# Introduction to Algorithms

## L2. Algorithmic analysis. II

Instructor : Kilho Lee

# Outline

- Techniques to analyze correctness and runtime
  - ~~Proving correctness with induction~~ **Done!**
  - Proving runtime with asymptotic analysis **Today!**
  - *Problems: Comparison-sorting*
  - *Algorithms: Insertion sort*
  - Reading: CLRS 2.1, 2.2, 3

How do we measure the runtime of an algorithm?

# Runtime Analysis

- We might care about the runtime of an algorithm in a few cases.
  - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
    - We'll focus on this type of analysis since it tells us that an algorithm performs at least this fast for *every* input.
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
  - **Average-case analysis** What is the runtime of the algorithm on the average input?

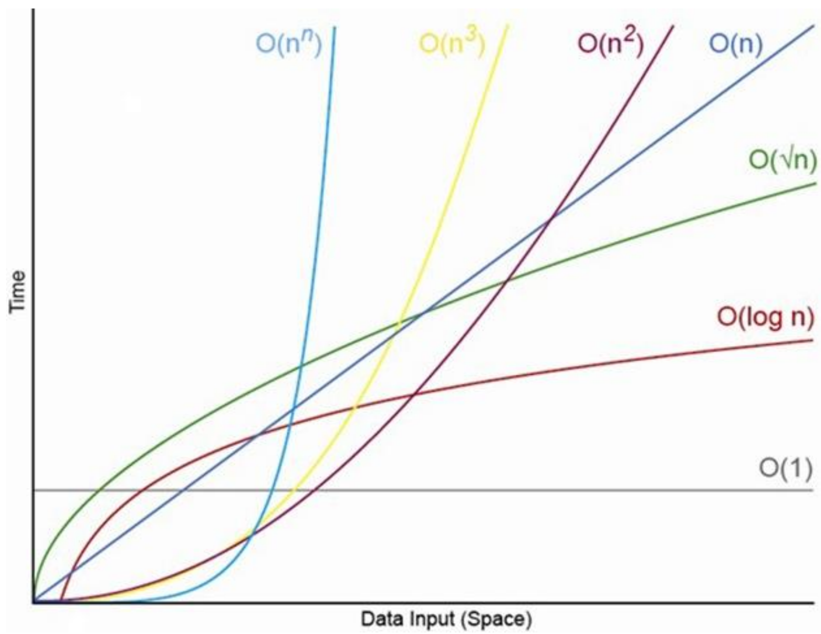
# Asymptotic Analysis

- What does it mean to measure “runtime” of an algorithm?
  - Engineers probably care most about the “real-world time”: how long does the algorithm take in seconds, minutes, hours, days, etc.?
  - This heavily depends on computer hardware, programming language, etc.
  - While important, it will not be the emphasis of this course.
  - Instead, we want to use a universal measure of runtime that’s independent of these considerations.
  - Time-complexity.

# Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).

One algorithm is “faster” than another if its runtime scales better with the size of the input.
(입력사이즈 n값이 매우 커질 때에도 알고리즘이 빠르게 작동하도록)



# Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).

One algorithm is “faster” than another if its runtime scales better with the size of the input.

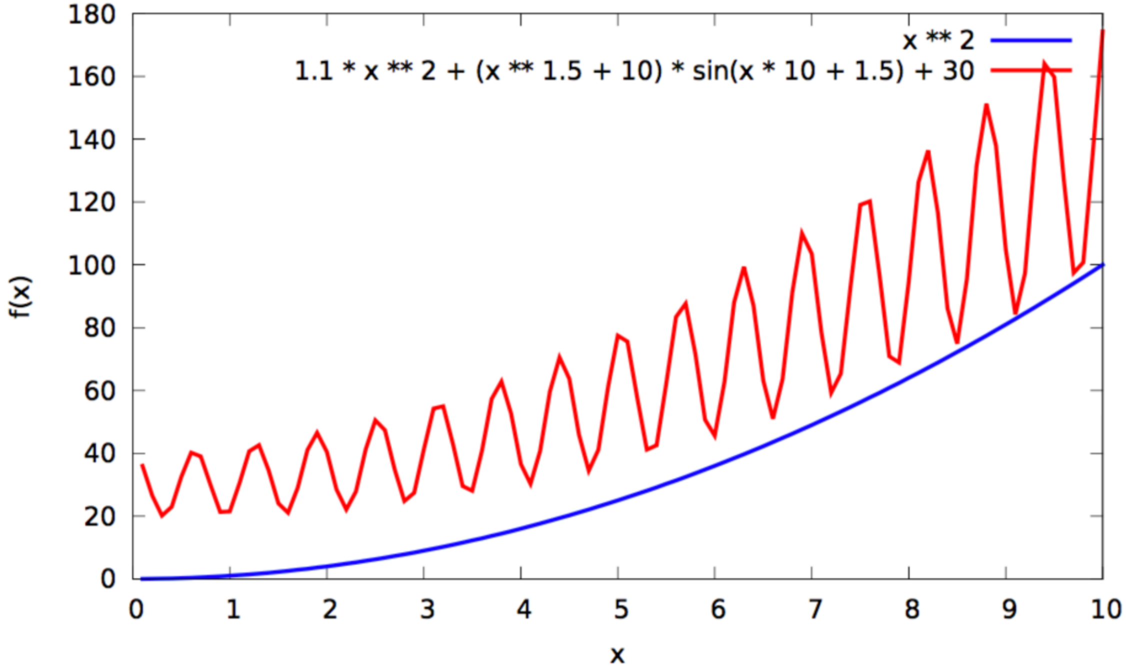
- **Pros**
  - Abstracts away from hardware-/language- specific issues (추상화)
  - Make algorithm analysis much more tractable (다루기 쉬움)
- **Cons**
  - Only makes sense if n is large (compared to the constant factors).

9,999,999,999,999 n
is “better” than $n^2$ ?!?!

# Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).
  - Are the following functions similar or not?

$$f_1(x) = x^2 \qquad f_2(x) = 1.1x^2 + (x^{1.9} + 10)\sin(10x + 1.5) + 30$$

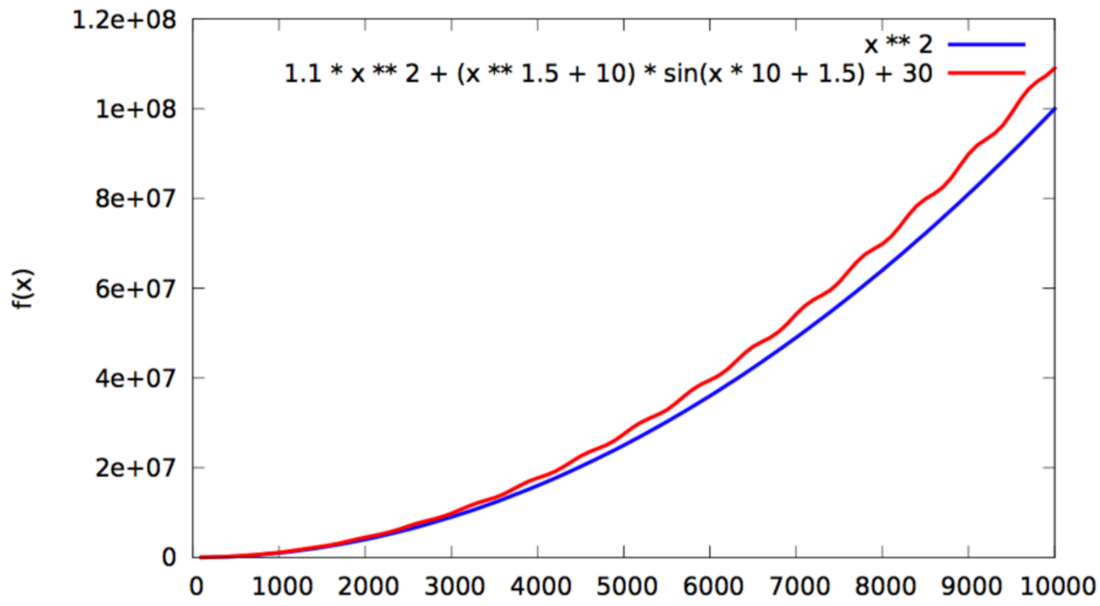


# Asymptotic Analysis

- **Key insight** Focus on how the runtime scales with n (the input size).
  - Are the following functions similar or not?

Ignore lower-order terms

$$f_1(x) = x^2 \qquad f_2(x) = 1.1x^2 + (x^{1.9} + 10)\sin(10x + 1.5) + 30$$




Informally, it can be determined by ignoring constants and non-dominant growth terms.

Running times for $f_1(x)$ and $f_2(x)$ are both $n^2$

# Big-O (O(…)) Means Upper-Bound

- Big-O (“빅-오” 로 읽음) notation (빅-오 표기법) is a mathematical notation for upper-bounding a function’s rate of growth.

# Big-O Notation

- Let T(n), g(n) be functions of positive integers.
  - You can think of T(n) as being a runtime: positive and increasing as a function of n.
- We say **"T(n) is O(g(n))"** if g(n) grows at least as fast as T(n) as n gets large. ( n이 증가할 때, g(n)이 최소한 T(n) 만큼은 빠르게 증가하는 경우, 곧 T(n)이 g(n) 보다는 느리게 증가한다.)
- O(g(n)) is a set. T(n) belongs to the set
- Formally,

$$T(n) = O(g(n))$$

$$\textbf{iff}$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$

# Big-O Notation

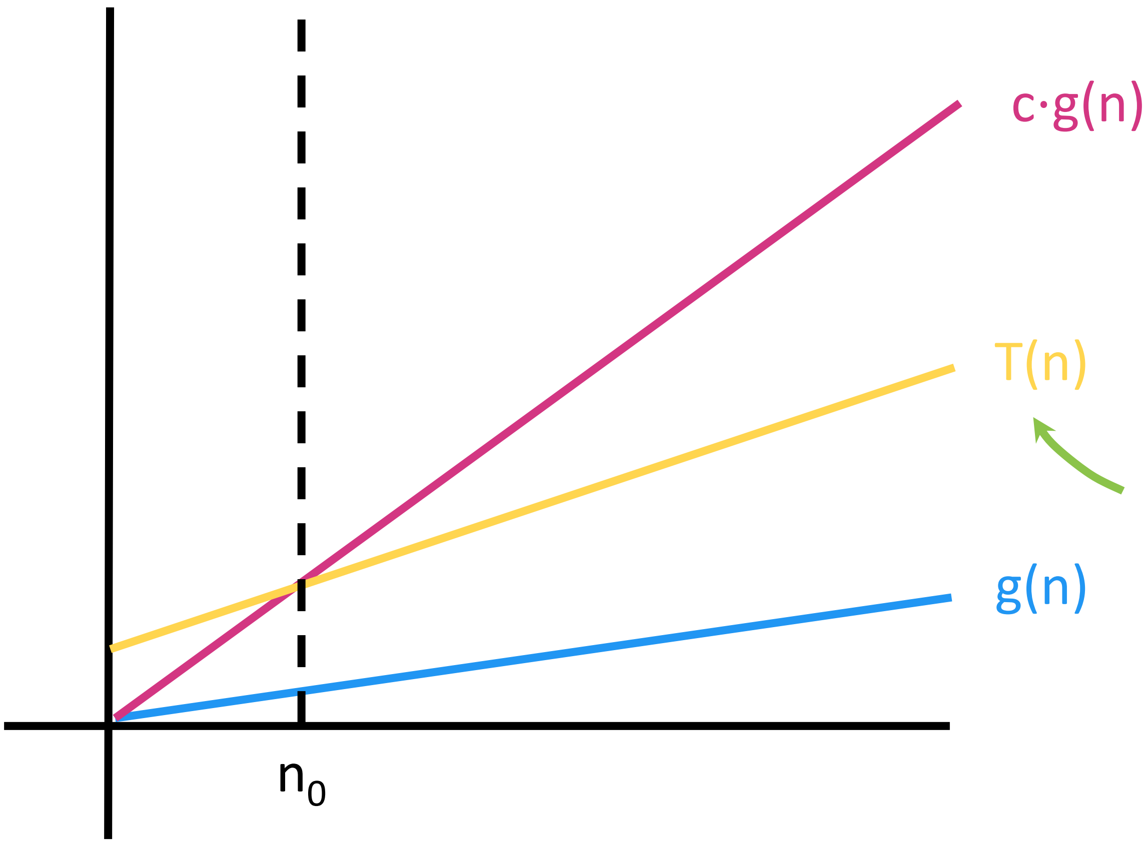
$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- Graphically,




Big-O defines **"T(n) = O(g(n))"** to mean there exists some c and $n_0$ such that the pink line given by c·g(n) is **above** the yellow line for all values to the right of $n_0$.

Big-O defines **"T(n) = O(g(n))"** $n_0$ 의 우측에서는 pink line 이 yellow line 보다 항상 위쪽에 그려지도록 만드는 $n_0$ 와 c가 존재한다.

# Big-O Notation

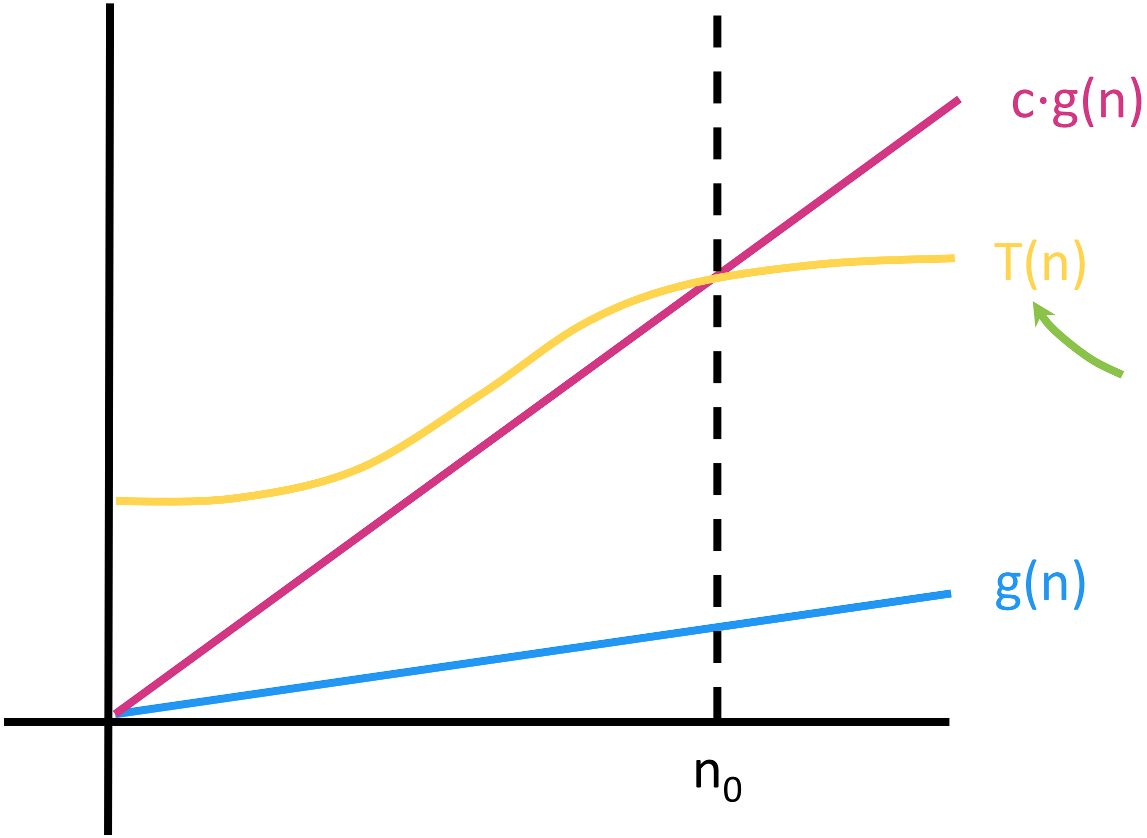
$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- Graphically,




T(n) doesn't always have to be linear.

# Big-O Notation

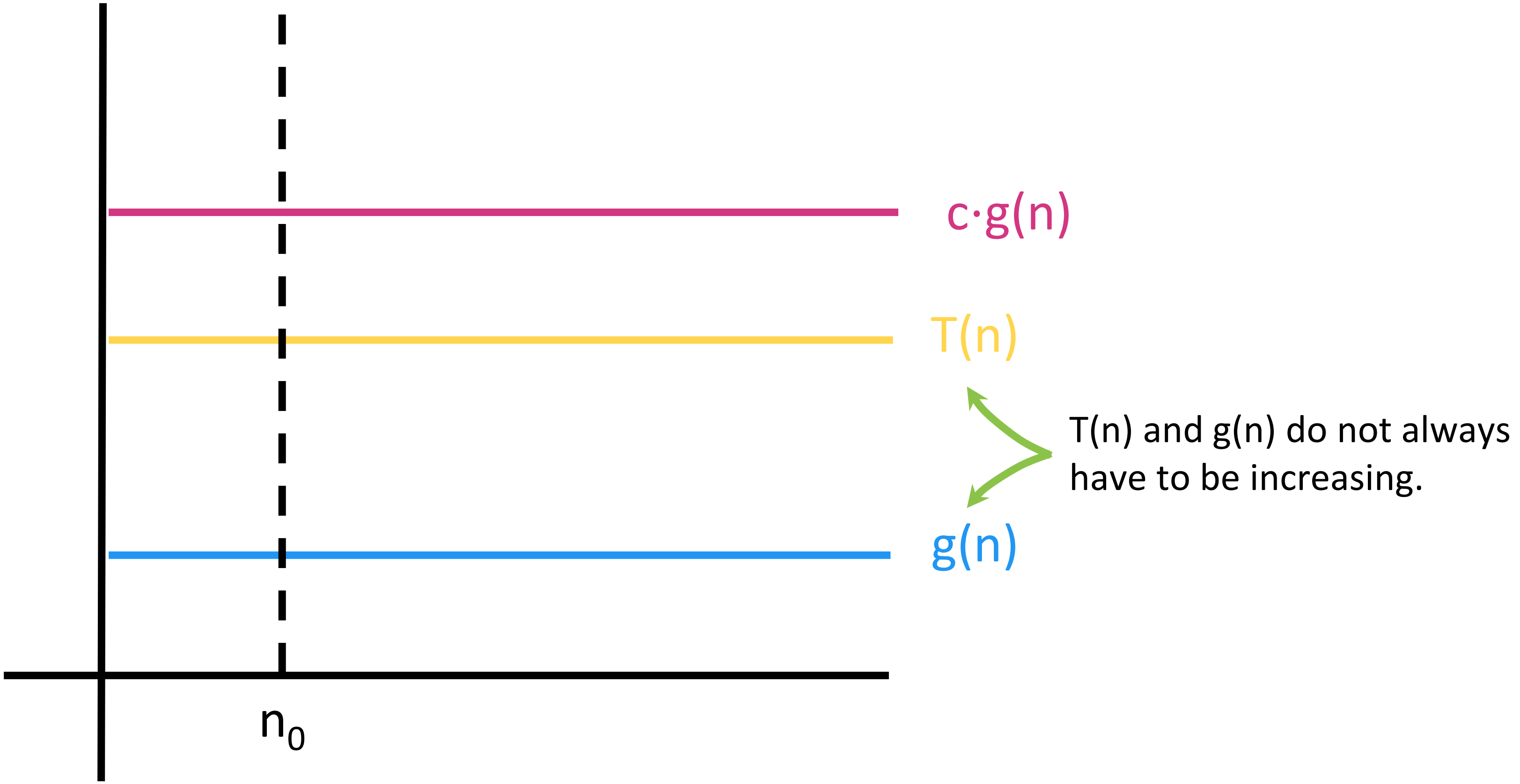
$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- Graphically,



# Big-O Notation

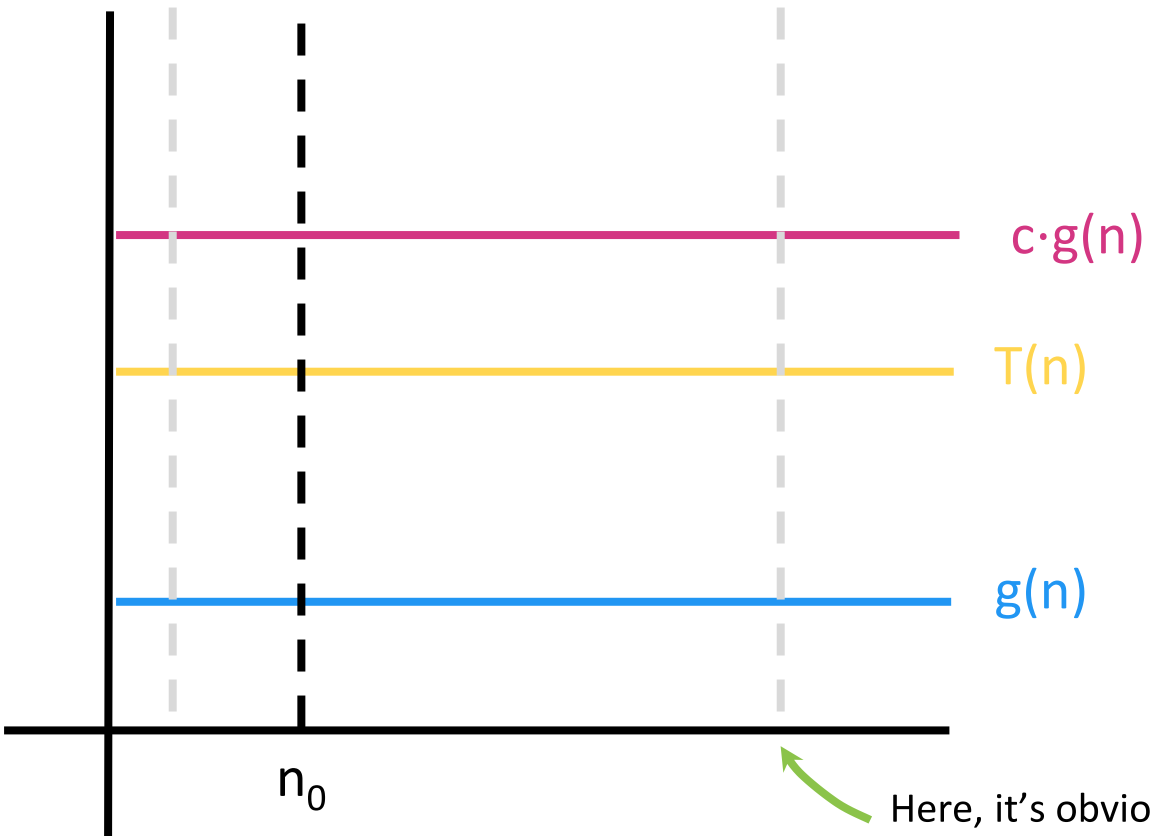
$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- Graphically,




Here, it's obvious that there isn't one "correct" choice for $n_0$.

# Big-O Notation

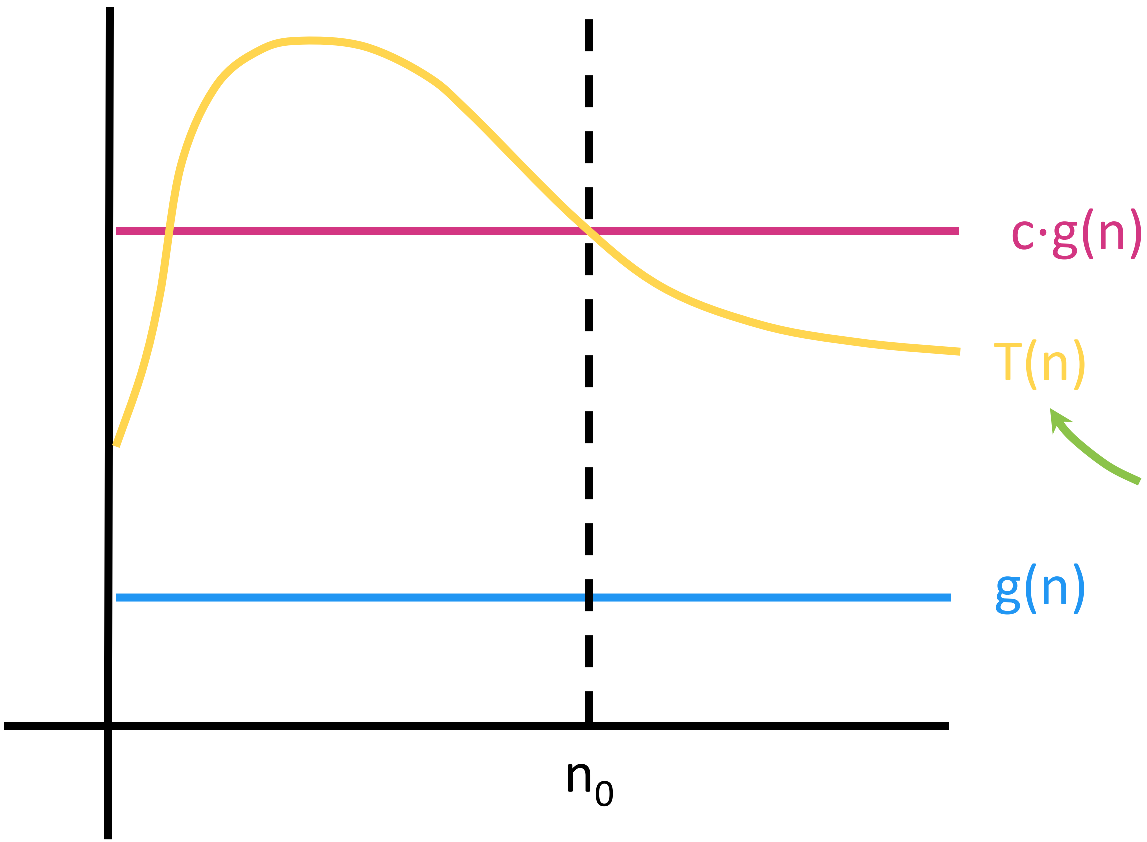
$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- Graphically,




T(n) looks weird, but it's still O(g(n)).

# Big-O Notation

$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- To prove $T(n) = O(g(n))$, show that there exists a $c$ and $n_0$ that satisfies the definition.

# Big-O Notation

**$T(n) = O(g(n))$**

**iff**

**$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,**

**$0 \leq T(n) \leq c \cdot g(n)$**

- To prove $T(n) = O(g(n))$, show that there exists a c and $n_0$ that satisfies the definition.
  - **For example,**

    Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

    Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since n is positive and $1 \leq \log n$ for $n \geq 2$.

# Big-O Notation

$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- To prove $T(n) = O(g(n))$, show that there exists a $c$ and $n_0$ that satisfies the definition.
  - **For example,**

    Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

    Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since $n$ is positive and $1 \leq \log n$ for $n \geq 2$.
- To prove $T(n) \neq O(g(n))$, proceed by contradiction.

# Big-O Notation

$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- To prove $T(n) = O(g(n))$, show that there exists a $c$ and $n_0$ that satisfies the definition.
  - **For example,**

    Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

    Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since $n$ is positive and $1 \leq \log n$ for $n \geq 2$.
- To prove $T(n) \neq O(g(n))$, proceed by contradiction (귀류법).
  - **For example,**

    Suppose $T(n) = n^2$ and $g(n) = n$. We prove that $T(n) \neq O(g(n))$.

    Suppose there exists some $c$ and $n_0$ such that for all $n \geq n_0$, $n^2 \leq c \cdot n$. Consider $n = \max\{c, n_0\} + 1$. By construction, we have both $n \geq n_0$ and $n > c$, which implies that $n^2 > c \cdot n$.

# Big-O Notation

$T(n) = O(g(n))$

iff

$\exists c, n_0 > 0$ s.t. $\forall n \geq n_0$,

$0 \leq T(n) \leq c \cdot g(n)$

- To prove $T(n) = O(g(n))$, show that there exists a c and $n_0$ that satisfies the definition.
    - **For example,**

      Suppose $T(n) = n$ and $g(n) = n \log(n)$. We prove that $T(n) = O(g(n))$.

      Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq c \cdot n \log(n)$ for $n \geq n_0$ since n is positive and $1 \leq \log n$ for $n \geq 2$.
- To prove $T(n) \neq O(g(n))$, proceed by contradiction.
    - **For example,**

      Suppose $T(n) = n^2$ and $g(n) = n$. We prove that $T(n) \neq O(g(n))$.

      Suppose there exists some c and $n_0$ such that for all $n \geq n_0$, $n^2 \leq c \cdot n$. Consider $n = \max\{c, n_0\} + 1$. By construction, we have both $n \geq n_0$ and $n > c$, which implies that $n^2 > c \cdot n$.

Here's the contradiction: assuming $n^2 \leq c \cdot n$ implies $n^2 > c \cdot n$ (the opposite)

# Big-Ω Means Lower-Bound

- Big-Ω (“빅-오메가”) notation is a mathematical notation for **lower**-bounding a function’s rate of growth.
  - Informally, it can be determined by ignoring constants and non-dominant growth terms.

# Big-Ω Notation

- Let T(n), g(n) be functions of positive integers.
  - You can think of T(n) as being a runtime: positive and increasing as a function of n.
- We say **"T(n) is Ω(g(n))"** if g(n) grows at most as fast as T(n) as n gets large. ( n이 증가할 때, g(n)이 아무리 많이 증가해도 T(n) 보다는 빠르게 증가하지 않는 경우. 곧, T(n) 이 g(n)보다는 빠르게 증가한다.)
- Formally,

$$T(n) = \Omega(g(n))$$

$$\text{iff}$$

$$\exists c, n_0 > 0 \text{ s.t. } \forall n \geq n_0,$$

$$0 \leq c \cdot g(n) \leq T(n)$$

Switched these!

# Big-Ω Notation

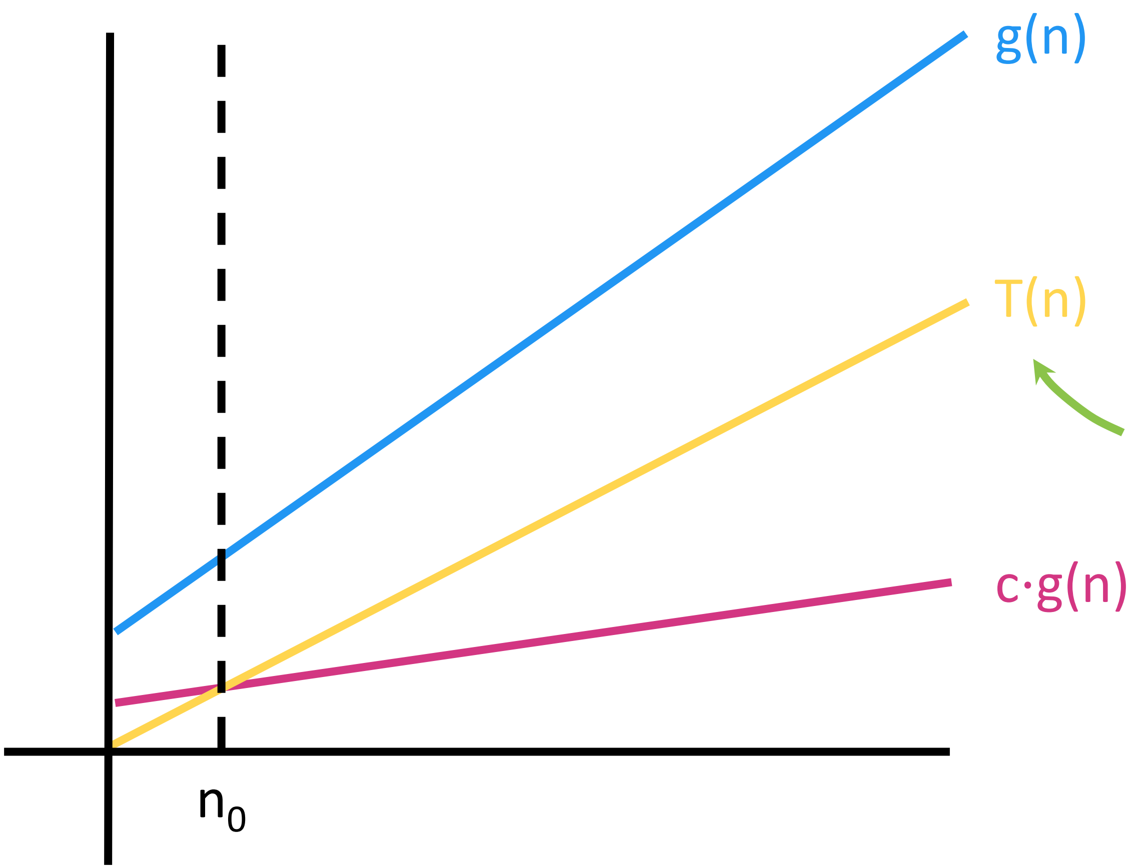
$T(n) = \Omega(g(n))$

**iff**

$\exists c, n_0 > 0$ **s.t.** $\forall n \geq n_0,$

$0 \leq c \cdot g(n) \leq T(n)$

- Graphically,




Big-**Ω** defines **"T(n) = Ω(g(n))"** to mean there exists some c and $n_0$ such that the pink line given by $c \cdot g(n)$ is **below** the yellow line for all values to the right of $n_0$.

Big-**Ω** defines **"T(n) = O(g(n))"** $n_0$ 의 우측에서는 pink line 이 yellow line 보다 항상 아래쪽에 그려지도록 만드는 $n_0$ 와 c가 존재한다.

# Big-Θ Means Upper and Lower-Bound

- We say **"T(n) is Θ(g(n))"** iff

**$T(n) = O(g(n))$**

**AND**

**$T(n)$ is $\Omega(g(n))$**

# More examples

- Claim I:
  - Prove that $\frac{3}{2}n^2 + \frac{5}{2}n - 3 = O(n^2)$
    - Equivalent: find $c$ and $n_0$ such that $\frac{3}{2}n^2 + \frac{5}{2}n - 3 \le cn^2$
    - Example proof

# More examples

- Claim II:
  - Prove that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$
    - Equivalent: find $c_1, c_2$ and $n_0$ such that $\mathrm{c_1 n^2} \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$
    - Example proof

# More examples

- Claim III:
  - Let $f(n) = 5n + 12$. Which of the following statements are true?
    1. $f(n) = O(n)$
    2. $f(n) = O(nlogn)$
    3. $f(n) = O(n^2)$
    4. $f(2^n)$

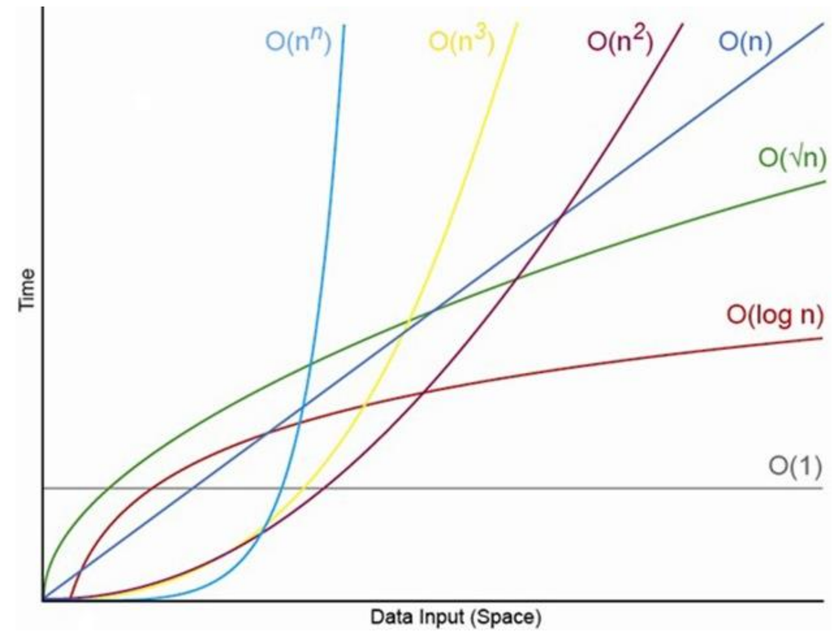
# Asymptotic Analysis

- Typically we call this asymptotic runtime as “time-complexity”.
- Again,
  - Informally, it can be determined by ignoring constants and non-dominant growth terms.
  - f1 and f2 has same time-complexity.

$$f_1(x) = x^2 \qquad f_2(x) = 1.1x^2 + (x^{1.9} + 10)\sin(10x + 1.5) + 30$$

# Asymptotic Analysis

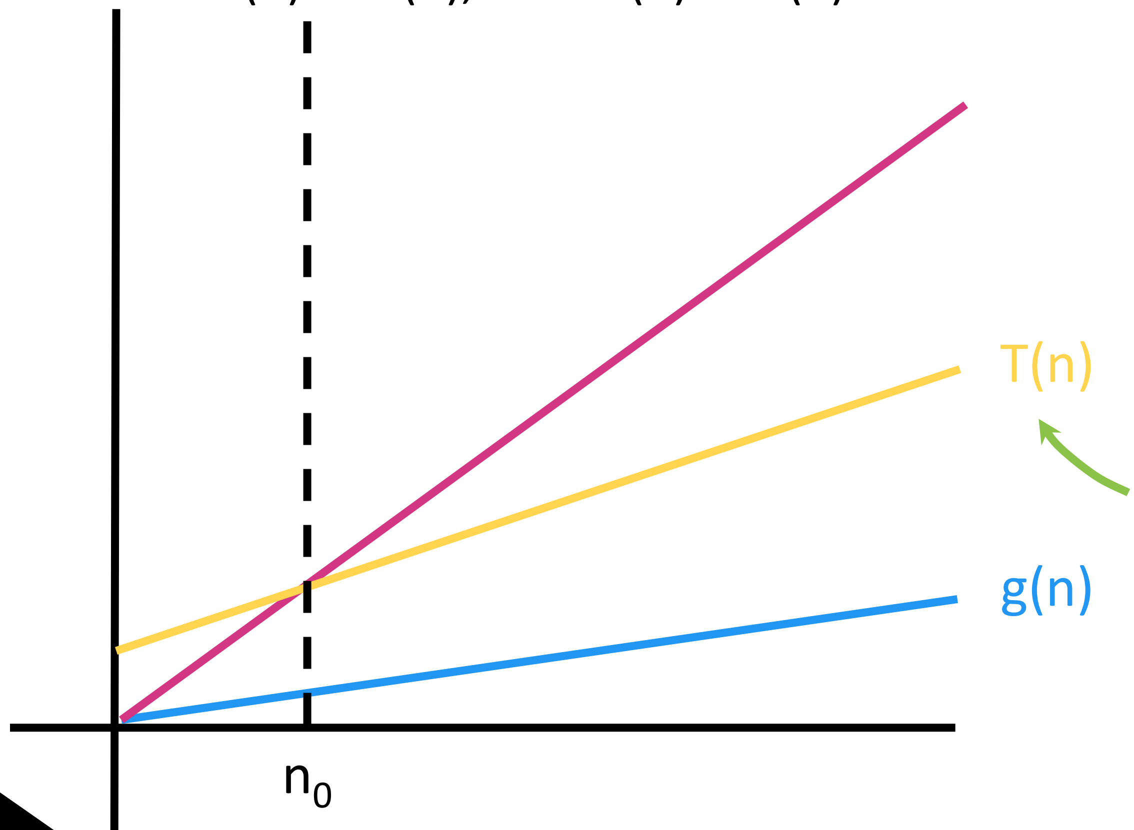
- Time scales (increasing order)
  - O (1) (constant, 상수)
  - O (log n) (logarithmic, 로그)
  - O ($\sqrt{n}$)
  - O (n) (linear, 선형)
  - O ($n^k$) (polynomial, 다항식)
  - O ($a^n$) (exponential, 지수)
  - O (n!) (factorial, 계승)




- Quiz. Which one is higher?
  - O(n) vs O(n*log n)

# Asymptotic Analysis

- Big-O is an upper-bound
  - If f(n) = O(n), then f(n) = O($n^2$)
- Big-Omega is a lower-bound
  - If f (n) = Ω (n), then f (n) = Ω (1)



# Runtime Analysis

- We might care about the runtime of an algorithm in a few cases.
  - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
  - **Average-case analysis** What is the runtime of the algorithm on the average input?

# Worst-Case Analysis

- What is the worst possible input for insertion sort?
  - Notice it’s possible for the inner `while` loop to iterate anywhere between 1 and i times. What if it iterated i times every single time? What input causes this pattern?

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

# Worst-Case Analysis

- What is the worst possible input for insertion sort?
  - Notice it's possible for the inner `while` loop to iterate anywhere between 1 and i times. What if it iterated i times every single time? What input causes this pattern?

| | | | | | | |
|---|---|---|---|---|---|---|
| n | … | 5 | 4 | 3 | 2 | 1 |

Reverse order

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

# Runtime Analysis

- Counting operations of the sorting algorithm

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

(n) Times

(n-1) times

(n-1) times

$\sum_{i=1}^{n-1} t_i$ times

$\sum_{i=1}^{n-1} (t_i - 1)$ times

$\sum_{i=1}^{n-1} (t_i - 1)$ times

(n-1) times

- Worst case: $\frac{3}{2}n^2 + \frac{7}{2}n - 4$ = O($n^2$)

# Runtime Analysis

- We might care about the runtime of an algorithm in a few cases.
  - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
  - **Average-case analysis** What is the runtime of the algorithm on the average input?

# Best-Case Analysis

- What is the best possible input for insertion sort?
  - Notice it’s possible for the inner `while` loop to iterate anywhere between 1 and i times. What if it iterated 1 time every single time? What input causes this pattern?

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

# Best-Case Analysis

- What is the best possible input for insertion sort?
  - Notice it's possible for the inner `while` loop to iterate anywhere between 1 and i times. What if it iterated 1 time every single time? What input causes this pattern?

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | ... | n |

Sorted order

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

# Runtime Analysis

- Counting operations of the sorting algorithm

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

(n) Times
(n-1) times
(n-1) times
$\sum_{i=1}^{n-1} t_i$ times
(n-1) times

- Best case: $5n - 4$ = O(n)

# Worst-Case vs. Best-Case Analysis

- We might care about the runtime of an algorithm in a few cases.
  - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
    - The worst-case runtime of insertion sort is **$O(n^2)$**.
  - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
    - The best-case runtime of insertion sort is **$\Theta(n)$**.
  - **Average-case analysis** What is the runtime of the algorithm on the average input?

# Runtime Analysis

- We might care about the runtime of an algorithm in a few cases.
    - **Worst-case analysis** What is the runtime of the algorithm on the worst possible input?
        - The worst-case runtime of insertion sort is **$O(n^2)$**.
    - **Best-case analysis** What is the runtime of the algorithm on the best possible input?
        - The best-case runtime of insertion sort is **$\Theta(n)$**.
    - **Average-case analysis** What is the runtime of the algorithm on the average input?
        - The average-case runtime of insertion sort is **$O(n^2)$**.

# Analyzing Runtime

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

**Upper-bound for <u>worst-case</u> runtime $O(n^2)$**

# Analyzing Runtime

```
void sort (int l[], int N)
{
    for (int i=1; i<N; i++)
    {
        int key = l[i];
        int j = i-1;
        while (j >= 0 && l[j] > key)
        {
            l[j+1] = l[j];
            j--;
        }
        l[j+1] = key;
    }
```

**Lower-bound for <u>best-case</u> runtime $\Omega(n)$**

# Outline

- Techniques to analyze correctness and runtime
    - Proving **correctness with induction**
        - Analyzing correctness of iterative algorithms
            - ✓ Loop invariant, proof by induction
    - Proving **runtime with asymptotic analysis**
        - Insertion sort
            - ✓ Worst-case analysis $O(n^2)$
            - ✓ Best-case analysis $\Omega(n)$
    - *Problems: Comparison-sorting*
    - *Algorithms: Insertion sort*
    - Reading: CLRS 2.1, 2.2, 3