

How can we prove the non-existence of something?

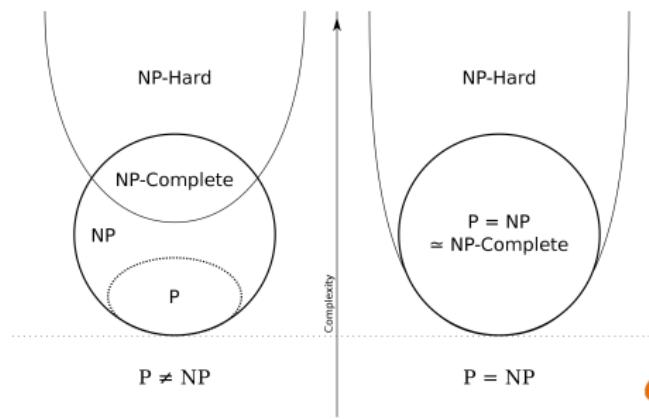
We will now learn about **NP-Complete problems**, which provides us with a way to approach this question.

# NP-Complete

**P:** Problems that are solvable using an algorithm with polynomial worst-case complexity

**NP:** Problems for which a solution can be checked in polynomial time.

**NP-Complete:** If any of these problems can be solved by a polynomial worst-case time algorithm, then all problems in the class NP can be solved by polynomial worst-case time algorithms.



# NP-Complete

Researchers have spent many years trying to find efficient solutions to these problems but **failed**.

NP-Complete and NP-Hard problems are very likely to be **hard**.

Thus, to proving that no efficient algorithm exists for a particular problem?

**Prove that your problem is NP-Complete or even NP-Hard:**

- Show that your problem can be reduced to a typical (well-known) NP-Complete or NP-Hard problem.

# NP-Complete

What do you actually do:



I couldn't find a polynomial-time algorithm,  
but neither could all these other smart people!

# Dealing with Hard Problems

- Input size of a problem
- Different types of problems
- Polynomial-time algorithms
- Details for P and NP

# Encoding the Inputs of Problems

Complexity of a problem is measure with respect to **the size of input**:

- E.g., for insertion sort,  $\Theta(n^2)$  is the average-case complexity, where  $n$  is the length of the array.

In order to formally discuss how hard a problem is, we need to be much more **formal** than before about **the input size of a problem**.

# The Input Size of Problems

The input size of a problem might be defined in a number of ways.

Now, we consider the following definition:

**Definition:** The input size of a problem is the minimum number of bits (i.e., {0, 1}) needed to encode the input of the problem.

The exact input size  $s$ , determined by an optimal encoding method, is hard to compute in most cases.

For most problems, it is sufficient to choose some natural and (usually) simple encoding and use the size  $s$  of this encoding.

- E.g., 5 can be encoded as 101.

## Input Size Example: Composite

**Example:** Input a positive integer  $n$ ; output if there are integers  $j, k > 1$  such that  $n = jk$ ? (i.e., is  $n$  a composite number?)

**Question:** What is the input size of this problem?

Any integer  $n > 0$  can be represented in the binary number system as a string  $a_0a_1\dots a_k$  of length  $\lceil \log_2(n + 1) \rceil$ .

Thus, a natural measure of input size is  $\lceil \log_2(n + 1) \rceil$  (or just  $\log_2 n$ )

# Input Size Example: Sorting

**Example:** Sort  $n$  integers  $a_1, \dots, a_n$ .

**Question:** What is the input size of this problem?

Using **fixed length** encoding, we write  $a_i$  as a binary string of length  $m = \lceil \log_2 \max(|a_i| + 1) \rceil$ .

This coding gives an **input size of  $nm$** .

**Note:** Back to our earlier discussions for complexity, when we use fixed length encoding regardless of  $a_i$  for  $i = 1, 2, \dots, n$ , the value of  $m$  becomes a constant. Thus, we can omit the constant  $m$ .

# Complexity in terms of Input Size

**Example (Composite):** The naive algorithm for determining whether  $n$  is composite compares  $n$  with the first  $n - 1$  numbers to see if any of them divides  $n$ .

This makes  $\Theta(n)$  comparisons, so it might seem linear and very efficient.

**But**, the input size of this problem is  $\log_2 n$  instead of  $n$ . The number of comparisons performed is actually  $\Theta(n)$ , which can be represented as  $\Theta(2^{(\log_2 n)})$ . It is **exponential** with respect to the input size.

# Functions of the Same Type

**Definition:** Two positive functions  $f(n)$  and  $g(n)$  are of the same type if

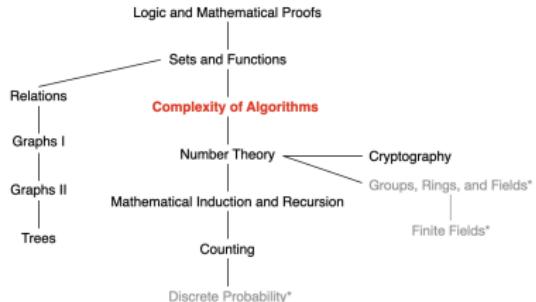
$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$

for all large  $n$ , where  $a_1, b_1, c_1, a_2, b_2, c_2$  are some positive constants.

## Example:

- All polynomials are of the **same** type
- Polynomials and exponentials are of **different** types.

# This Lecture



The growth of functions, complexity of algorithm,  
P and NP:

- Input size of a problem
- Different types of problems
- Polynomial-time algorithms
- Details for P and NP