

Lecture 2

Analysis of Algorithms

The need for running time analysis



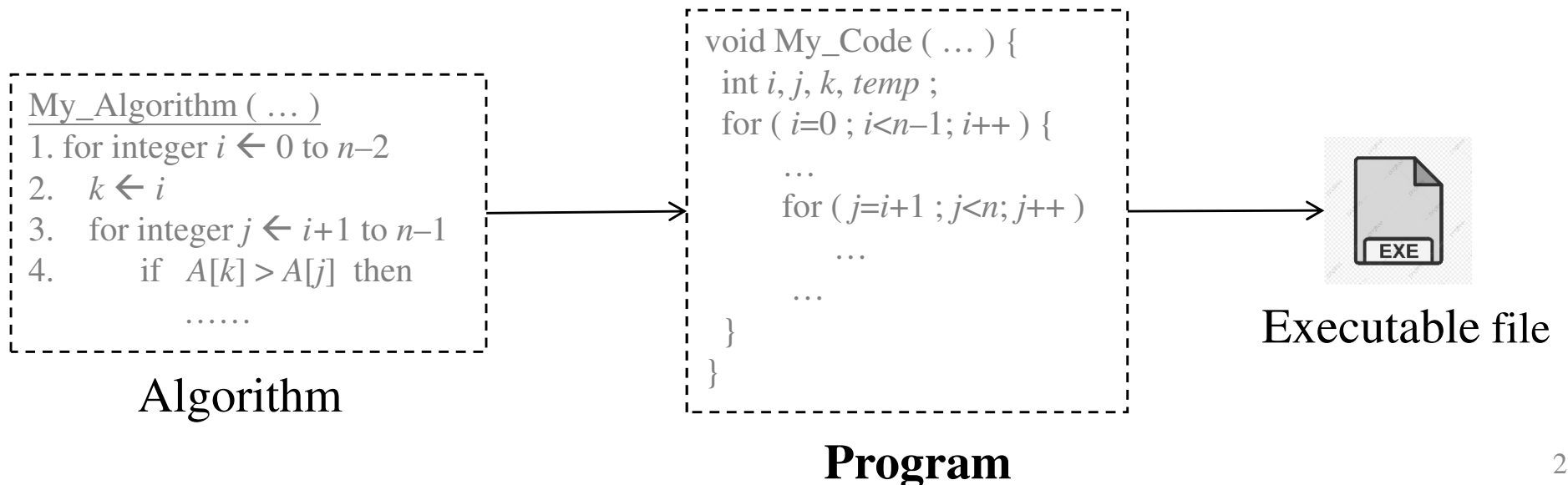
Your boss

Write the fastest program to solve problem X!

This problem can be solved by several different algorithms (e.g., A, B, C).

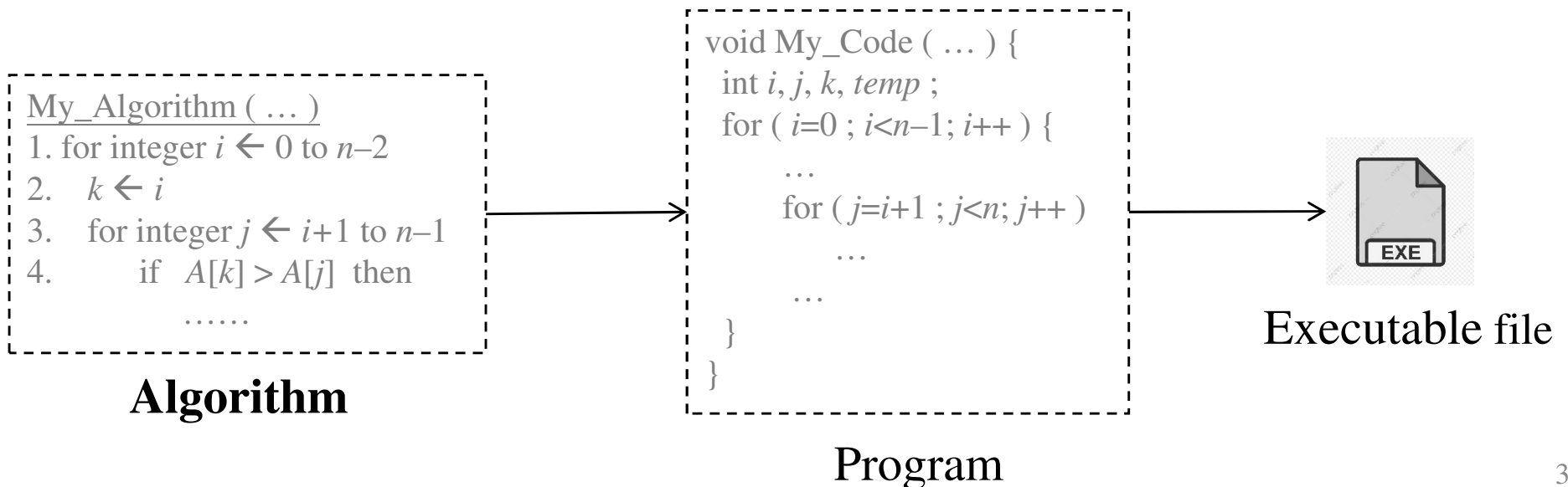
Which one is the fastest?

- ◆ Time consuming to write several programs and run them!
- ◆ Before writing a **program**, how to estimate its running time?



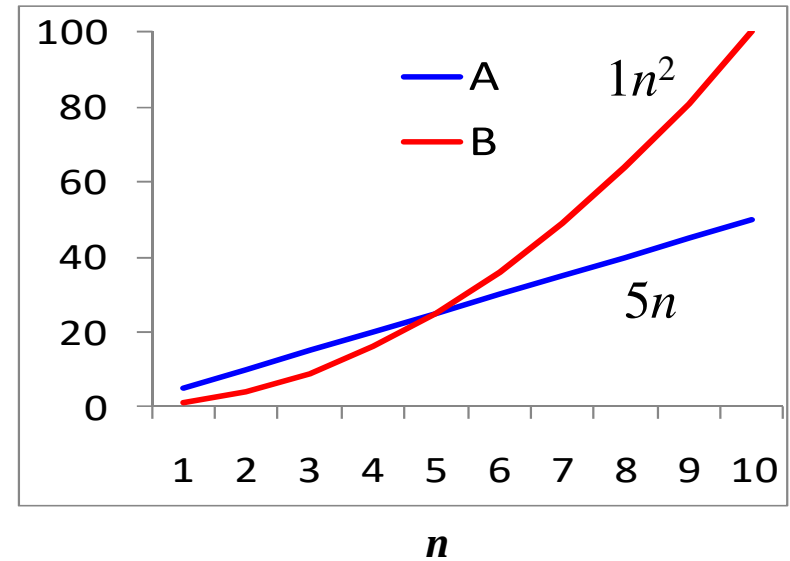
The need for running time analysis

- ◆ Before writing a program, how to estimate its running time?
- ◆ How to estimate the running time of an **algorithm**?
- ◆ How does the running time of an algorithm $T(n)$ vary with the input data size n ?
 - ◆ E.g., linear to n ? sublinear to n ? superlinear to n ?



Order of growth

time



Function	n=16	n=1024	n=1000000
$\log_2 n$	4	10	~20
$\sqrt{n}$	4	32	1000
n	16	1024	1000000
$n \lg n$	64	10240	~20000000
n^2	256	1048576	10^{12}
n^3	4096	$\sim 10^9$	10^{18}
2^n	65536	... too large	... too large ₄

Our Roadmap



- ◆ Our first attempt on running time analysis
 - ◆ *Just to help you understand the remaining parts*
- ◆ What is asymptotic running time?
- ◆ Running time of recursive algorithms

Assumptions for Running Time Analysis

- ◆ To simplify the analysis, we make the following assumptions on the CPU and the memory
- ◆ 1. Each CPU instruction takes constant time c_j
 - ◆ Examples of instructions:
 - ◆ Add two integers
 - ◆ Compare two variables/values
 - ◆ Load/Store copy a value from/to a variable
- ◆ 2. CPU executes instructions one-by-one
- ◆ 3. The memory is large enough to store all input/intermediate/output data



How to estimate running time?

- ◆ Find the time cost of each line
 - ◆ Each line uses a constant number of instructions
 - ◆ Let c_j be the time cost of line j
- ◆ Find the frequency of executing each line
 - ◆ Each line is executed once
- ◆ Running time of the algorithm
$$= c_1 * 1 + c_2 * 1 + c_3 * 1$$
$$= c_1 + c_2 + c_3$$

<u>FunS (Integer p)</u>	<u>$cost$</u>	<u>$frequency$</u>
1. $k \leftarrow p + p$	c_1	1
2. $m \leftarrow p * p$	c_2	1
3. $x \leftarrow m - k$	c_3	1

How to estimate running time?

- ◆ Let c_j be the time cost of line j
 - ◆ because each line uses a constant number of instructions
- ◆ Find the frequency of executing each line
 - ◆ Each line is executed n times
- ◆ Running time of the algorithm
$$= c_1 * n + c_2 * n + c_3 * n$$
$$= (c_1 + c_2 + c_3) n$$

<u>FunI (Integer n)</u>	<u>$cost$</u>	<u>$frequency$</u>
1. for integer $i \leftarrow 1$ to n	c_1	n
2. $k \leftarrow i * i$	c_2	n
3. print k	c_3	n

How to estimate running time?

◆ Analysis

- ◆ Lines 1 and 2 are always executed
- ◆ Based on the comparison result (at Line 2), either Line 3 or 5 will be executed

◆ Worst-case running time of the algorithm

We consider the worst-case here!

$$= c_1 + c_2 + \max(c_3, c_5)$$

FunC (Integer a , b)

1. $c \leftarrow 0$

2. if $a < b$ then

3. $c \leftarrow a$

4. else

5. $c \leftarrow b$

cost

frequency

c_1

1

c_2

1

c_3

at most 1

c_5

at most 1

How to estimate running time?

- ◆ Find the frequency of executing Line 1 (& Line 2)
 - ◆ It depends on the array position that contains the result
 - ◆ Executed **at most** n times
- ◆ The algorithm terminates after it executes a return statement (e.g., Line 3 or Line 4)
- ◆ **Worst-case** running time of the algorithm

$$= c_1 * n + c_2 * n + c_3 * 1 + c_4 * 1$$

$$= (c_1 + c_2) n + c_3 + c_4$$

*[Note] A more accurate result:
 $(c_1 + c_2)n + \max(c_3, c_4)$*

Search (Array $A[0..n-1]$, Key k)

1. for integer $i \leftarrow 0$ to $n-1$

2. if $A[i] = k$

3. return i

4. return -1

5 | 2 | 4 | 9 | 7

<u>cost</u>	<u>frequency</u>
c_1	at most n
c_2	at most n
c_3	at most 1
c_4	at most 1 ¹⁰

Worst case input

Search (Array $A[0..n-1]$, Key k)

1. for integer $i \leftarrow 0$ to $n-1$
2. if $A[i] = k$
3. return i
4. return -1

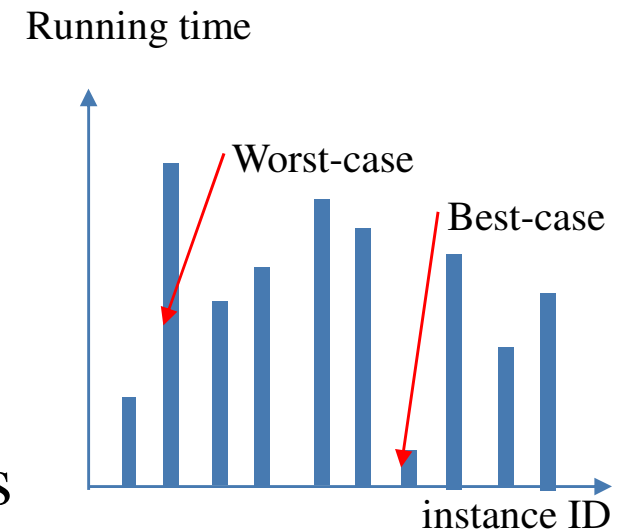
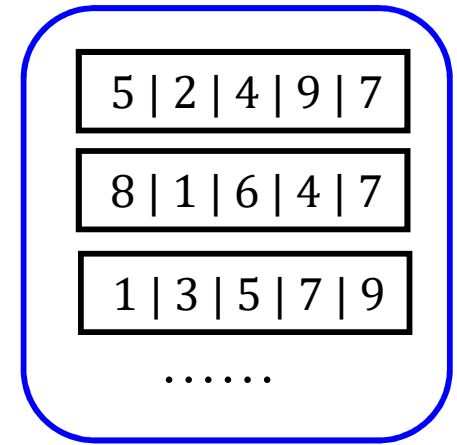
- ◆ Even if the input size is fixed, the running time of an algorithm depends on the input

◆ Exercises

- ◆ The input of the Search algorithm: array A and key k
- ◆ Suppose that the input size is fixed at $n = 5$
- ◆ (Q1) Given the array $A = \boxed{5 \mid 2 \mid 4 \mid 9 \mid 7}$,
find a key k to obtain the worst-case input
- ◆ (Q2) Given the key $k = 8$,
find an array A to obtain the worst-case input

Running time

- ◆ There are many possible **input instances**
- ◆ We focus on the worst-case running time of an algorithm
 - ◆ The longest running time for any input of the same size n
- ◆ The running time involves many constants $c_1, c_2, c_3, \dots$
 - ◆ E.g., $(c_1 + c_2)n + c_3 + c_4$
 - ◆ Especially for an algorithm with many lines
- ◆ How to simplify the running time expression and still obtain the **trend** of the running time?
 - ◆ E.g., linear to n ? sublinear to n ? superlinear to n ?



Trend of running time

- Running time of two algorithms

- Algorithm A: $5n$

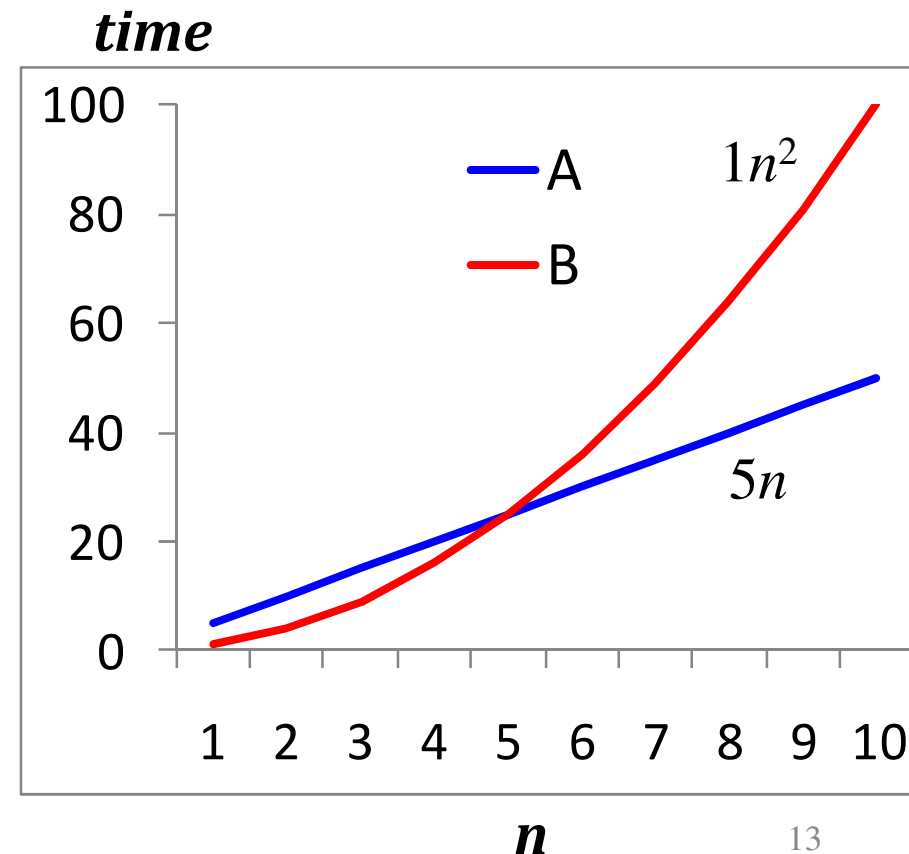
- Algorithm B: $1n^2$

- Which algorithm is faster?

- Algorithm A. Why?

- We care about the performance at *large input size*

- Constant factors do not affect the *order of growth*



Our Roadmap

~~◆ Our first attempt on running time analysis~~



◆ What is asymptotic running time?

◆ Running time of recursive algorithms

Asymptotic Notation

- ◆ *Asymptotic notation* describes the trend of running time in a concise way, for example,

$$\diamond c_1 + c_2 + c_3 \quad \rightarrow \quad O(1)$$

$$\diamond (c_1 + c_2 + c_3) n \quad \rightarrow \quad O(n)$$

$$\diamond (c_1 + c_2) n + c_3 + c_4 \quad \rightarrow \quad O(n)$$

- ◆ We will cover these asymptotic notation
 - ◆ Asymptotic upper-bound, O -notation
 - ◆ Asymptotic lower-bound, Ω -notation
 - ◆ Asymptotic tight-bound, Θ -notation

Asymptotic Notation

◆ Asymptotic **upper-bound**, **O**-notation

◆ We write **$f(n)$ is $O(g(n))$** if

there exist positive constants c, n_0 such that

$$0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0$$

◆ $(5n^2 + 3n)$ is $O(n^2)$?

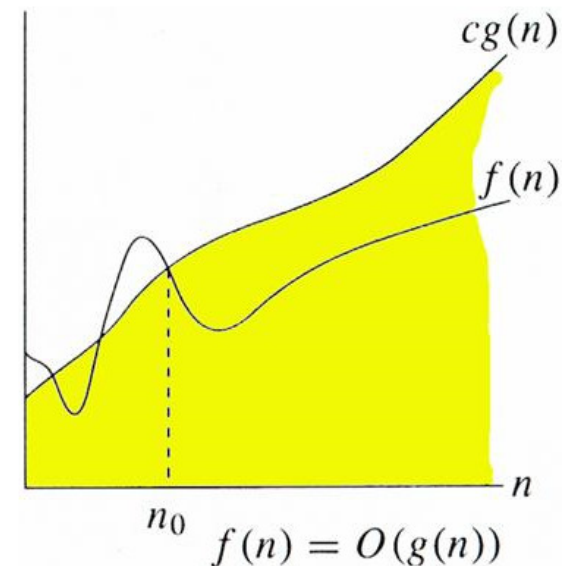
$$\begin{aligned} &5n^2 + 3n \\ &\leq 5n^2 + 3n^2 \quad [\text{true for all } n \geq 1] \\ &\leq 8n^2 \end{aligned}$$

◆ YES, we choose $c=8$ and $n_0=1$.

◆ $(5n^2 + 3n)$ is $O(n^3)$?

◆ $(5n^2 + 3n)$ is $O(n)$?

◆ NO. *Hint*: use “Proof by contradiction”.



Asymptotic Notation

- Asymptotic **lower-bound**, Ω -notation

- We write **$f(n)$ is $\Omega(g(n))$** if

there exist positive constants c, n_0 such that

$$0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0$$

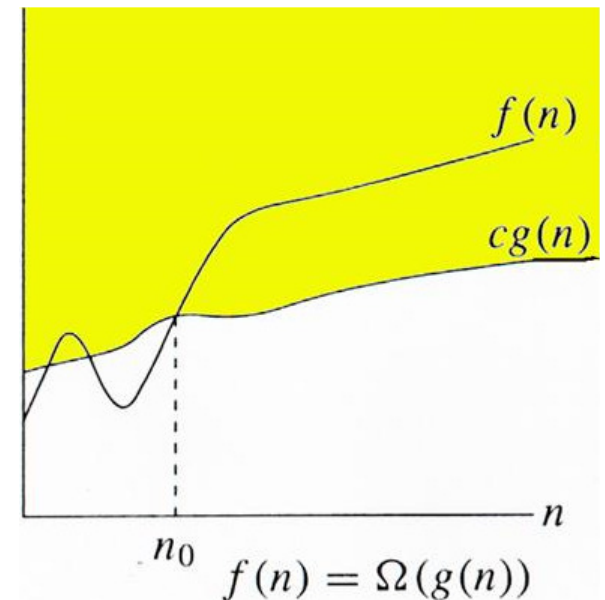
- $(5n^2 + 3n)$ is $\Omega(n^2)$?

$$\begin{aligned} 5n^2 + 3n &\geq 5n^2 && [\text{true for all } n \geq 1] \\ &= 5n^2 \end{aligned}$$

- YES, we choose $c=5$ and $n_0=1$.

- $(5n^2 + 3n)$ is $\Omega(n^3)$?

- $(5n^2 + 3n)$ is $\Omega(n)$?



Asymptotic Notation

- Asymptotic **tight-bound**, Θ -notation

- We write **$f(n)$ is $\Theta(g(n))$** if

there exist positive constants c_l, c_h, n_0 such that

$$0 \leq c_l g(n) \leq f(n) \leq c_h g(n) \quad \text{for all } n \geq n_0$$

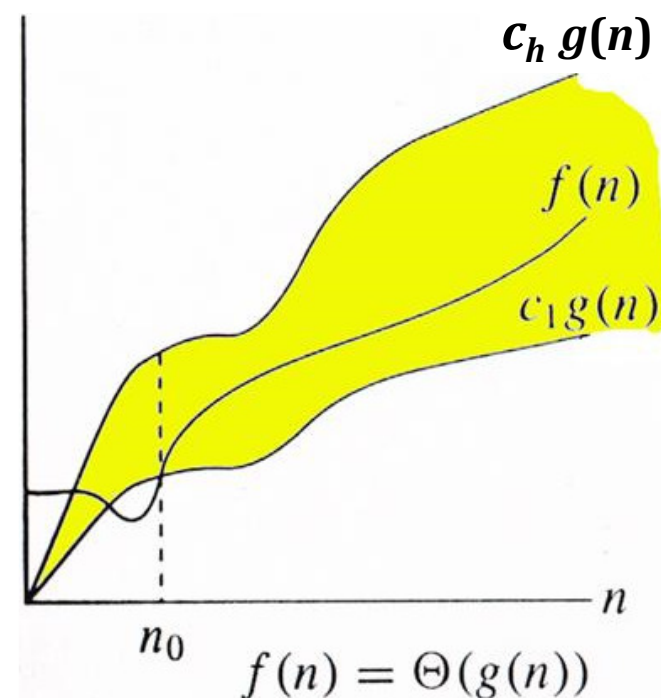
- $(5n^2 + 3n)$ is $\Theta(n^2)$?

- YES, we choose $c_l=5, c_h=8$ and $n_0=1$.

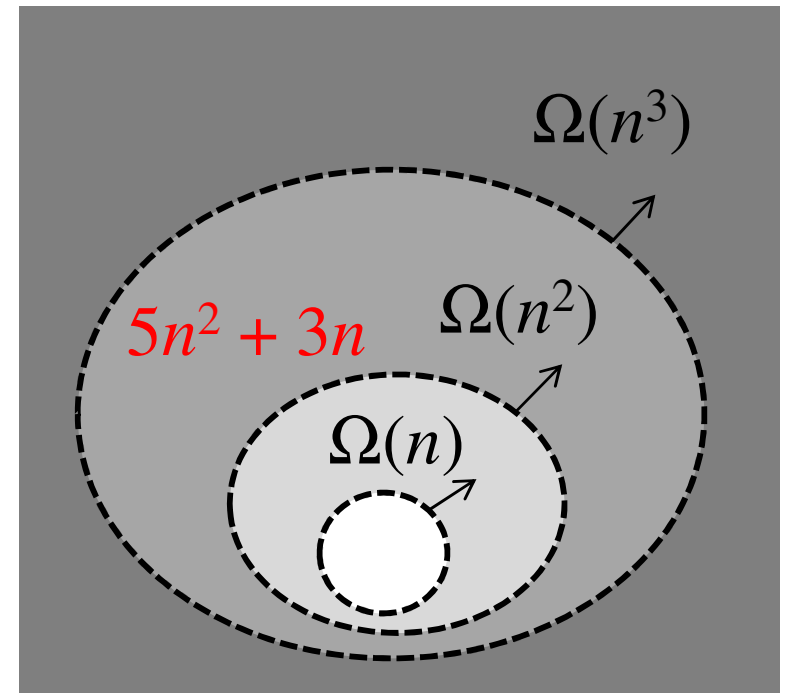
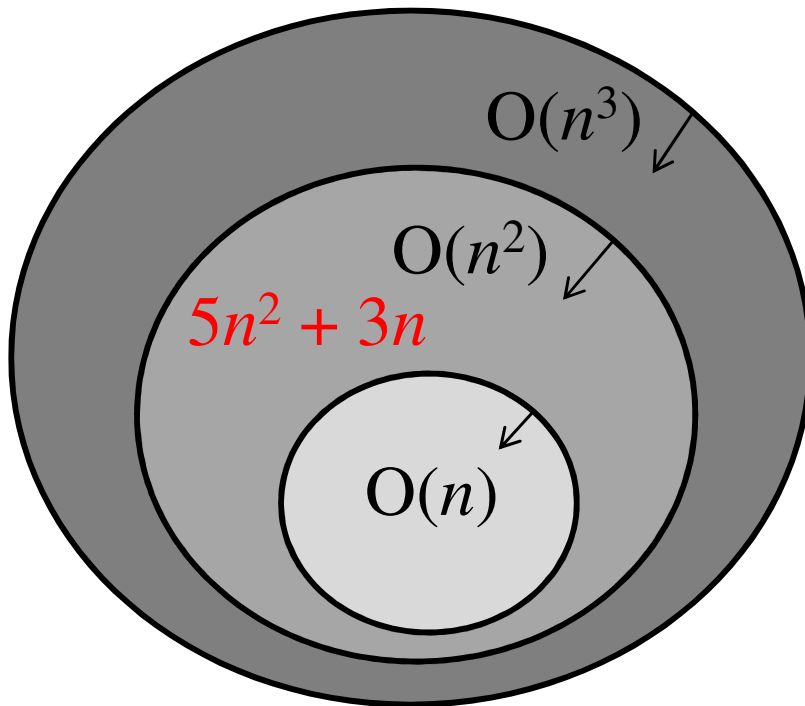
[see the proofs on the previous two pages]

- $(5n^2 + 3n)$ is $\Theta(n^3)$?

- $(5n^2 + 3n)$ is $\Theta(n)$?



Graphical View of Asymptotic Notation

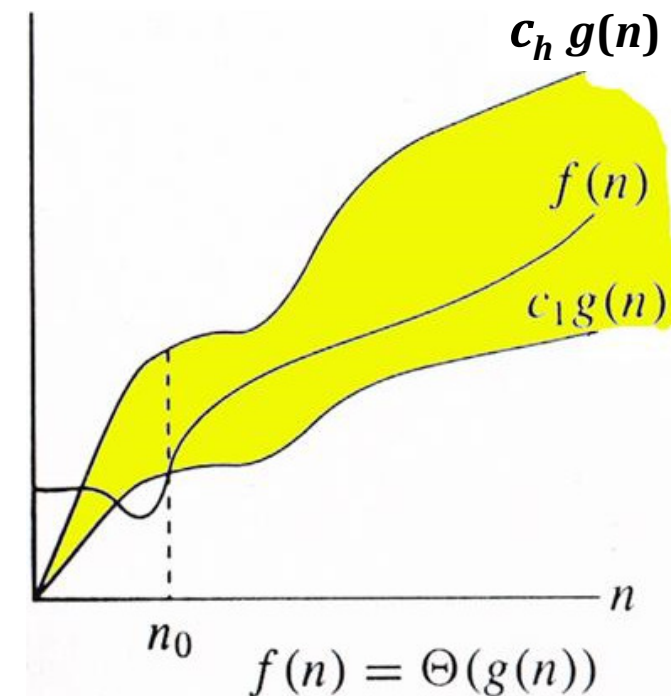


Reminder

- ◆ Upper-bound notation: $O(\dots)$
- ◆ Lower-bound notation: $\Omega(\dots)$

Asymptotic Notation

- Let $T(n)$ be the running time of an algorithm, where the input size is n
- What does “ $T(n)$ is $O(n)$ ” mean?
- What does “ $T(n)$ is $\Omega(n)$ ” mean?
- What does “ $T(n)$ is $\Theta(n)$ ” mean?



Simplification Rules of O-Notation

- ◆ Ignore constant factors

- ◆ E.g., 3 is $O(1)$

- ◆ E.g., $3n$ is $O(n)$

*These rules provide
shortcuts for showing that
 $f(n)$ is $O(g(n))$*

- ◆ Combine fixed number of terms with same complexity

- ◆ E.g., $O(n) + O(n)$ is $O(n)$ $\longleftrightarrow c_1 n + c_2 n = (c_1 + c_2) n$

- ◆ E.g., $O(n) + O(n) + O(n)$ is $O(n)$

- ◆ Polynomial

- ◆ Just use the highest-degree term.

Why?

- ◆ E.g., $5n^2 + 3n$ is $O(n^2)$

- ◆ E.g., $2n^3 + 5n^2 + 3n$ is $O(n^3)$

Simplification Rules of O-Notation

- ◆ Can we combine a variable number of terms with the same complexity?

- ◆ How do you simplify

$$\underbrace{O(n) + O(n) + \dots + O(n)}_{n \text{ terms}}$$

- ◆ Is it $O(n)$ or $O(n^2)$?

Asymptotic running time of some algorithms (for your reference)

<i>Complexity</i>		<i>Algorithm</i>
$O(1)$	Constant time	Compare two numbers
$O(\log n)$	Logarithmic	Binary search (on a sorted array)
$O(n)$	Linear time	Search (on a unsorted array)
$O(n \log n)$		Merge sort
$O(n^2)$	Quadratic	Selection sort
$O(n^3)$	Cubic	Matrix multiplication
$O(2^n)$	Exponential	Brute-force search on boolean satisfiability problem
$O(n!)$	Factorial	Brute-force search on traveling salesman problem

Asymptotic running time: Example 1

- ◆ Let's analyze the worst-case running time of this algorithm again

- ◆ This time we analyze its asymptotic running time!
- ◆ Line 1: at most n times basic steps = $O(n)$
- ◆ Line 2: at most $1 * n$ times basic steps = $O(n)$
- ◆ Lines 3 and 4: $O(1)$

- ◆ Worst-case running time $T(n)$

$$= O(n) + O(n) + O(1)$$

$$= O(n)$$

Search (Array $A[0..n-1]$, Key k)

1. for integer $i \leftarrow 0$ to $n-1$
2. if $A[i] = k$
3. return i
4. return -1

5 | 2 | 4 | 9 | 7

Asymptotic running time: Example 1

◆ Remarks

- ◆ The statement “ $O(n)$ is $O(n^2)$ ” is **correct**, according to the definition of the O-notation
- ◆ However, $O(n^2)$ is **not an accurate description** of the worst-case running time of this algorithm!
- ◆ In the analysis, try to make $O(\dots)$ as small as possible

Search (Array $A[0..n-1]$, Key k)

1. for integer $i \leftarrow 0$ to $n-1$
2. if $A[i] = k$
3. return i
4. return -1

Asymptotic running time: Example 2

◆ Analysis of *NestedLoop*

- ◆ Line 1: n times basic steps
- ◆ Line 2: $n*n$ times basic steps
- ◆ Line 3: same as Line 2

NestedLoop (Integer n)

1. for integer $i \leftarrow 1$ to n
2. for integer $j \leftarrow 1$ to n
3. print-line i, j

◆ Worst-case running time $T(n)$

$$= O(n) + O(n^2) + O(n^2)$$

$$= O(n^2)$$

Asymptotic running time: Example 3

Selection-Sort (Array $A[0..n-1]$)

1. for integer $i \leftarrow 0$ to $n-2$

2. $k \leftarrow i$

3. for integer $j \leftarrow i+1$ to $n-1$

4. if $A[k] > A[j]$ then

5. $k \leftarrow j$

6. swap $A[i]$ and $A[k]$

5 | 2 | 4 | 9 | 7

- ◆ Line 1: $n - 1$ basic steps $= O(n)$
- ◆ Line 2: $1 * (n - 1)$ basic steps $= O(n)$
- ◆ Lines 3 to 5:
 - ◆ What are possible *initial values* of j (before Line 3)?
 - ◆ For each initial value, how many times can we execute Line 3 at most?



Asymptotic running time: Example 3

Selection-Sort (Array $A[0..n-1]$)

1. for integer $i \leftarrow 0$ to $n-2$
2. $k \leftarrow i$
3. for integer $j \leftarrow i+1$ to $n-1$
4. if $A[k] > A[j]$ then
5. $k \leftarrow j$
6. swap $A[i]$ and $A[k]$

◆ Analysis of Line 3:

- ◆ Values of j in that loop: 1 to $n-1$, 2 to $n-1$, ..., $n-1$ to $n-1$
- ◆ Number of iterations: $n-1, n-2, \dots, 1$
- ◆ $\sum_{i=0..n-2} ((n-1)-(i+1)+1) = \sum_{i=0..n-2} (n-i-1) = (n-1)*n/2 = O(n^2)$

◆ Worst-case running time $T(n)$:

- ◆ Line 1. $n-1$ basic steps = $O(n)$
- ◆ Lines 2, 6. same as Line 1
- ◆ Line 3. $O(n^2)$
- ◆ Lines 4, 5. same as Line 3
- ◆ Thus, we have: $T(n) = 3*O(n) + 3*O(n^2) = O(n^2)$

As accurate as possible for running time analysis

- ◆ The worst-case running time of Selection-Sort is $O(n^2)$
- ◆ The worst-case running time of Selection-Sort is $O(n^3)$
 - ◆ This is correct but loose
- ◆ In the analysis, try to make $O(\dots)$ as small as possible

Our Roadmap

~~◆ Our first attempt on running time analysis~~

◆ What is asymptotic running time?



◆ Running time of recursive algorithms

Asymptotic running time: Function Call

- ◆ Running time of *Fun*: $O(n)$
- ◆ Running time of *Main*
 - ◆ Line 1: $O(n)$
 - ◆ Lines 2, 3: $O(1)$
 - ◆ Thus, $T(n)$
 $= O(n) + O(1) + O(1)$
 $= O(n)$

Main (Integer n)

1. **Fun** (n)
2. $k \leftarrow n * n$
3. **print** k

Fun (Integer n)

1. **for** integer $i \leftarrow 1$ **to** n
2. $k \leftarrow i * i$
3. **print** k

Binary Search

Initial call: BinarySearch (A, 0, $n-1$, k)

Requirement: array A sorted
in the ascending order

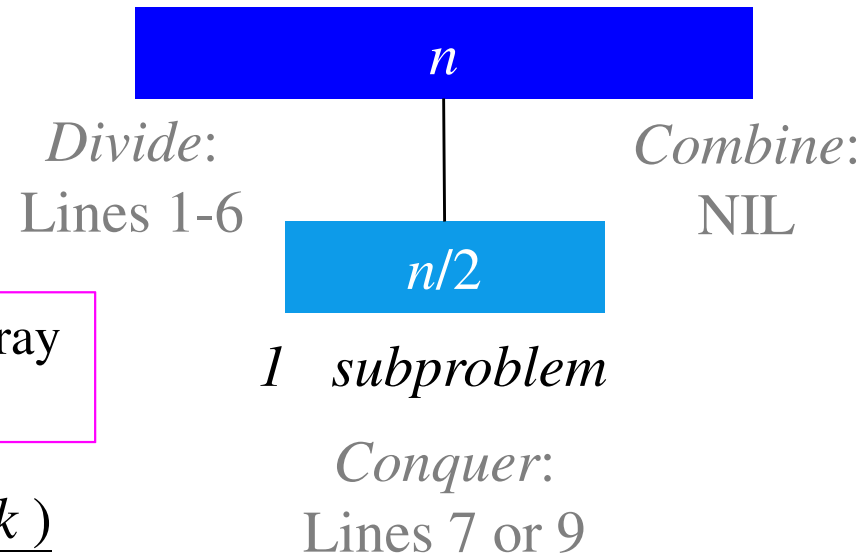
considers the sub-array
 $A[low, high]$ only

BinarySearch(Array A , Integers low , $high$, Key k)

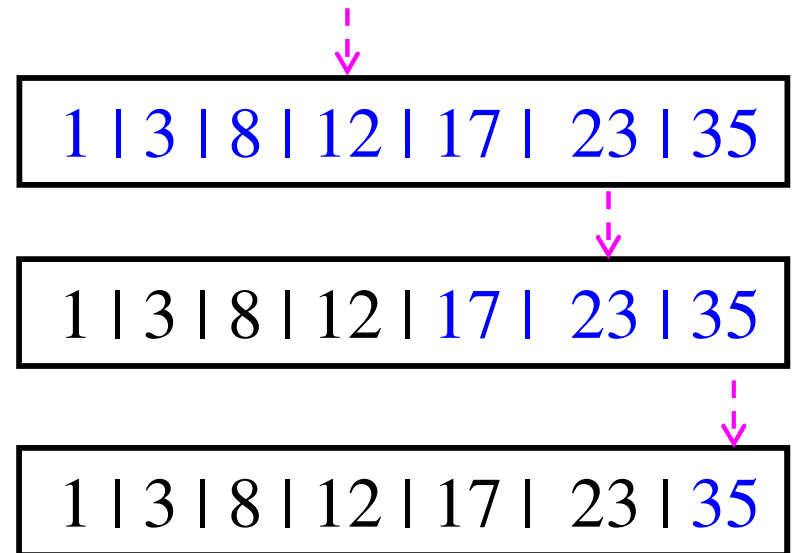
1. $mid \leftarrow \lfloor (low+high)/2 \rfloor$
2. if $A[mid] = k$
3. return mid
4. else if $low = high$
5. return -1
6. if $k < A[mid]$
7. return **BinarySearch**(A , low , $mid-1$, k)
8. else
9. return **BinarySearch**(A , $mid+1$, $high$, k)

Recursive call on the
1st part or the 2nd part

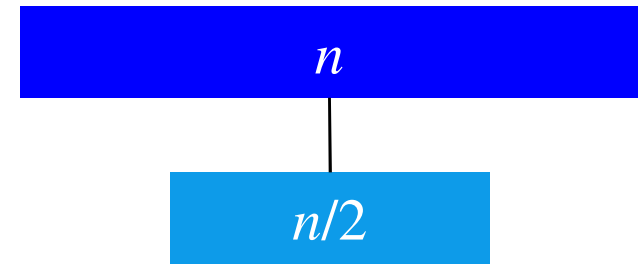
Return the key position;
or -1 (if not found)



Example: find the key “35”



Binary Search: running time analysis



BinarySearch (Array A , Integers low , $high$, Key k)

1. $mid \leftarrow \lfloor (low+high)/2 \rfloor$
2. if $A[mid] = k$
3. return mid
4. else if $low = high$
5. return -1
6. if $k < A[mid]$
7. return BinarySearch(A , low , $mid-1$, k)
8. else
9. return BinarySearch(A , $mid+1$, $high$, k)

Initial call: BinarySearch (A , 0 , $n-1$, k)

1 subproblem *Conquer:*
Lines 7 or 9

Running time analysis

Let $T(n)$ be the running time of BinarySearch, where n is the size of the input sub-array $A[low..high]$.

Lines 1-6: $O(1)$ time

Either Line 7 or Line 9 is executed

Line 7: $T(n/2)$ time because the sub-array interval shrinks by half

Line 9: same as Line 7

We obtain the recurrence:

$$T(n) = O(1) + T(n/2)$$

How do we solve $T(n)$?

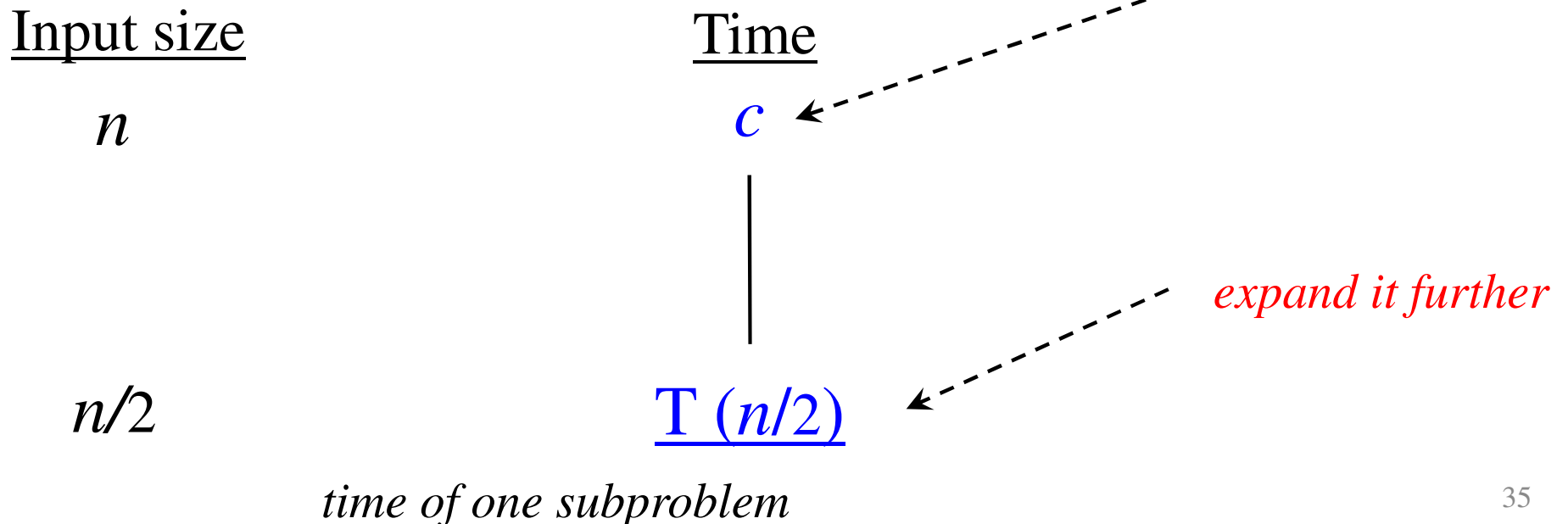
Solving a Recurrence

- ◆ A *recurrence* is a recursive equation
 - ◆ E.g., $T(n) = T(n/2) + O(1)$
 - ◆ However, it does not directly tell us the asymptotic running time
 - ◆ We need to solve it ...
- ◆ Use the *recursion tree* method to solve recurrence
 - ◆ Draw a tree of the time used at each level of problem
 - ◆ Sum up the times used at all levels
- ◆ Let's try to solve a simple recurrence today
 - ◆ More details will be discussed in the next lecture

Solving: Recursion Tree

- ◆ We first convert $T(n) = T(n/2) + O(1)$ to:
$$T(n) = T(n/2) + c \quad (\text{where } c \text{ is a constant})$$

- ◆ Draw a recursion tree by expanding $T(n)$



Recursion Tree for: $T(n) = T(n/2) + c$

Input size

n

$n/2$

$n/2^2$

Time

c

c

$T(n/2^2)$

expand it further

Recursion Tree for: $T(n) = T(n/2) + c$

Input size

Time

- Time at the **leaf** level: c
- Time at each **non-leaf** level: c
- Let L be the number of **non-leaf** levels

We have: $n / 2^L = 1$

$\rightarrow 2^L = n$

$\rightarrow L = \log_2 n$

- Total time $T(n) = c L + c$
 $= c (\log_2 n) + c$
 $= O (\log_2 n)$

n

c

$n/2$

c

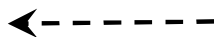
$n/2^2$

c

.....

1

c



*A constant-size problem
also takes constant time*

Summary

- ◆ *Asymptotic notation:* $O(\dots)$, $\Omega(\dots)$, $\Theta(\dots)$
- ◆ *Analyze* the asymptotic running time of algorithm
- ◆ Solve a recurrence by the *recursion tree method*
- ◆ Please read Chapters 2.2 and 3 in the textbook