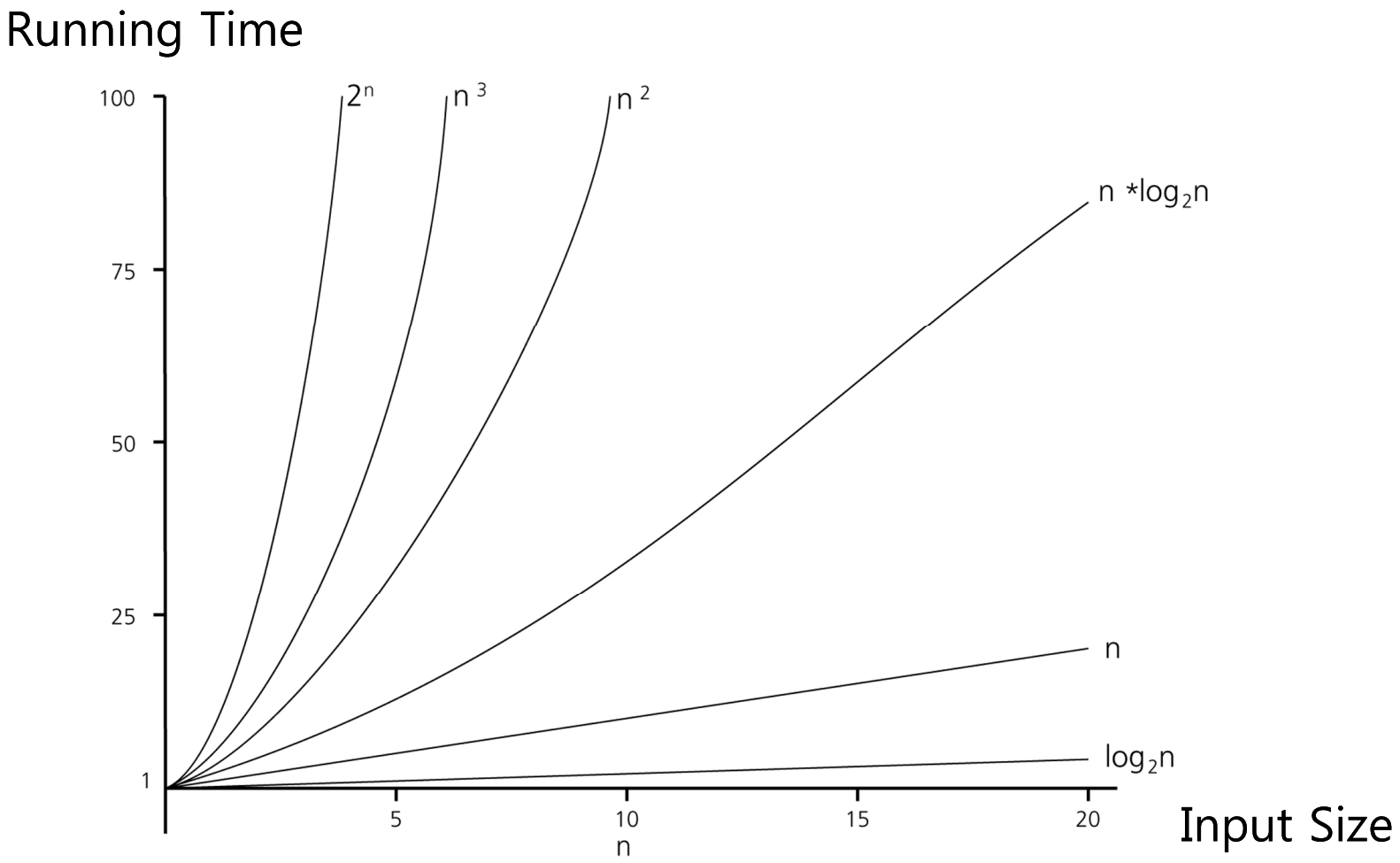
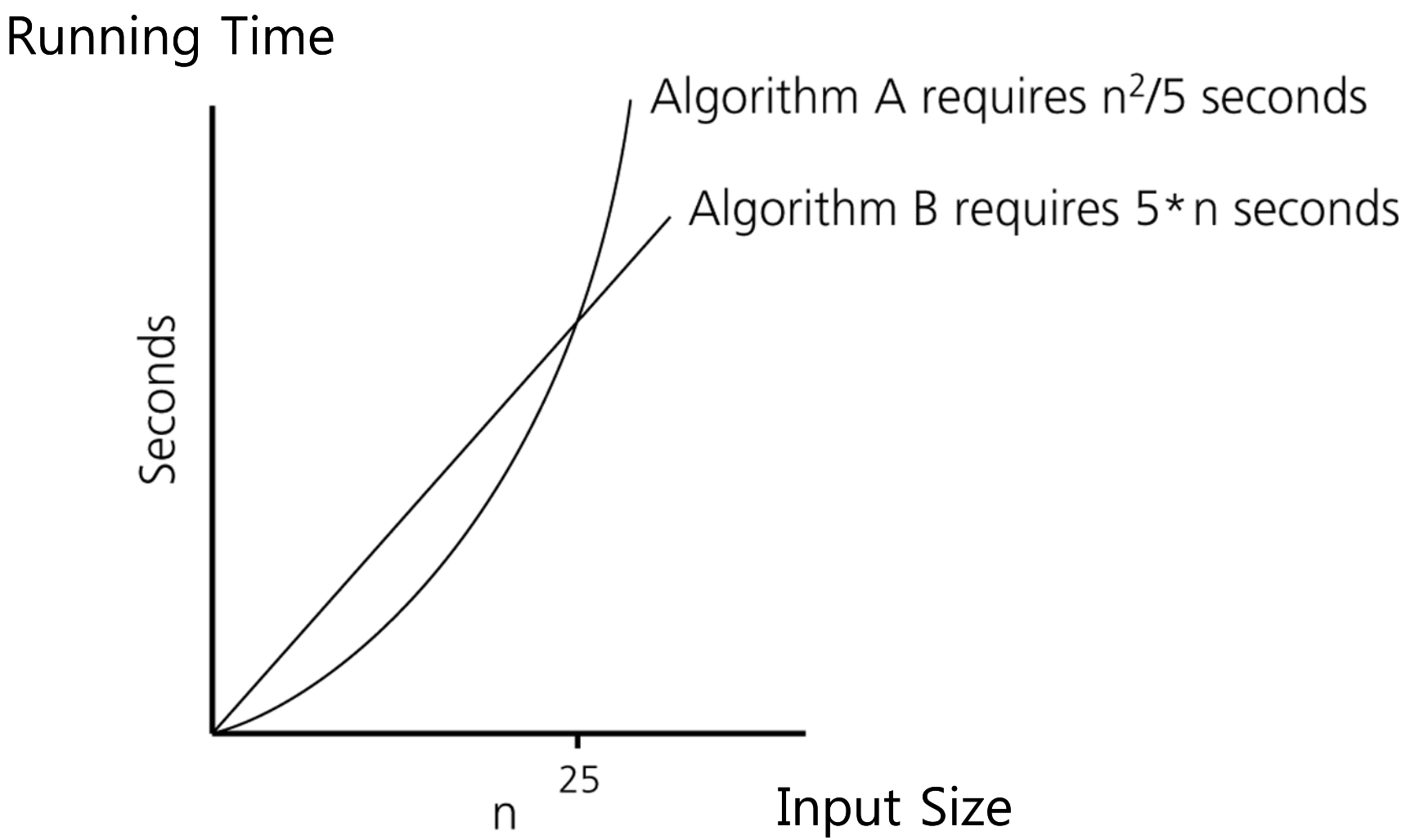
Asymptotic Complexity

Running Time



Running Time



Running Time

	n					
Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Running Time

- How do you count the running time?
  - Ex: # of iterations in for loop, # of executions of a line, # of function calls, etc.
- Check the following examples.

Running Time

```
sample1(A[ ], n)
{
        k = n/2 ;
        return A[k] ;
}
```

✓ Constant time regardless of n

Running Time

```
sample2(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                sum← sum+ A[i] ;
        return sum ;
}
```

✓ Time proportional to n

Running Time

```
sample3(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                for j ← 1 to n
                        sum← sum+ A[i]*A[j] ;
        return sum ;
}
```

✓ Time proportional to n^2

Running Time

```
sample4(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                for j ← 1 to n {
                        k ← Max of n/2 elements randomly drawn
    from A[1 ... n];
                        sum ← sum + k ;
                }
        return sum ;
}
```

✓ Proportional to n^3

Running Time

```
sample5(A[ ], n)
{
        sum ← 0 ;
        for  i ← 1 to n
                for  j ←  i+1 to n
                        sum← sum+ A[i]*A[j] ;
        return sum ;
}
```

✓ Proportional to n^2

Running Time

```
factorial(n)
{
        if (n=1) return 1 ;
        return n*factorial(n-1) ;
}
```

✓ Proportional to n

Recurrence and Inductive Thinking

- Recursive structure
  - A problem contains its smaller version.
  - Ex 1: factorial
    - $N! = N \times (N-1)!$
  - Ex 1:
    - $a_n = a_{n-1} + 2$

Example of Recurrence: Mergesort

```
mergeSort(A[ ], p, r)
▷ Sort A[p ... r]
{
        if (p < r) then {
                q ← (p+q)/2;    ----------  ①   ▷ the center of p & q
                mergeSort(A, p, q);  ----  ②   ▷ Sort the first
                mergeSort(A, q+1, r); ---  ③   ▷ Sort the next
                merge(A, p, q, r);   ------  ④   ▷ Merge
        }
}

merge(A[ ], p, q, r)
{
        sort the two arrays A[p ... q] and A[q+1 ... r]
        merge the two into one array A[p ... r]
}
```

```
mergeSort(A[ ], p, r)
▷ Sort A[p ... r]
{
        if (p < r) then {
                q ← (p+q)/2;    ----------  ①    ▷ the center of p & q
                mergeSort(A, p, q);  ----  ②    ▷ Sort the first
                mergeSort(A, q+1, r); ---  ③    ▷ Sort the next
                merge(A, p, q, r);    ------  ④    ▷ Merge
        }
}
```

- ✓ ②, ③ - Recursion
- ✓ ①, ④ - Overhead

Diverse Applications of Algorithms

- Car navigation
- Scheduling
  - TSP, Vehicle Routing, Job Task, …
- Human Genome Project
  - Matching, Phylogenetic tree, functional analyses, …
- Search and Retrieval
  - Database, Web pages, …
- Resource Allocation
- Semiconductor Design
  - Partitioning, placement, routing, …
- …

Why do we analyze algorithms?

- Assure the integrity
- Measure the efficiency in resource management
  - Resource
    - **Time**
    - Memory, Communication Bandwidth, …

Algorithm Analysis

- Small input?
  - The algorithm doesn't have to be efficient
- Big input?
  - Efficiency matters!
  - Inefficient algorithm can be fatal
- Asymptotic analysis deals the case with big inputs.

Asymptotic Analysis

- Analysis on a big input
- Example of Asymptotic Notation

$$\lim_{n \to \infty} f(n)$$

- O, Ω, Θ, ω, o notations

Asymptotic Notations

- $O(f(n))$
  - A function increases at most at the rate of $f(n)$
  - e.g., $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, …
- Formal definition
  - $O(\boldsymbol{f}(n)) = \{ \boldsymbol{g}(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t.} \forall n \geq n_0, c\boldsymbol{f}(n) \geq \boldsymbol{g}(n) \}$
  - Conventionally, we write $\boldsymbol{g}(n) \in O(\boldsymbol{f}(n))$ as $\boldsymbol{g}(n) = O(\boldsymbol{f}(n))$
- Intuitive meaning
  - $\boldsymbol{g}(n) = O(\boldsymbol{f}(n)) \Rightarrow \boldsymbol{g}$ does not increase faster than $\boldsymbol{f}$
  - Ignore the constant ratios

Asymptotic Notations

- Example, $O(n^2)$
  - $3n^2 + 2n$
  - $7n^2 - 100n$
  - $n\log n + 5n$
  - $3n$
- As tight as possible
  - When $n\log n + 5n = O(n\log n)$, we don't have to write as $O(n^2)$
  - If not tight, we lose information

Asymptotic Notations

- $\Omega(\ f(n)\ )$
  - A function increases at least at the rate of $f(n)$
  - Symmetric to $O(\ f(n)\ )$
- Formal definition
  - $\Omega(\boldsymbol{f}(n)) = \{\ \boldsymbol{g}(n) \mid \exists c > 0,\ n_0 \geq 0 \text{ s.t.} \forall n \geq n_0,\ c\boldsymbol{f}(n) \leq \boldsymbol{g}(n)\ \}$

- Intuitive meaning
  - $\boldsymbol{g}(n) = \Omega(\boldsymbol{f}(n)) \Rightarrow \boldsymbol{g}$ does not increase slower than $\boldsymbol{f}$

Asymptotic Notations

- $\Theta(\ f(n)\ )$
  - An increasing function at the rate of $f(n)$
- Formal definition
  - $\Theta(\ \boldsymbol{f}(n)\ ) = O(\ f(n)\ ) \cap \Omega(\ f(n)\ )$
- Intuitive meaning
  - $\boldsymbol{g}(n) = \Theta(\boldsymbol{f}(n)) \Rightarrow \boldsymbol{g}$ increases as much as $\boldsymbol{f}$

Intuitive Meanings

- $O(f(n))$
  - Tight *or* loose upper bound
- $\Omega(f(n))$
  - Tight *or* loose lower bound
- $\Theta(f(n))$
  - Tight bound

Examples of Asymptotic Complexity

- Time complexities of Sorting Algorithms
  - Selection Sort
    - $\Theta(n^2)$
  - Heap Sort
    - $O(n \log n)$
  - Quick Sort
    - $O(n^2)$
    - Average $\Theta(n \log n)$

Types of Complexity Analysis

- Worst-case
  - Analysis for the worst-case input(s)
- Average-case
  - Analysis for all inputs
  - More difficult to analyze
- Best-case
  - Analysis for the best-case input(s)
  - Not useful!

Complexity in Storing/Search

- Array
  - $O(n)$
- Binary search trees
  - Worst case $\Theta(n)$
  - Average $\Theta(\log n)$
- Balanced binary search trees
  - Worst case $\Theta(\log n)$
- B-trees
  - Worst case $\Theta(\log n)$
- Hash table
  - Average $\Theta(1)$

Find an element in an array of size n

- Sequential search
  - When the elements are randomly distributed:
  - Worst case: $\Theta(n)$
  - Average case: $\Theta(n)$
- Binary search
  - When the array is sorted:
  - Worst case: $\Theta(\log n)$
  - Average case: $\Theta(\log n)$