fit@hcmus

DATA STRUCTURES & ALGORITHMS

Lecture 1: Introduction to Algorithms & Algorithm Analysis

Lecturer: Dr. Nguyen Hai Minh

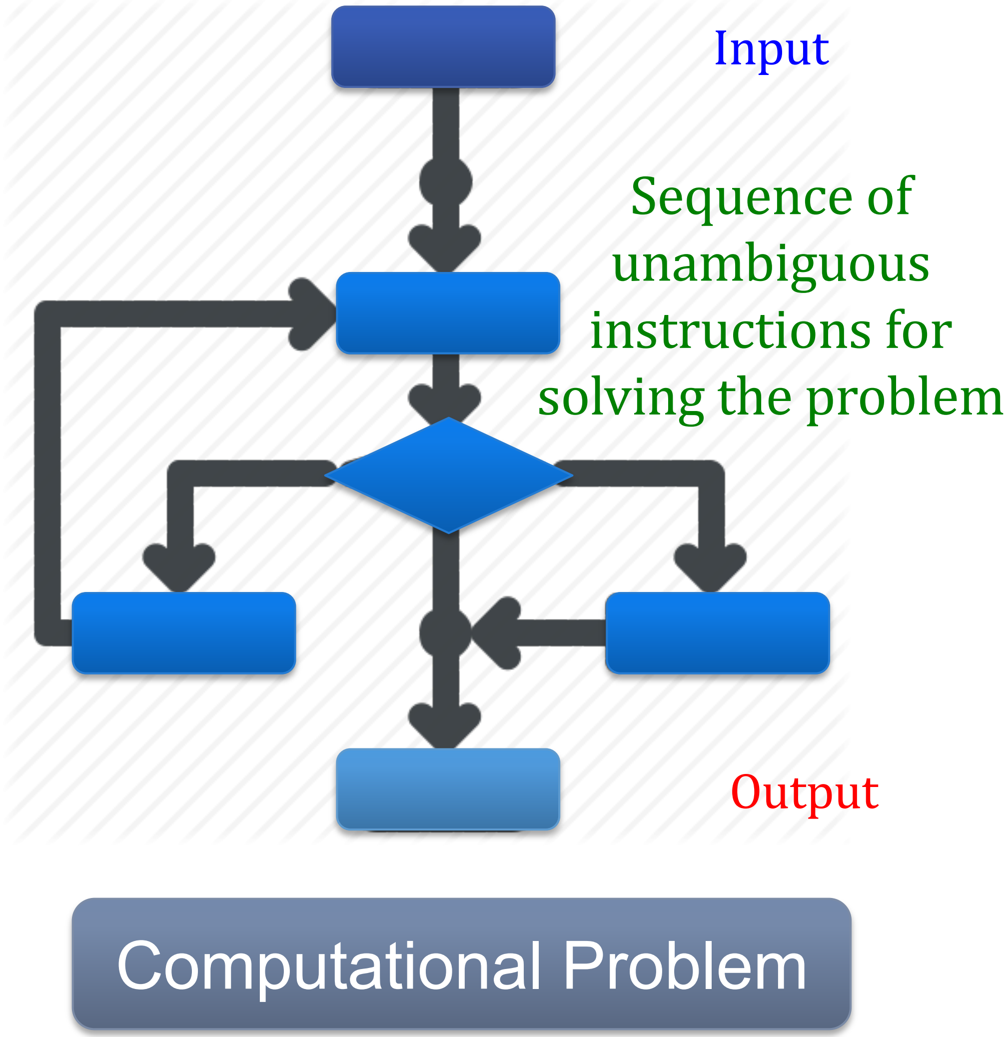
CONTENT

- Role of Algorithms in Computing
- Algorithm Analysis Framework
- Asymptotic Annotations
- Mathematical Analysis of Algorithm

ROLE OF ALGORITHMS IN COMPUTING

What is Algorithm?

- Algorithm:
  - well-defined ***computational procedure*** that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***



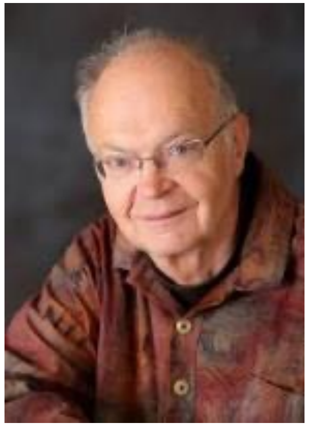
Why should we study algorithm?

- Computer programs would not exist without algorithms.
- Studying algorithms help developing analytical skill
  - Algorithm can be seen as special kinds of solutions to problems – not just answer but precisely **defined procedures** for getting answers.
  - Consequently, specific algorithm design techniques can be interpreted as **problem-solving strategies** that can be useful in other fields, not just in computing.

→ *Algorithmic thinking*

Why should we study algorithm?

A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them.



- Donald Knuth -

What kind of problems are solved by Algorithm?

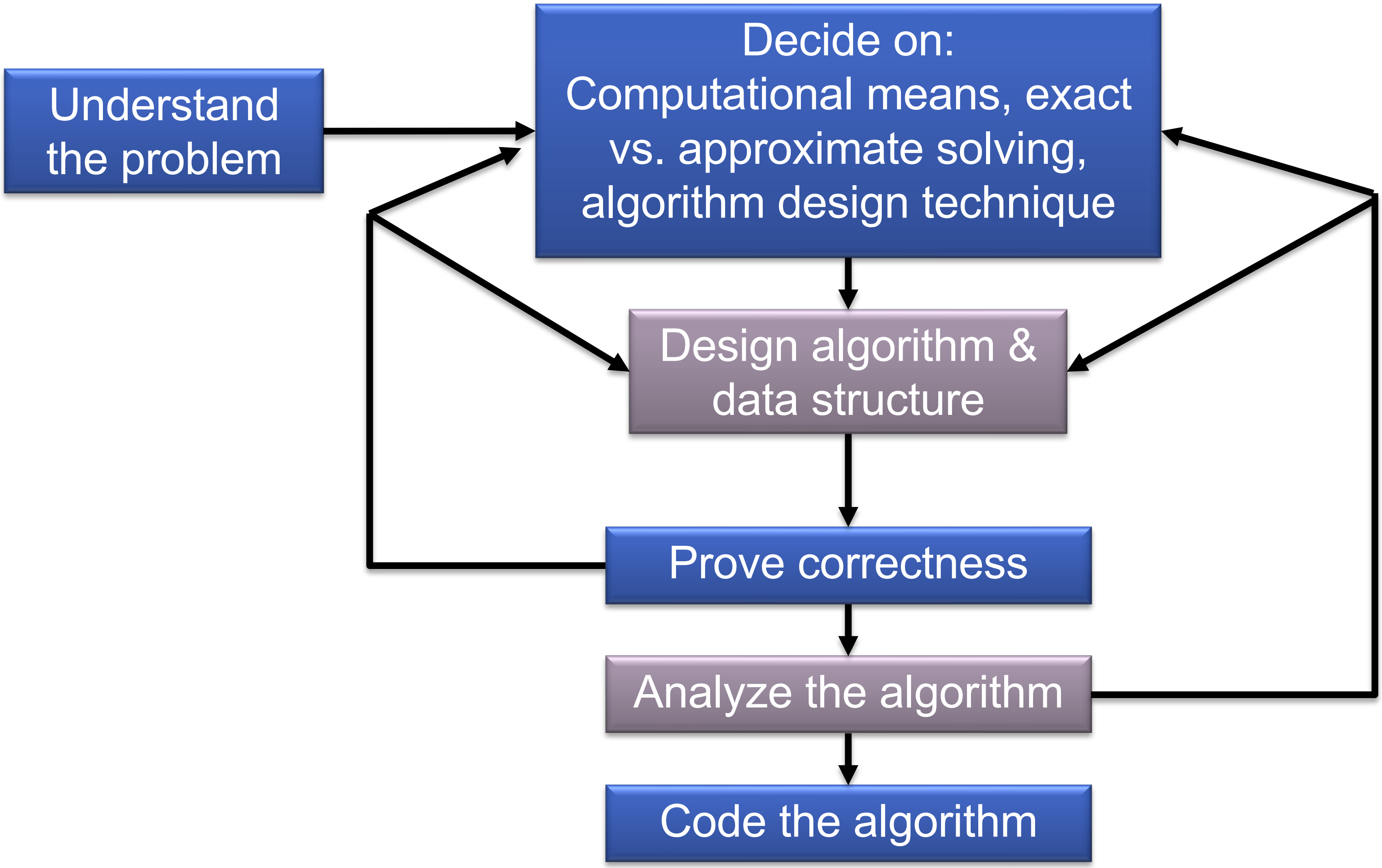
- Important Problem Types:
  - Sorting
  - Searching
  - String matching
  - Graph problems
  - Combinatorial problems
  - Geometric problems
  - Numerical problems

→ These problems are introduced in the subsequent lectures to illustrate different algorithm design techniques and methods of algorithm analysis

Problems that cannot be solved by Algorithm?

- The precision inherently imposed by algorithmic thinking limits the kinds of problems that can be solved with an algorithm.
- You will not find algorithms for:
  - Living a happy life
  - Becoming a millionaire
  - Living forever
  - …

Algorithmic Problem Solving



Designing Algorithms

- Brute-force & Exhaustive Search
- Decrease and Conquer
- Divide and Conquer
- Transform and Conquer
- Space and Time Trade-offs
- Dynamic Programming
- Greedy Technique
- Iterative Improvement
- Backtracking
- Branch-and-bound
- Approximation algorithms

Designing Data Structure

- Linear Data Structure:
  - Array
  - Linked List
  - Stack
  - Queue
  - Hash Table
- Trees
- Graphs

ALGORITHM ANALYSIS FRAMEWORK

Measuring an Input's size

Units for Measuring Running Time

Order of Growth

Kinds of Analysis

Algorithm Analysis

- The theoretical study of computer-program performance and resource usage.
  - Time efficiency
  - Space efficiency
- What is more important than performance?
  - modularity
  - correctness
  - maintainability
  - functionality
  - robustness
  - user-friendliness
  - programmer time
  - simplicity
  - extensibility
  - reliability

Algorithm Analysis

- Nowadays, the amount of extra space required by an algorithm is typically not of as much concern.
- In most problems, we can achieve much more spectacular progress in **speed** than in space.

→ *We primarily concentrate on* ***time efficiency****, but analytical framework in this course is applicable to analyzing space efficiency as well.*

Performance (efficiency) of Algorithms

- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a language for talking about program behavior.
- Performance is the currency of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

Measuring an Input's Size

- Almost all algorithms run *longer on larger inputs*.
- For example:
  - Sorting arrays: A1 = {12, 1, 3}
  - Sorting arrays: A2 = {88, 12, 3, 19, 32, 9, 1, 3, 45, 17, 89, 12, 34, 52, 61, 41, 24, 98, 19, 38}
- Algorithm's efficiency is investigated as a function of some parameter ***n*** indicating the algorithm's **input size**.

Measuring an Input's Size

- **Straightforward:** problems dealing with lists (e.g., sorting, searching, min, max, …)
  - *n* is the size of the list
- **Not straightforward:**
  - Computing the product of two matrix
  - Checking primality of a positive integer n
  - Finding GCD of two numbers
  - Spell-checking a document
  - …

Units for Measuring Running Time

- Algorithm's running time depends on:
  - Computer speed (hardware, software).
  - Using resource (memory, disk).
  - Implementation of algorithm
- How to analyze running time correctly?
  - Ignore machine-dependent time
  - Using "logic" metrics (ex: numbers of operations: +,-,*,/,<,>,=…) rather than real time metrics (mili-seconds, seconds, minutes, hours, …)

Machine-independent Time

Units for Measuring Running Time

- Count the number of primitive operations or steps executed (the most time-consuming operation in the algorithm's ***innermost loop***)
- For example:
  - Most sorting algorithms work by comparing elements (keys) of a list & exchanging elements → basic operation is **key comparison** (<, >, ==) and **assignment** (=)
- Then, running time of an algorithm can be seen as a **cost function** that depends on the *size of input*.

Units for Measuring Running Time

- Sum of *n* integer:

```
sum = 0;
    for  (i = 0; i < n; i++)
        sum = sum + i;
```

Assignment: $2n+2$

```
sum = 0;
    for  (i = 0; i < n; i++)
        sum = sum + i;
```

Comparison: $n+1$

- Running time: $T(n) = 3n + 3$

Order of Growth

- We should focus on the count's **order of growth** for large input size!
  - For small inputs, the difference in running time is not what really distinguishes efficient algorithms from inefficient ones.
  - Example: powering a number by *n*
    - Decrease-by-one technique
    - Divide-and-Conquer technique

    → *The efficiency of two algorithms becomes clear and important when n is large.*

Order of Growth (Rate of Growth)

- For large values of n ($n \to \infty$), the function's order of growth is important!
  - The growth of T depends on n

n	$T(n)$	$3n$		3	
	Value	Value	%	Value	%
100	303	300	99.01	3	0.99
1000	3,003	3,000	99.93	3	0.10
10,000	30,003	30,000	99.99	3	0.01
100,000	300,003	300,000	100	3	0.00

  - Ignore ***very small parts*** in the cost function.
  - $T(n) = 3\boldsymbol{n} + 3$

Order of Growth (Rate of Growth)

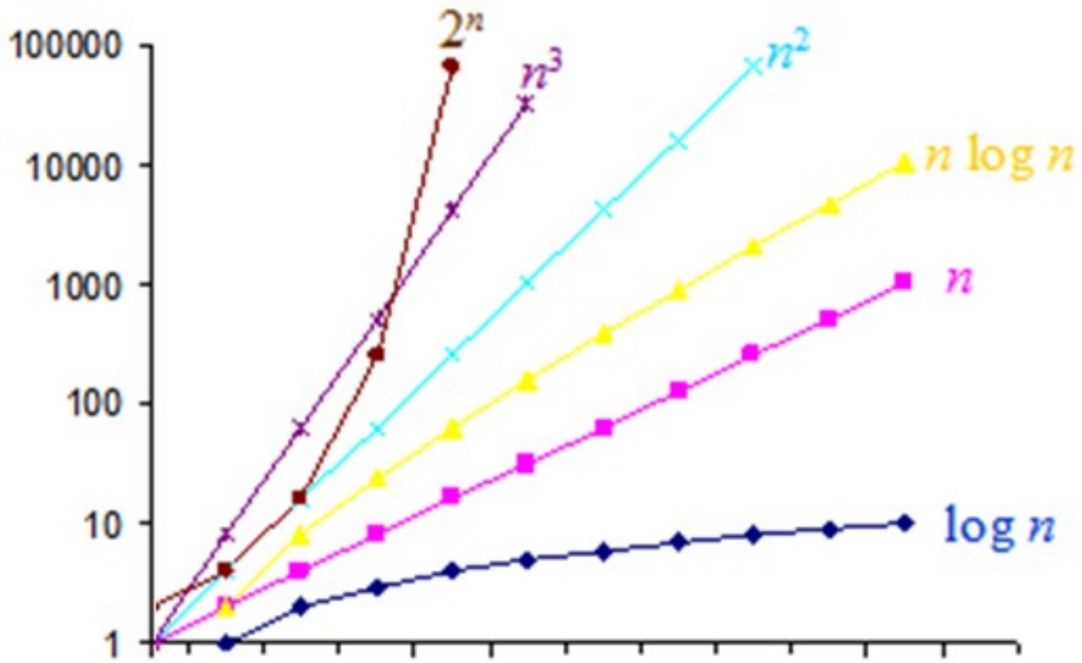
- Another example:

n	$T(n)$	n^2		$100n$		$\log_{10}n$		1000	
	Value	Value	%	Value	%	Value	%	Value	%
1	1,101	1	0.09	100	9.08	0	0.0	1,000	90.83
10	2,101	100	4.76	1,000	47.6	1	0.05	1,000	47.60
100	21,002	10,000	47.6	10,000	47.6	2	0.991	1,000	4.76
1,000	1,101,003	1,000,000	90.8	100,000	9.1	3	0.0003	1,000	0.09
10,000	101,001,004	100,000,000	99.0	1,000,000	0.99	4	0.0	1,000	0.001
100,000	10,010,001,005	10,000,000,000	99.9	10,000,000	0.099	5	0.0	1,000	0.0

$$T(n) = n^2 + 100n + \log_{10}n + 1000$$

➔ The growth of T depends on $\boldsymbol{n^2}$

Comparison of functions



1	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	0	1	0	1	1	2	1
1	1	2	2	4	8	4	2
1	2	4	8	16	64	16	24
1	3	8	24	64	512	256	40,320
1	4	16	64	256	4096	65,536	$2.092279*10^{13}$
1	5	32	160	1,024	32,768	4,294,967,296	$2.6313084*10^{35}$

Kinds of analyses

- There are many algorithms for which running time depends not only on input size but also on the specifics of a **particular input**.
- For example: Insertion Sort runs fastest if the array is already sorted, slowest if the array is in decreasing order.

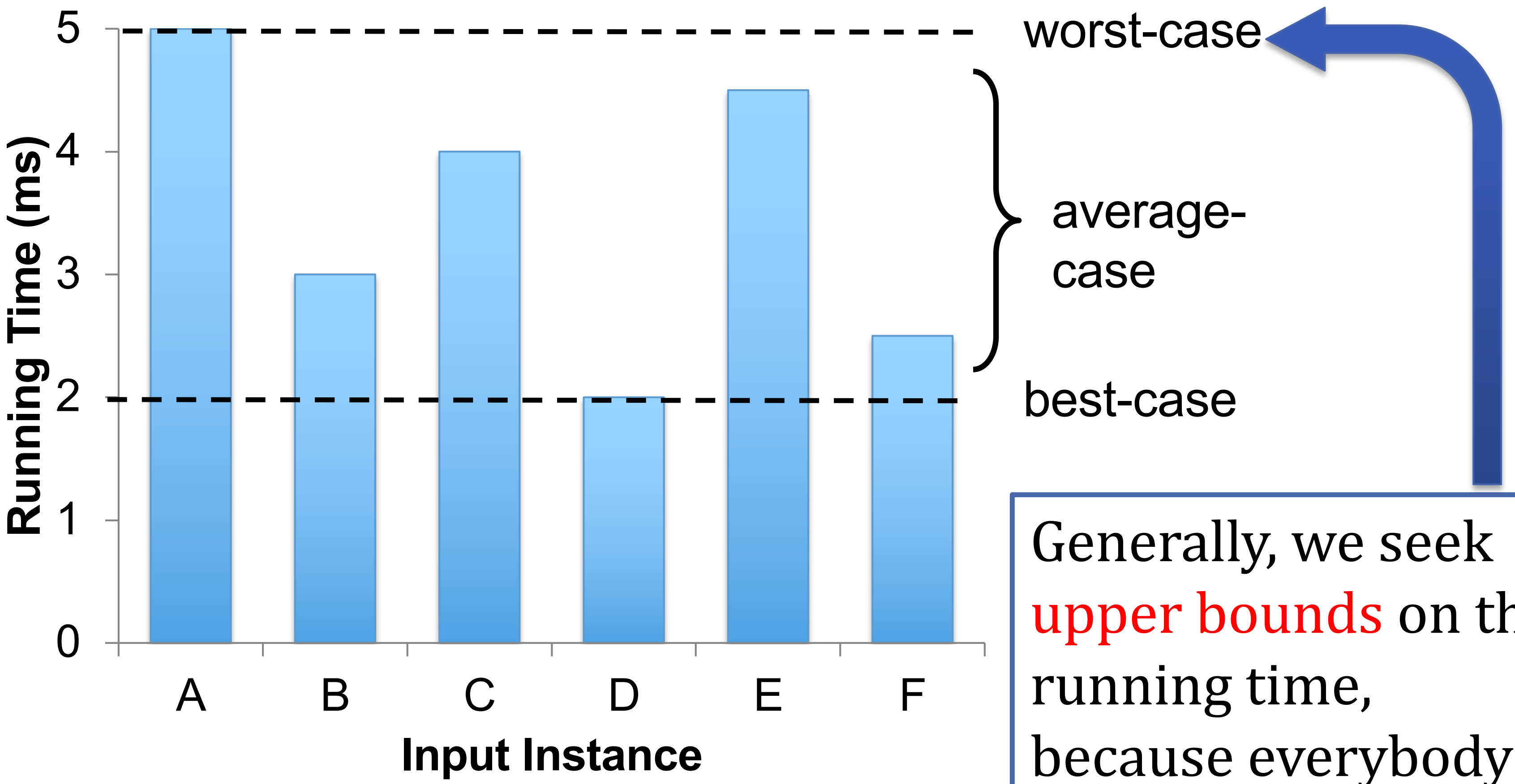
Kinds of analyses

- **Worst-case:** (usually)
  - $T(n)$ = maximum time of algorithm on any input of size n.
- **Average-case:** (sometimes)
  - $T(n)$ = expected time of algorithm over all inputs of size n.

  → Need assumption of statistical distribution of inputs.
- **Best-case:** (bogus)

  → Cheat with a slow algorithm that works fast on *some* input.

Kinds of analyses



Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Insertion Sort Analysis

INSERTION-SORT(A,n)	Cost times
for i = 1 **to** n - 1	$c_1 \quad n$
key = A[i]	$c_2 \quad n-1$
//Insert A[i] into the sorted subarray A[1:i-1]	$0 \quad n-1$
j = i – 1	$c_4 \quad n-1$
while j ≥ 0 and A[j] > key	$c_5 \quad \sum_{i=1}^{n-1} t_i$
A[j+1] = A[j]	$c_6 \quad \sum_{i=1}^{n-1} (t_i - 1)$
j = j – 1	$c_7 \quad \sum_{i=1}^{n-1} (t_i - 1)$
A[j+1] = key	$c_8 \quad n-1$

$$\boldsymbol{T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=1}^{n-1} t_i + c_6 \sum_{i=1}^{n-1} (t_i - 1) + c_7 \sum_{i=1}^{n-1} (t_i - 1) + c_8(n-1)}$$

Insertion Sort Analysis

- **Best case**: the array has been sorted
  - While loop always exists upon the first test in line 5
  - Therefore, $t_i = 1$ for all i = 1, …, $n – 1$
  - The best case running time is given by:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$
$$= (c_1 + c_2 + c_4 + c_8)n - (c_2 + c_4 + c_5 + c_8) = an + b$$

→ The running time is thus a ***linear function*** of *n*

Insertion Sort Analysis

- **Worst case**: the array is in reverted sorted
  - The procedure must compare each element A[i] with each element in the entire sorted subarray
  - Therefore, $t_i = i$ for all $i = 1, \ldots, n-1$
  - The worst case running time is given by:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n-1)}{2} - 1\right)$$
$$+ c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$
$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_{4+}\frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n$$
$$-(c_2 + c_4 + c_5 + c_8) = an^2 + bn + c$$

→ The running time is thus a ***quadratic function*** of *n*

Insertion Sort Analysis

- **Average case**: the array is in randomly chosen number.
  - On average, half the elements in A[1 : i – 1] are less than A[i], and half the elements are greater.
  - Therefore, $t_i = i/2$ for $i = 1, \ldots, n - 1$
  - The average case running time is thus a ***quadratic function*** of n

ASYMPTOTIC NOTATIONS

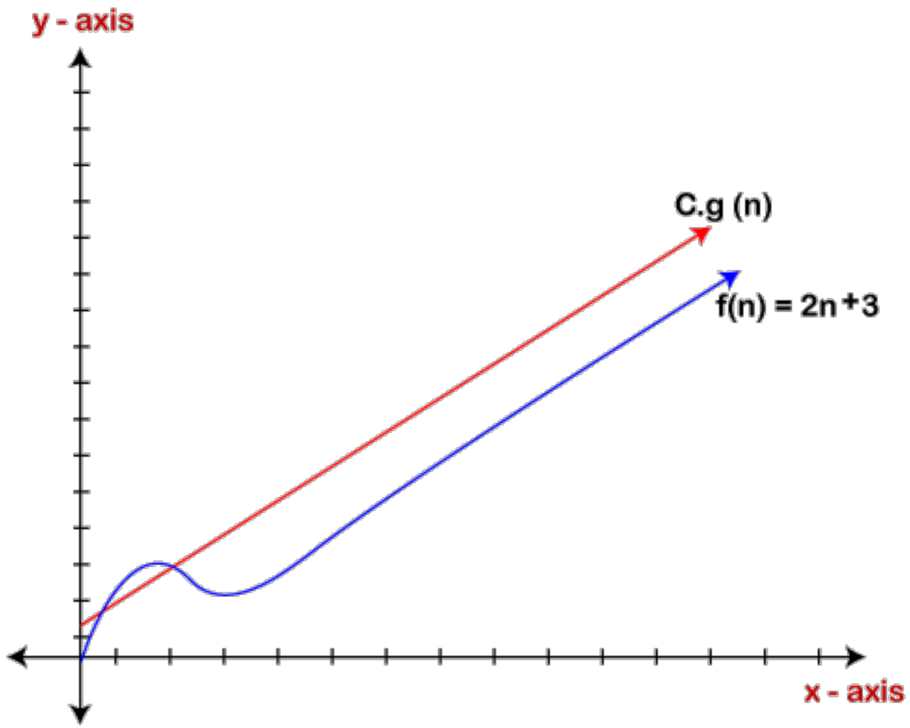
Big-O notation

Basic Efficiency Classes

Asymptotic Analysis

Look at ***growth*** of $f(n)$ as $n \to \infty$

"Asymptotic Analysis"



Asymptotic Notations

- Efficiency analysis concentrates on the **order of growth** of an algorithm's basic operation count.
- To compare such order of growth, computer scientists use 3 notations:

O **Big-Oh**

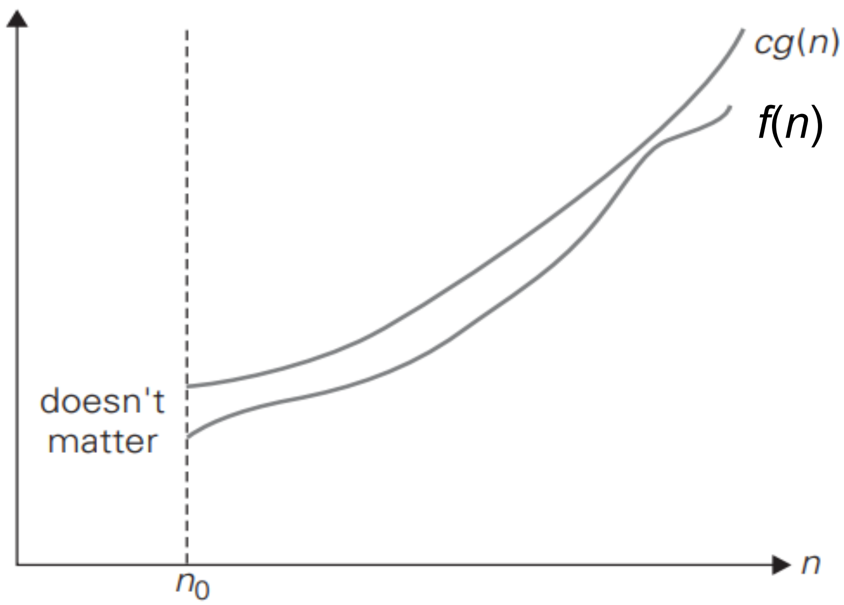
Ω **Big Omega**

Θ **Big Theta**

Asymptotic Notations – Big-Oh

- $\boldsymbol{O(g(n))}$: set of all functions with a **lower** or **same** order of growth as $g(n)$
  - E.g., $n \in O(n^2), 100n + 5 \in O(n^2), \frac{1}{2}n(n-1) \in O(n^2)$
  - $n^3 \notin O(n^2), 0.0001n^3 \notin O(n^2), n^4 + n + 1 \notin O(n^2)$

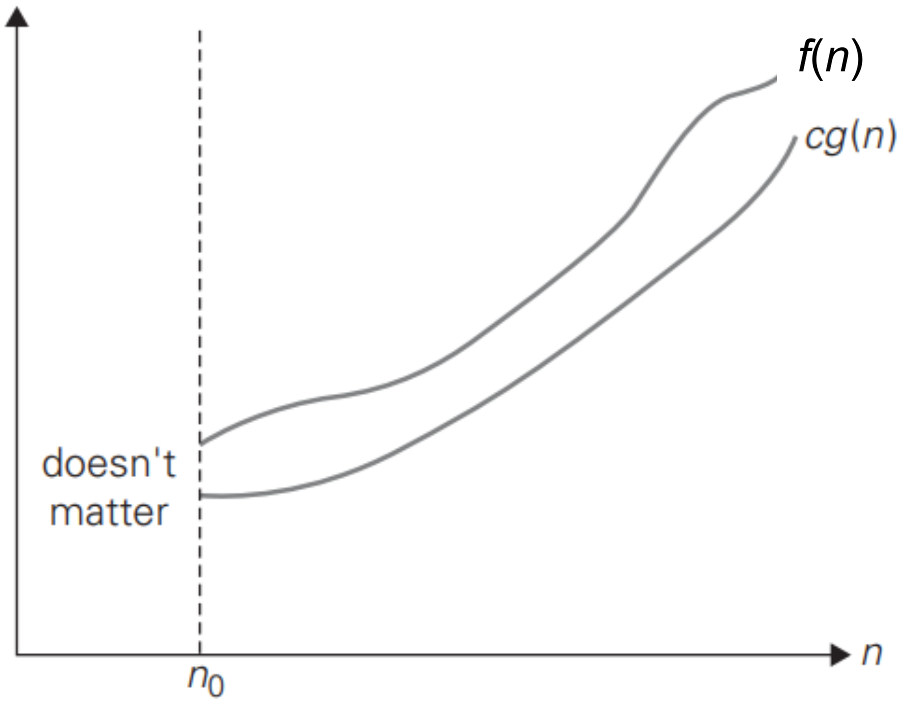
$\boldsymbol{f(n) \in O(g(n))}$



Asymptotic Notations – Big Omega

- $\boldsymbol{\Omega(g(n))}$: set of all functions with a **higher** or **same** order of growth as $g(n)$
  - E.g., $n^3 \in \Omega(n^2)$, $\frac{1}{2}n(n-1) \in \Omega(n^2)$
  - $100n + 5 \notin \Omega(n^2)$

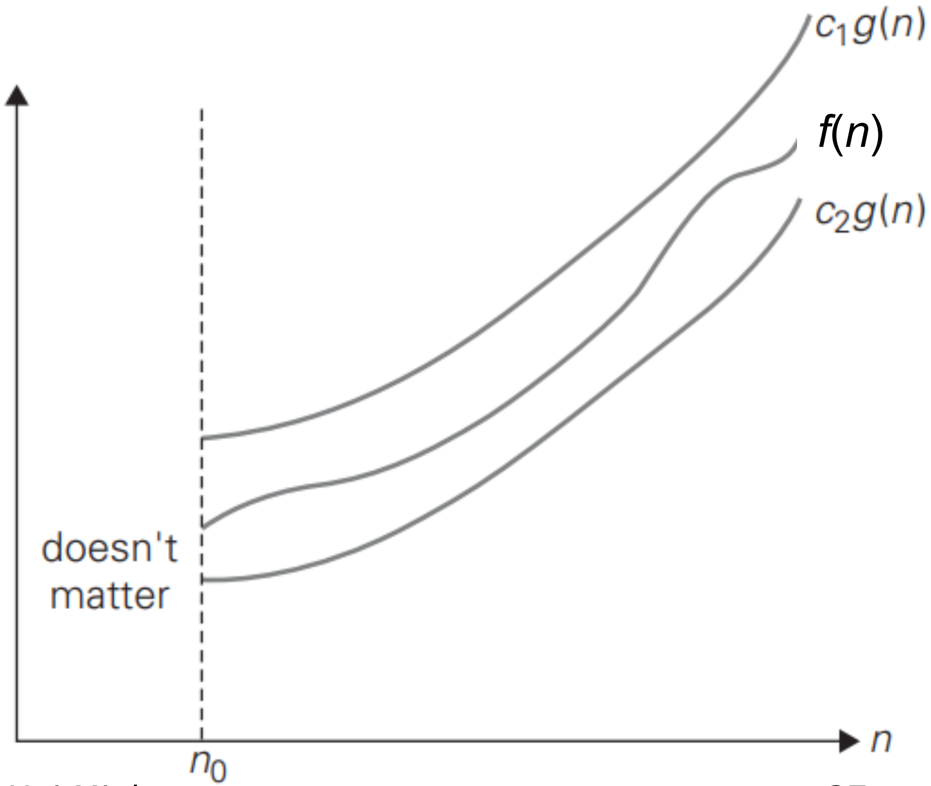
$\boldsymbol{f(n) \in \Omega(g(n))}$



Asymptotic Notations – Big-Theta

- $\boldsymbol{\Theta}\big(\boldsymbol{g}(\boldsymbol{n})\big)$: set of all functions with **same** order of growth as $g(n)$
  - E.g., $an^2 + bn + c \in \Theta(n^2)$ with $a > 0$
  - $n^3 + \log n \notin \Theta(n^2)$

$$\boldsymbol{f}(\boldsymbol{n}) \in \boldsymbol{\Theta}(\boldsymbol{g}(\boldsymbol{n}))$$



O-Notation

- **_Math:_**
  - For a given function $g(n)$, we denote by $O\big(g(n)\big)$ (pronounced "big-oh of *g* of *n*") the set of functions

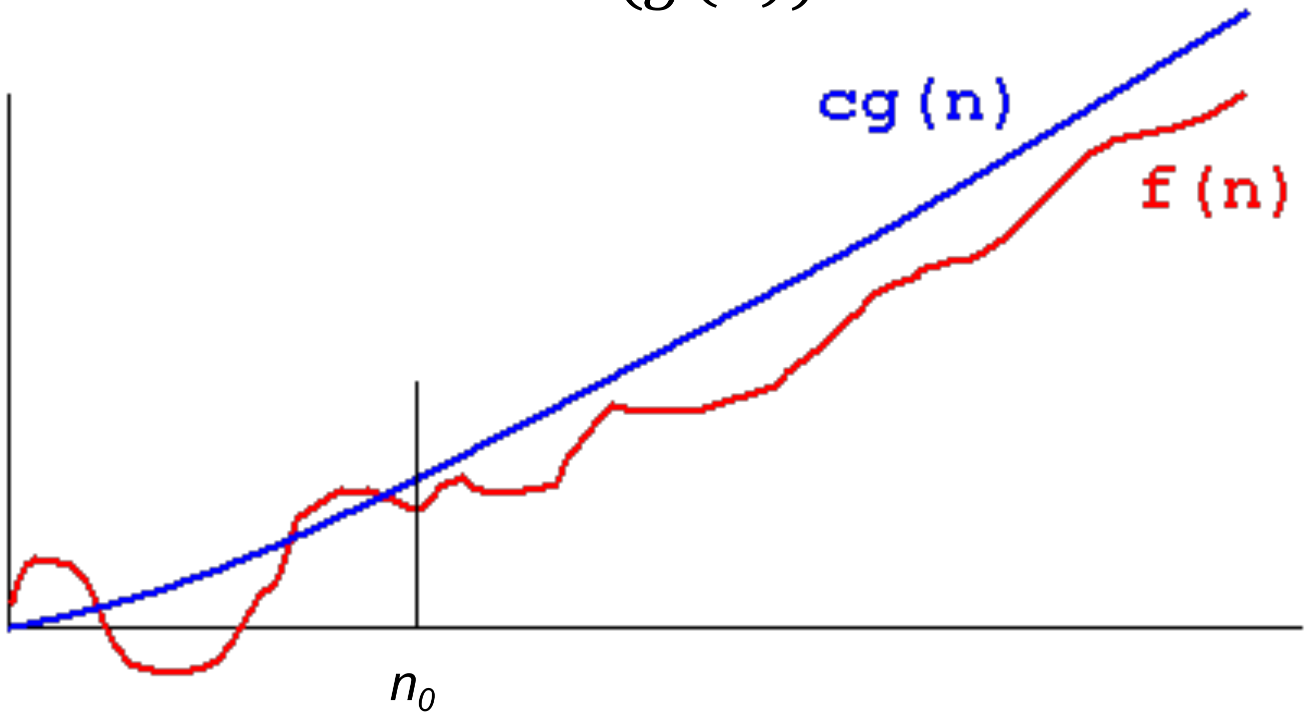
$O\big(g(n)\big) = \{f(n)$: there exist positive constants c and n_o such that: $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$

  - Explain: f is big-O of g if there is c so that f is not bigger than $c * g$ when n is large enough

- **_Engineering:_**
  - Drop low-order terms, ignore leading constants.
  - Ex: $3n^3 + 90n^2 - 5n + 6046 = O(n^3)$

O-Notation

- If n is large enough $(n \geq n_0)$, then $g(n)$ is the upper bound of $f(n)$
- We write $f(n) \in O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$



O-Notation

- O-notation is used to classify algorithms by how they respond to changes in input size.
- O-notation characterizes functions according to their growth rates:
  - different functions with the same growth rate may be represented using the same O-notation.

O-Notation – Example

- **Prove that $\boldsymbol{f(n) = 2n^2 + 6n + 1 \in O(n^2)}$**
  - Let $g(n) = n^2$
  - We have: $2n^2 + 6n + 1 \le 2n^2 + 6n^2 + n^2 = 9n^2$ (for all $n \ge 1$)
  - Thus, as $c = 9, n_0 = 1 \rightarrow f(n) < 9g(n)$
  - By definition of Big-Oh, $f(n) \in O(n^2)$

- Note that you can choose other specific values for constants c and n_0.
  - For example, we can choose $c = 3, n_0 = 7$

O-Notation – Example

- **Prove that $\boldsymbol{f(n) = n^3 - 100n^2 \notin O(n^2)}$**
  - If we have $f(n) \in O(n^2)$, then there would be positive constants c and n_0 such that
  - $n^3 - 100n^2 \leq cn^2$ (for all $n \geq n_0$)
  - We divide both sides by n^2, giving $n - 100 \leq c$
  - Regardless of what value we choose for c, this inequality does not hold for any value of $n > c + 100$

Classification of Algorithms

Order of growth	Class name
$O(1)$	Constants
$O(\log_2 n)$	Logarithms
$O(n)$	Linears
$O(n\log_2 n)$	$n\log_2 n$
$O(n^a)$	Polynomials
$O(a^n)$, $a > 1$	Exponentials
$O(n!)$	Fractorials

MATHEMATICAL ANALYSIS OF ALGORITHMS

Non-recursive Algorithms

Recursive Algorithms

ANALYSIS OF NON-RECURSIVE ALGORITHMS

1. Decide ***n*** – the input size
2. Identify the algorithm's **basic operation** (as a rule, it is located in the innermost loop)
3. Check whether the number of times the basic operation is executed depends only on *n*
   - If it depends on some additional property, specify the **worst-case** for Big-Oh
4. Set up a **sum** expressing the number of times the algorithm's basic operation is executed.
5. Find a closed-form formula for the count and establish its **order of growth**.

ANALYSIS OF NON-RECURSIVE ALGORITHMS

- **Example:** Check whether all the elements in a given array of *n* elements are distinct.

```
UniqueElements(A[0..n – 1])
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns “true” if all the elements in A are distinct
// and “false” otherwise
 for i ← 0 to n – 2 do
   for j ← i + 1 to n – 1 do
     if A[i] = A[j]
       return false
 return true
```

Basic operation

ANALYSIS OF NON-RECURSIVE ALGORITHMS

- Worst-case:

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2$$

ANALYSIS OF RECURSIVE ALGORITHMS

1. Decide ***n*** – the input size
2. Identify the algorithm's **basic operation**
3. Check whether the number of times the basic operation is executed depends only on *n*
    1. If it depends on some additional property, specify the **worst-case** for Big-Oh
4. Set up a **recurrence relation**, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence and establish its **order of growth**.

ANALYSIS OF RECURSIVE ALGORITHMS

- **Example:** Compute the factorial function $F(n) = n!$ for an arbitrary non-negative integer n.

```
Factorial(n)
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
 if n = 0 return 1
 else return Factorial(n – 1) * n
```

Basic operation

ANALYSIS OF RECURSIVE ALGORITHMS

- Recurrence relation:

$$M(n) = \underset{\substack{\text{to compute} \\ F(n-1)}}{M(n-1)} + \underset{\substack{\text{to multiply} \\ F(n-1) \text{ by } n}}{1} \quad \text{for } n > 0.$$

  - We have: M(0) = 0. Thus:
  - M(n) = M(n − 1) + 1 = [M(n − 2) + 1] + 1 = M(n − 2) + 2
  - = [M(n − 3) + 1] + 2 = M(n − 3) + 3.

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

What's next?

- After today:
  - Read textbook 1 – section 1.3 (page 85~)
  - Read textbook 3 – chapter 1 & 2 (page 1~)
- Next Week:
  - Quiz 1 (20 mins, from 7:30~)
  - Lecture 2: Sorting Algorithms

Q&A