Basics of Algorithm Design and Analysis

전혀 새로운 아이디어를 갑자기
착상하는 일이 자주 있다.
하지만 그것을 착상하기까지 오랫동안
끊임없이 문제를 생각한다.
오랫동안 생각한 끝에 갑자기
답을 착상하게 되는 것이다.

– 라이너스 폴링

Good Algorithm

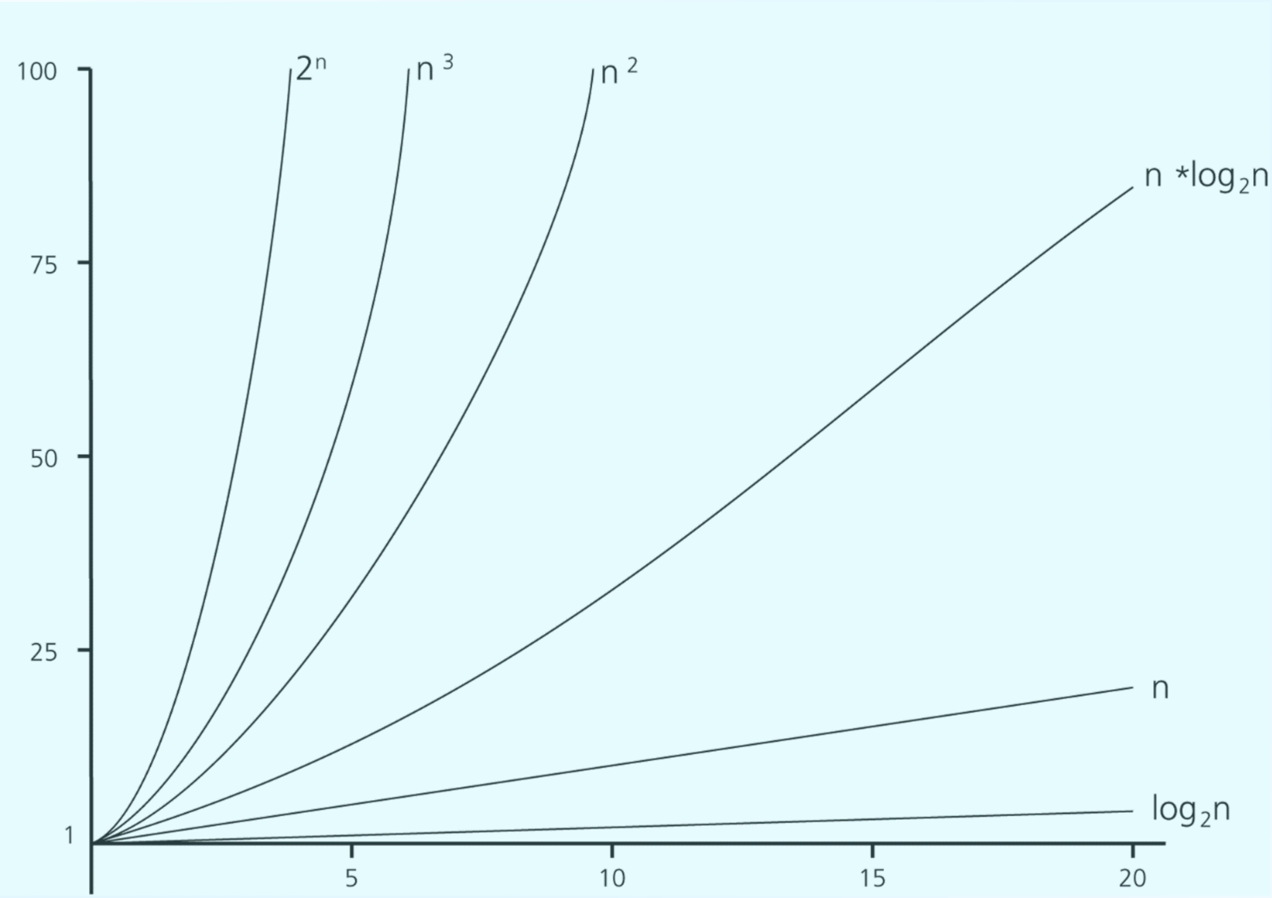
- Should be clear
  - Easy to understand, and simple, if possible
  - Overly symbolic representation is often hard to understand
  - If clear, natural language is also okay
- Should be efficient
  - An algorithm may take a billion times longer than another for a same problem

```
sample(A[], n):
    …
    sum ← 0
    for i ← 1 to n
        sum ← sum + A[i]
    avg ← sum / n
    …
```

```
sample(A[], n):
    …
    avg ← the average of A[1…n]
    …
```

Running Times

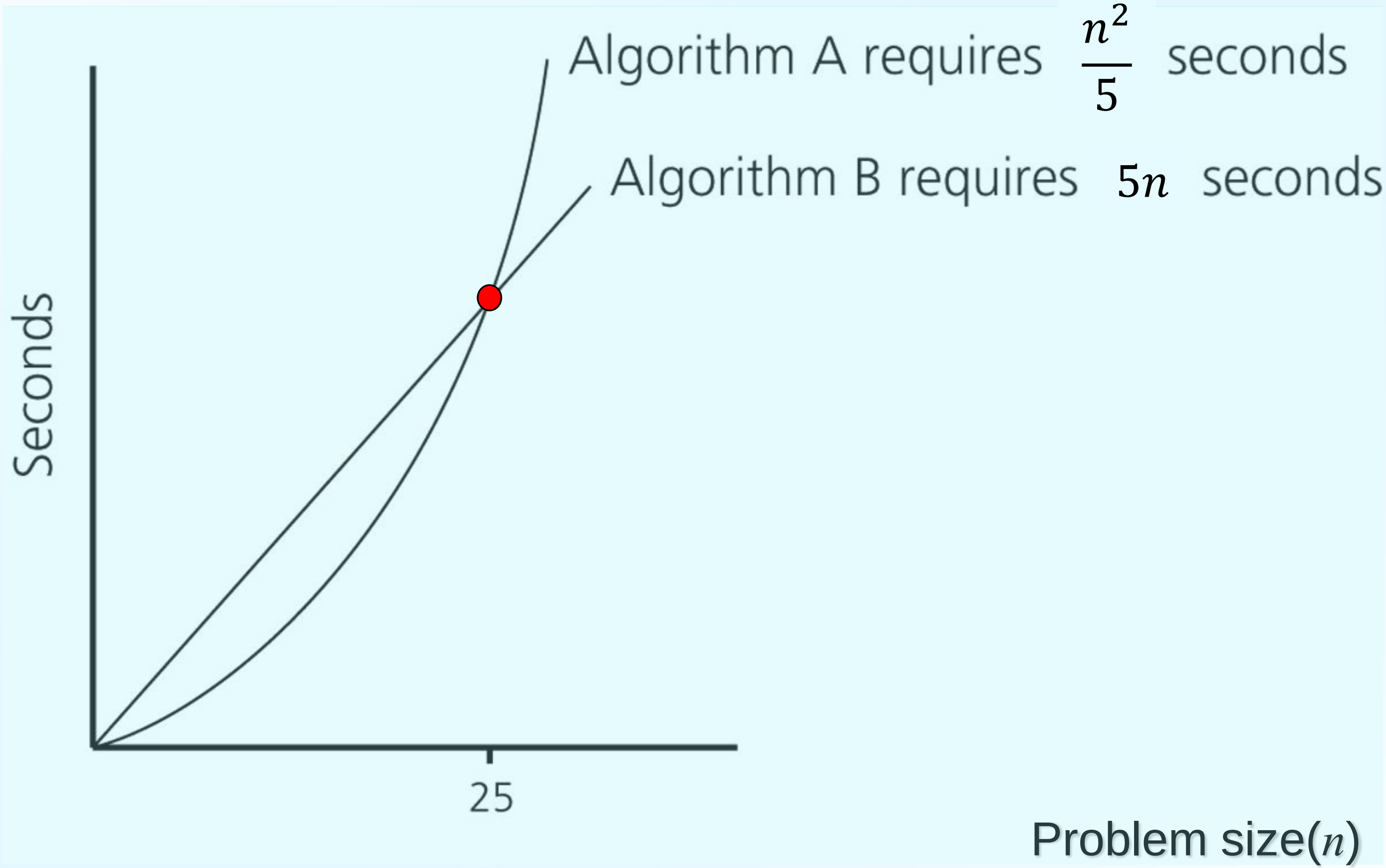
Running time



Problem size(n)

Running Times

Running time



크게 증가한다는 느낌 갖기

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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Criteria for Running Times

There are diverse criteria

examples:

- # of for/while loop iterations
- # of visiting a particular line
- # of calls for a particular function
- …

sample(n):
sum ← 0
for i ← 1 **to** n-1
for j ← i+1 **to** n
sum ← sum + A[i]*A[j]
return sum

factorial(n):
if n = 0 **or** n=1 **return** 1
else return n * factorial(n-1)

Examples of Running Time

```
sample1(A[], n):
        k = ⌊n/2⌋
        return A[k]
```

✓ constant time, independent of n

Examples of Running Time

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

✓ proportional to n

Examples of 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
```

✓ proportional to n^2

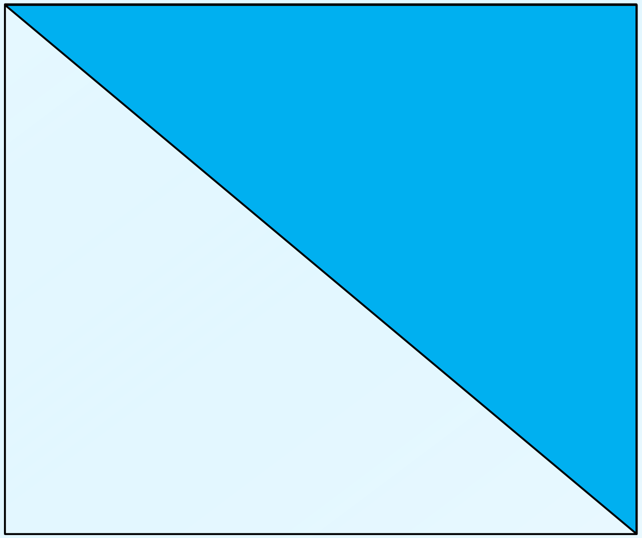
Examples of Running Time

```
sample4(A[], n):
    sum ← 0
    for i ← 1 to n
        for j ← 1 to n
            k ← choose at random ⌊n/2⌋ elements out of A[1 ... n]
                                        and take the maximum
            sum ← sum + k
    return sum
```

✓ proportional to n^3

Examples of Running Time

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



✓ proportional to n^2

Examples of Running Time

```
factorial(n):
    if (n=1) return 1
    return n*factorial(n-1)
```

✓ proportional to n

Examples of Running Time

```
sample7(A[], n):
    if (n > 1)
        sum ← 0
        for i ← 1 to n
            sum ← sum+ A[i]
    return (sum + sample7(A, n-1))
```

✓ proportional to n^2

Recursion and Inductive Thinking

Recursion

calling itself

Recursive structure

- A problem contains the same problem(s) of smaller size(s)
- e.g. 1: factorial
  - $N! = N \times (N\text{-}1)!$
- e.g. 2: recurrence in progression
  - $a_n = a_{n-1} + 2$ (arithmetic progression)

Example of Recursion: Mergesort

```
mergeSort(A[], p, r):          ▷ Sort A[p ... r]
    if (p < r)
        q ← ⌊(p + r)/2⌋  ------  ❶  ▷ center location of p and q
        mergeSort(A, p, q)  -----  ❷  ▷ sorting the former half
        mergeSort(A, q+1, r) ---  ❸  ▷ sorting the latter half
        merge(A, p, q, r)  -------  ❹  ▷ merge

merge(A[], p, q, r):
    Merge two sorted arrays A[p ... q] and A[q+1 ... r]
    to a sorted array A[p ... r]
```

```
mergeSort(A[], p, r):          ▷ Sort A[p ... r]
    if (p < r)
        q ← ⌊(p + q)/2⌋  ------  ❶  ▷ center location of p and q
        mergeSort(A, p, q)  -----  ❷  ▷ sorting the former half
        mergeSort(A, q+1, r) ---  ❸  ▷ sorting the latter half
        merge(A, p, q, r)  -------  ❹  ▷ merge
```

- ✓ ❷, ❸: recursive calls
- ✓ ❶, ❹: overhead for weaving recursive relation

Why Do We Analyze an Algorithm

- To guarantee integrity, correctness
- Efficiency of using resources
  - Resources
    - Time
    - Memory, network bandwidth, …

Algorithm Analysis

- Small problems
  - Efficiency is not a big issue
  - Non-efficient algorithms are also okay
- Large-enough problems
  - Efficiency is critical
  - Non-efficient algorithms might be fatal
- Asymptotic analysis
  - Analysis for large-enough problems

Asymptotic Analysis

- Analysis for large-enough problems
- You already encountered some examples

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

- O, Ω, Θ, ω, o-notations

Asymptotic Notations 점근적 표기법

O()

$O(g(n))$ – big Oh
- Set of functions growing at most at the ratio of $g(n)$
- e.g., $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, …

- Formal definition
  - $O(g(n)) = \{ f(n) \mid \exists\, c > 0, n_0 \geq 0 \text{ s.t. } \forall\, n \geq n_0, \boldsymbol{f}(n) \leq c\boldsymbol{g}(n) \}$
  - In practice, we use $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$
- Intuitive meaning
  - $\boldsymbol{f}(n) = O(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$ grows no faster than $\boldsymbol{g}$
  - Difference in a constant ratio is negligible
- Examples
  - $O(n^2) = \{3n^2 + 2n,\ 7n^2 - 100n,\ n\log n + 5n,\ 3n,\ \ldots\}$

Asymptotic Notations

$\Omega(g(n))$ – big Omega

- Set of functions growing at least at the ratio of $g(n)$
- Symmetric to $O(g(n))$

- Formal definition
  - $\Omega(g(n)) = \{ f(n) \mid \exists\, c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, \boldsymbol{f}(n) \geq c\boldsymbol{g}(n) \}$

- Intuitive meaning
  - $\boldsymbol{f}(n) = \Omega(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$ grows no slower than $\boldsymbol{g}$

- Examples
  - $\Omega(n^2) = \{3n^2 + 2n,\ 7n^2 - 100n,\ n^3 + n\log n + 5n,\ 2^n + 3n,\ \ldots\}$

Asymptotic Notations

$\Theta()$

$\Theta(g(n))$ – big Theta

- Set of functions growing at the same ratio of $g(n)$

- Formal definition
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

- Intuitive meaning
  - $\boldsymbol{f}(n) = \Theta(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$ grows at the same ratio to $\boldsymbol{g}$

- Examples
  - $\Theta(n^2) = \{7n^2 + 9n + 4,\ 15n^2 - 100n,\ 2n^2 - 1000n,\ \ldots\}$

Asymptotic Notations

o()

$o(g(n))$ – little oh

- Set of functions growing at a lower ratio than $g(n)$

- Formal definition
  - $o(\boldsymbol{g}(n)) = \{\, \boldsymbol{f}(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$

- Intuitive meaning
  - $\boldsymbol{f}(n) = o(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$ grows slower than $\boldsymbol{g}$

- Examples
  - $o(n^2) = \{9n + 4,\ 100n\log n + 25n,\ 2n - 1000,\ 5n^{1.99} + 17n + 4,\ \ldots\}$

Asymptotic Notations

ω()

$\omega(g(n))$ – little omega

- Set of functions growing at a greater ratio than $g(n)$

- Formal definition
  - $\omega(g(n)) = \{ f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \}$

- Intuitive meaning
  - $\boldsymbol{f}(n) = \omega(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$ grows faster than $\boldsymbol{g}$

- Examples
  - $\omega(n^2) = \{3n^2\log 3n + 2n,\ 7n^3 - 100n,\ n^4 + n\log n + 5n,$
    $2^n + 3n,\ 0.5n^{2.01} - 59n - 45,\ \ldots\}$

Asymptotic Notations

An example

sample3(A[], n):
    sum ← 0
    for i ← 1 **to** n
        for j ← 1 **to** n
            sum← sum+ A[i]*A[j]
    return sum

$O(n^2)$: O	$O(n)$: X	$O(n^3)$: O
$\Omega(n^2)$: O	$\Omega(n)$: O	$\Omega(n^3)$: X
$\boxed{\Theta(n^2)\text{: O}}$	$\Theta(n)$: X	$\Theta(n^3)$: X

The most informative

Asymptotic Notations

(10 points) What is the asymptotic time in O-notation of this algorithm?

sample3(A[], n):
    sum ← 0
    for i ← 1 **to** n
        for j ← 1 **to** n
            sum← sum+ A[i]*A[j]
    return sum

Be as tight as possible.
Leave as little information loss as possible.

Students' answers

$O(n^2)$: right, 10 points

$O(n)$: wrong, 0 points

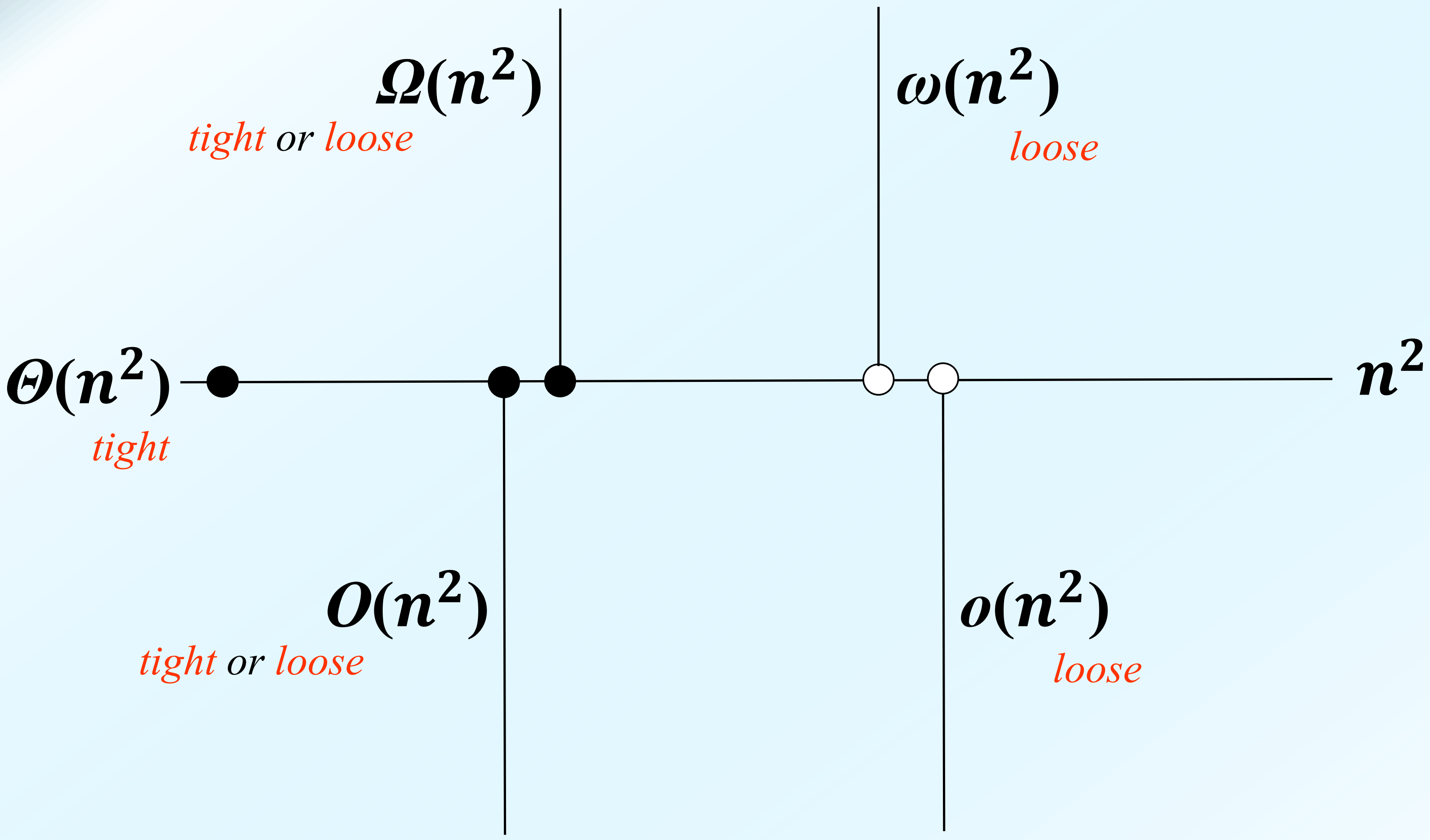
$O(n^{100})$: right, but 0 points

$O(n^3)$: right, but 0.1 points

Intuitive Meanings

- $O(g(n))$
  - Tight *or* loose upper bound
- $\Omega(g(n))$
  - Tight *or* loose lower bound
- $\Theta(g(n))$
  - Tight bound
- $o(g(\mathrm{n}))$
  - Loose upper bound
- $\omega(g(n))$
  - Loose lower bound

Yet Another Intuitive View



Examples of Asymptotic Complexity

Sorting algorithms

- Selection sort: $\Theta(n^2)$
- Heapsort: $O(n\log n)$
- Quicksort:
  - $O(n^2)$
  - Average $\Theta(n\log n)$
  - Worst-case $\Theta(n^2)$

Types of Complexity Analyses

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)
- Usually no useful

Asymptotic Complexities of Indexes

- Array
  - $O(n)$
  - At least one of insertion, deletion, and search is $\Theta(n)$
- 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-case $\Theta(1)$

Finding an Element in an array of length n

Sequential search

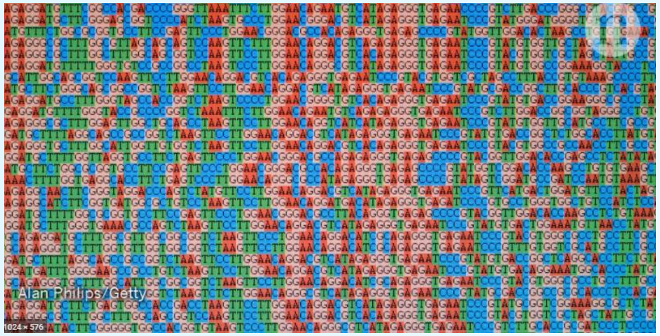
- Elements are stored at random
- Worst case: $\Theta(n)$
- Average case: $\Theta(n)$
- Best case: $\Theta(1)$

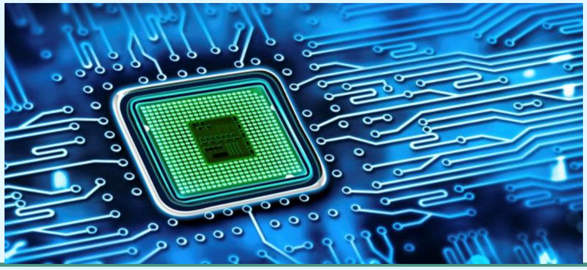
Binary search

- Elements are stored in sorted order
- Worst case: $\Theta(\log n)$
- Average case: $\Theta(\log n)$
- Best case: $\Theta(1)$

Diverse Applications of Algorithms

- Car navigation
- Scheduling
  - TSP, vehicle routing, operations process, …
- Human genome project
  - Matching, phylogenetic tree, functional analyses, …
- Search
  - Databases, web pages, texts, …
- Placement of resources
  - Wharf, logistic warehouse, …
- Semiconductor design
  - Partitioning, placement, routing, …
- …

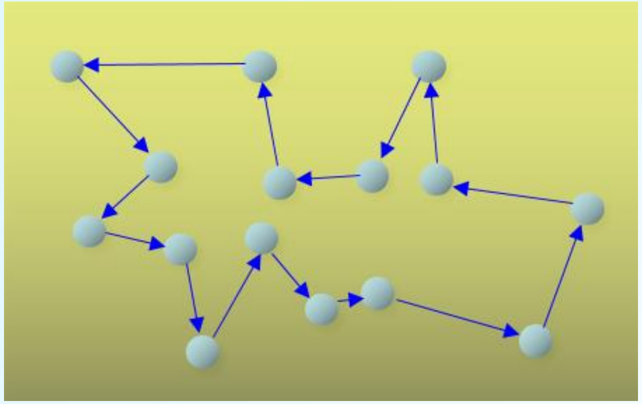




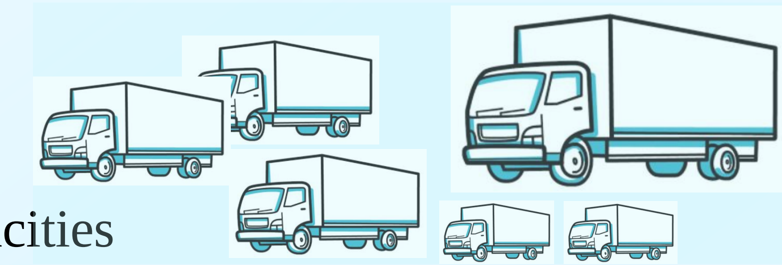
An Example

Traveling Salesman Problem (TSP)

- Given locations of N customers
- Objective
  - circulate all the customers
    while minimizing the traveling distance



An Example



Vehicle Routing Problem (VRP)

- Multiple vehicles with their own capacities
- Multiple customer destinations(or plus due times)
- Objective
  - serve all the customers while minimizing
    the # of vehicles and the traveling distance