Lecture #2
Algorithm Analysis (2)

Algorithm
JBNU
Jinhong Jung

In This Lecture

❑ Asymptotic Notations

- Understand Big-Omega and Big-Theta notations

❑ Simplifying Rules

- Quickly figure out asymptotic notations

❑ Other Discussions

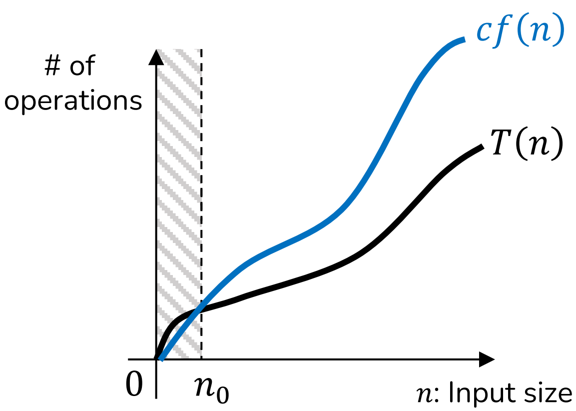
- Analysis with control statements
- Analysis with multiple parameters
- Complexity category

Outline

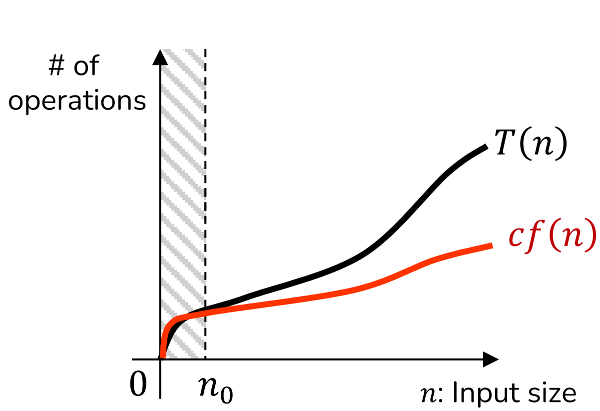
- Big-Omega Notation
- Big-Theta Notation
- Simplifying Rules
- Other Discussions

Asymptotic Notations

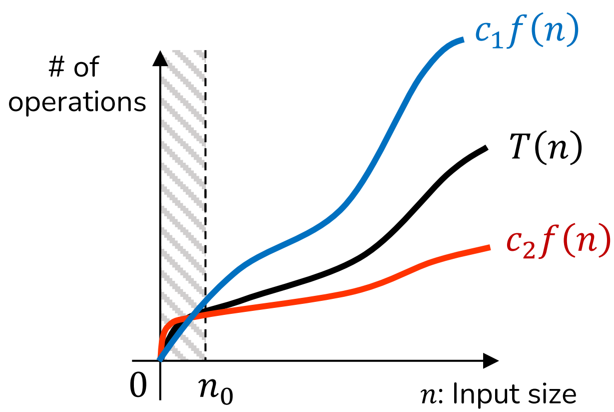
- Simple way to represent the limiting behaviors of an arbitrary complexity function
  - Big-O notation
  - Big-Omega notation
  - Big-Theta notation



Big-O



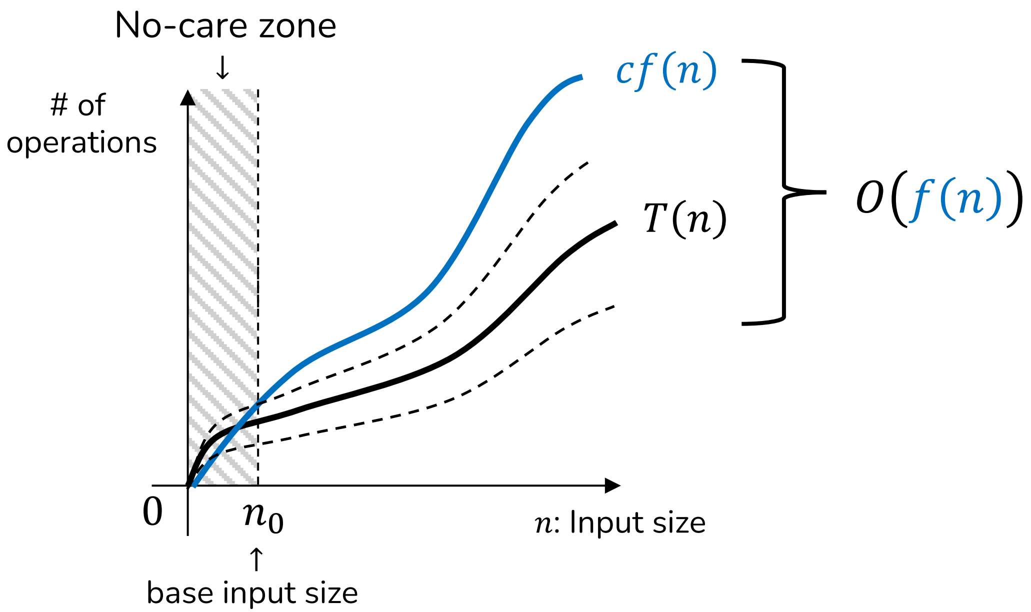
Big-Omega



Big-Theta

Big-O Notation (Recall)

- $T(n) = O\big(f(n)\big)$ **for [best | average | worst] case.**
  - $O\big(f(n)\big)$ = Set of functions $\leq\ cf(n)$ for all $n \geq n_0$
    - When the input size is large enough, it always executes in less than or equal to $cf(n)$ steps for the case.
    - $T(n)$ grows asymptotically no faster than $f(n)$ as upper bound.

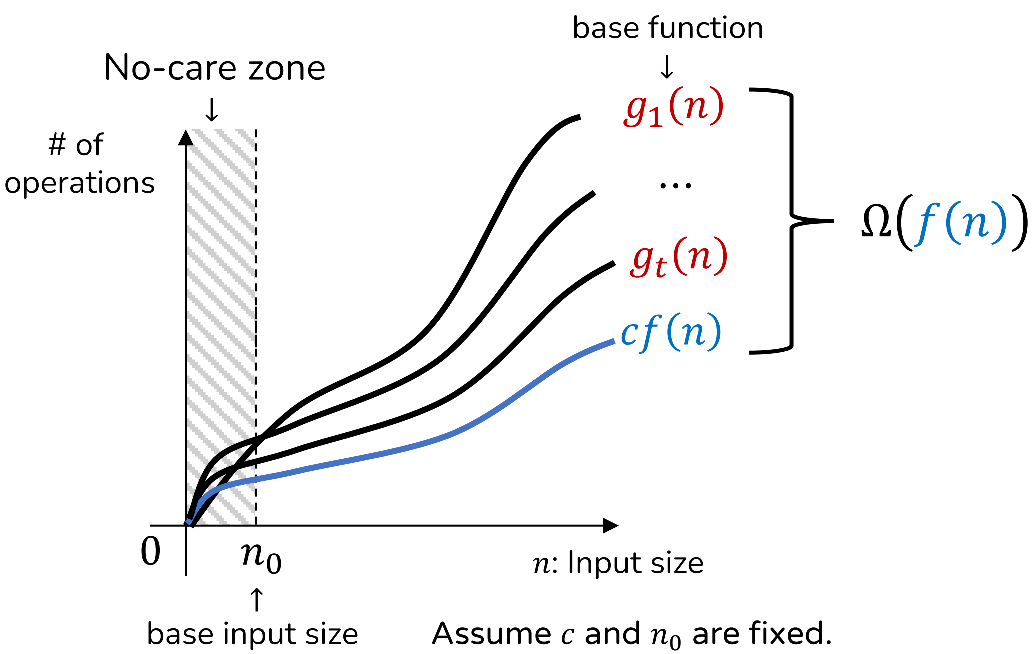


Big-Omega Notation (1)

❑ Definition of $\Omega(f(n))$

$$\Omega(f(n)) = \{\, g(n) \mid \text{there exist two positive constants } c \text{ and } n_0 \text{ such that } g(n) \geq cf(n) \text{ for all } n \geq n_0 \,\}$$

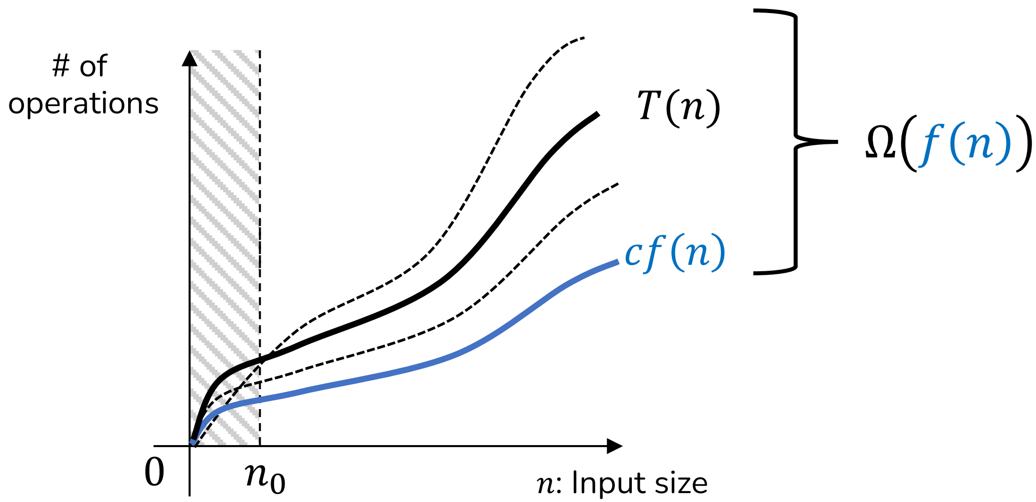
- Set of functions $\geq\ cf(n)$ for large input size n



Big-Omega Notation (2)

❑ Interpretation of $T(n) = \Omega(f(n))$

- The time complexity $T(n)$ of the algorithm is in $\Omega(f(n))$ for [best | average | worst] case.
  - When the input size is large enough, it always requires more than or equal to $cf(n)$ steps for the case.
  - $T(n)$ grows asymptotically faster than $f(n)$ as lower bound.



Big-Omega Examples (1)

- **Claim)** $T(n) = 5n^2 = \Omega(n^2)$
  - Let $c = 4$ and $n_0 = 1$; then, $5n^2 \geq 4n^2$ for all $n \geq n_0 = 1$.

- **Claim)** $T(n) = 5n + 3 = \Omega(n)$
  - Let $c = 1$; then, $5n + 3 \geq n \Leftrightarrow 4n \geq -3$ for all n.
  - Any $n_0 > 0$ can be good, e.g., $n_0 = 1$.

- **If a polynomial has the term of largest degree $\geq n^r$, then it is $\Omega(n^r)$.**

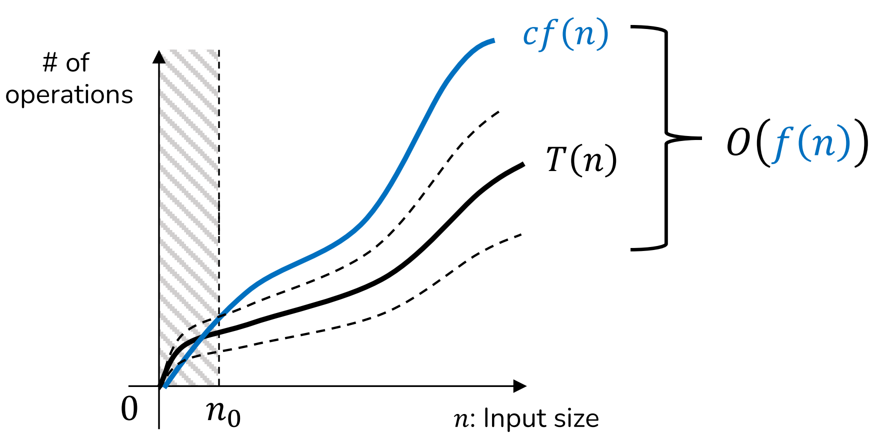
Big-Omega Examples (2)

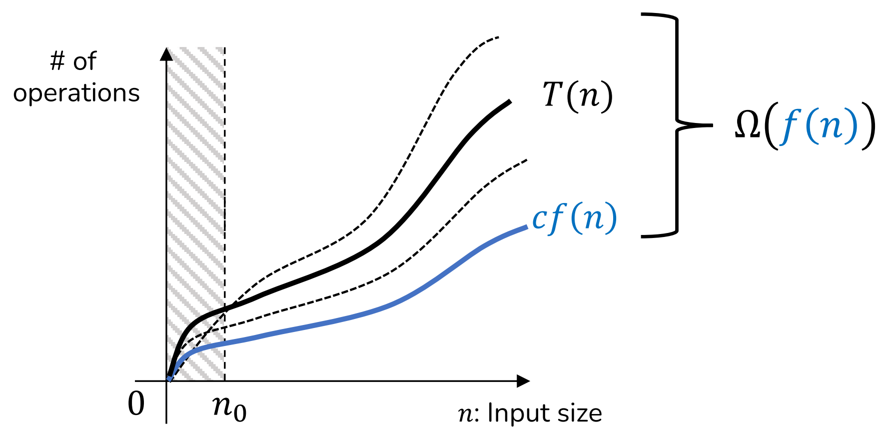
- **Claim)** $T(n) = 5n^3 + 3 = \Omega(n^2)$
  - Let $c = 1$; then, $5n^3 + 3 \geq n^2$ for all $n \geq n_0 = 1$.
  - Big-Omega also results in loose lower bound as above.
    - $T(n) = 5n^3 + 3 = \{\cdots, \Omega(n), \Omega(n^2), \Omega(n^3)\}$
  - Like Big-O, estimate Big-Omega notation as tight as possible!

Big-O v.s. Big-Omega

❑ Difference between Big-O and Big-Omega

- Big-O tells us **asymptotic upper bound**
  - The algorithm of $T(n)$ does not compute beyond the upper bound
- Big-Omega tells us **asymptotic lower bound**
  - The algorithm of $T(n)$ computes beyond the lower bound





- **Can we can get more precise bound?**

Outline

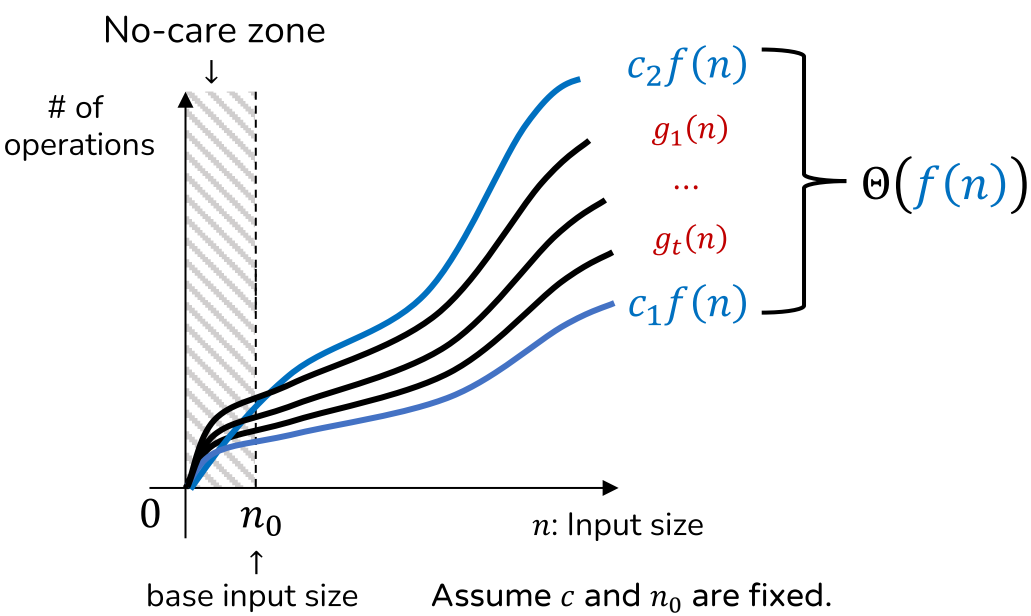
- Big-Omega Notation
- Big-Theta Notation
- Simplifying Rules
- Other Discussions

Big-Theta Notation (1)

❑ Definition of $\Theta(f(n))$

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$

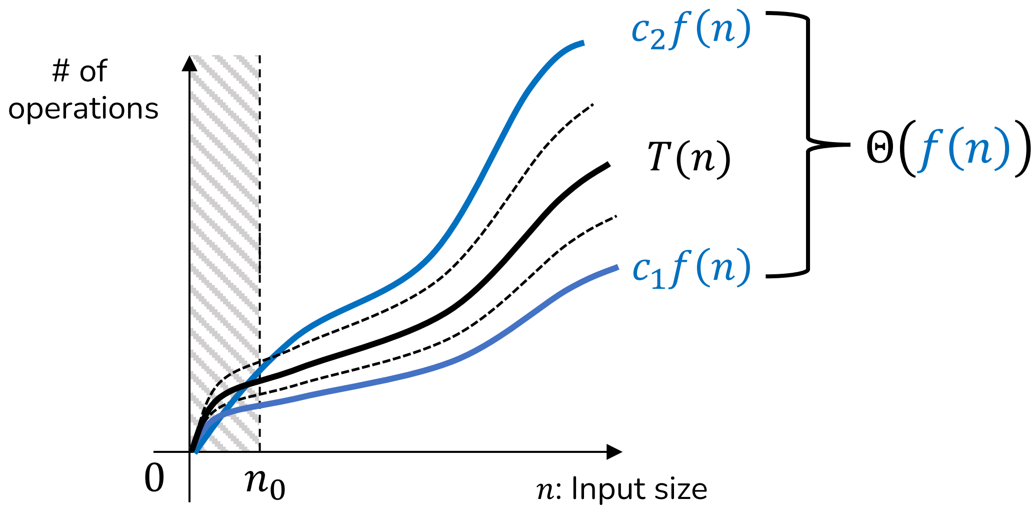
- Set of $c_1 f(n) \le \underbrace{\text{functions}}_{g(n)} \le c_2 f(n)$ for all $n \ge n_0$



Assume c and n_0 are fixed.

Big-Theta Notation (2)

- Interpretation of $T(n) = \Theta(f(n))$
  - The time complexity $T(n)$ of the algorithm is in $\Theta(f(n))$ for [best | average | worst] case.
    - When the input size is large enough, its complexity is proportional to $cf(n)$ for the case.
    - $T(n)$ grows asymptotically as fast as $f(n)$ as exact bound.



Big-Theta Examples

- **Claim)** $T(n) = 5n^2 = \Theta(n^2)$
  - Proof)
    - $5n^2 = O(n^2)$ and $5n^2 = \Omega(n^2)$
    - Thus, $T(n) = 5n^2 = \Theta(n^2)$ by its definition

- **Claim)** $T(n) = 5n + 3 = \Theta(n)$
  - Proof)
    - $5n + 3 = O(n)$ and $5n + 3 = \Omega(n)$
    - Thus, $T(n) = 5n + 3 = \Theta(n)$ by its definition

Discussion

❑ Try to obtain Big-Theta for worst case

- Big-Theta provides **asymptotic exact bound** so that we can expect precise asymptotic behavior of an algorithm

- Compare algorithms in terms of **Big-Theta notation for worst case**

- If Big-O and Big-Omega are not the same or it is not easy to estimate Big-Omega, then
    - Compare algorithms in terms of Big-O notation for worst case

Outline

- ❑ Big-Omega Notation
- ❑ Big-Theta Notation
- ❑ **Simplifying Rules**
- ❑ Other Discussions

Simplifying Rules (1)

❑ Rule 1

- Polynomial: $T(n) = c_p n^p + c_{p-1} n^{p-1} + \cdots c_1 n + c_0$
  - If $T(n)$'s largest term is $\leq n^r$, then $T(n) = O(n^r)$.
  - If $T(n)$'s largest term is $\geq n^r$, then $T(n) = \Omega(n^r)$.
- Implying if $T(n)$'s largest term is n^r, then $T(n) = \Theta(n^r)$.
  - e.g., $T(n) = 12n^4 + n^3 + 2n^2 = \Theta(n^4)$

❑ Rule 2

- If $f(n) \in O\big(g(n)\big)$ and $g(n) \in O\big(h(n)\big)$, then $f(n) \in O\big(h(n)\big)$.
  - e.g., $n \in O(n^2)$ and $n^2 \in O(n^3) \Rightarrow n \in O(n^3)$
- Rules 2-5 also hold for Ω and Θ

Simplifying Rules (2)

❑ Rule 3

- If $f(n) \in O\big(kg(n)\big)$ for constant $k > 0$, then $f(n) \in O\big(g(n)\big)$.
  - e.g. $n^3 + 2n^2 \in O(kn^3) \Rightarrow n^3 + 2n^2 \in O(n^3)$

❑ Rule 4

- If $f_1(n) \in O\big(g_1(n)\big)$ and $f_2(n) \in O\big(g_2(n)\big)$,
  then $f_1(n) + f_2(n) = (f_1 + f_2)(n) \in O\Big(\max\big(g_1(n), g_2(n)\big)\Big)$.
  - Used when two parts of a program run in sequence

$f_1(n) \in O(n)$

$f_2(n) \in O(n^2)$

$$(f_1 + f_2)(n) \in O(\max(n, n^2)) = O(n^2)$$

Simplifying Rules (3)

❑ Rule 5

- If $f_1(n) \in O\big(g_1(n)\big)$ and $f_2(n) \in O\big(g_2(n)\big)$, then $f_1(n) \times f_2(n) \in O\big(g_1(n) \times g_2(n)\big)$.
  - Used to analyze for-loops

```
for(i = 1; i <= n; i++)
    for(j = 1; j <= n; j++)
        do something in O(1)
```

$f_2(n) \in O(n)$

$f_1(n) \in O(n)$

$f_1(n) \times f_2(n) \in O(n \times n) = O(n^2)$

Simplifying Rules (4)

❑ Rule 5

- If $f_1(n) \in O\big(g_1(n)\big)$ and $f_2(n) \in O\big(g_2(n)\big)$, then $f_1(n)\times f_2(n) \in O\big(g_1(n)\times g_2(n)\big)$.
  - But, it can be overestimated for some complicated cases
  - In this case, we should directly count the number of operations

```
for(i = 1; i <= n; i*=2)
    for(j = 1; j <= i; j++)
        do something in O(1)
```

$f_2(n) \in O(i) \in O(n)$

$f_1(n) \in O(\log n)$

$f_1(n)\times f_2(n) \in O(n \log n)$

Assume $n = 2^K$

Then,

$$T(n) = 1 + 2 + 2^2 + \cdots + 2^K$$
$$= \frac{2^{K+1} - 1}{2 - 1}$$
$$= 2n - 1 \in O(n)$$

Analysis Examples (1)

❑ Sequential search problem

- **Input**: an array of size n, having keys & a querying key
- **Output**: the index for the querying key in the array

```
def sequential_search(array, n, key):
    for i in range(0, n):
        if array[i] == key:
            return i

    throw "out-of-key"
```

  - Best **case**: $T(n) = 1 = O(1) = \Omega(1) = \Theta(1)$
  - Worst **case**: $T(n) = n = O(n) = \Omega(n) = \Theta(n)$
  - Average **case**: $T(n) = \frac{n+1}{2} = O(n) = \Omega(n) = \Theta(n)$

Analysis Examples (2)

❑ Example 1

- $T(n) = \Theta(n)$

```
sum = 0;
for(i = 1; i <= n; i++)
    sum += n;
```

❑ Example 2

- $T(n) = \Theta(n^2)$

```
sum = 0;
for(i = 1; i <= n; i++)
    for(j = 1; j <= n; j++)
        sum += 1;

for(k = 1; k <= n; k++)
    A[k] = k;
```

Analysis Examples (3)

❑ Example 3

- $T(n) = \Theta(n^2)$

```
sum = 0;
for(i = 1; i <= n; i++)
    for(j = 1; j <= i; j++)
        sum += 1;
```

❑ Example 4 (assume $n = 2^K$)

- $T(n) = \Theta(n \log n)$

```
sum = 0;
for(i = 1; i <= n; i *= 2)
    for(j = 1; j <= n; j++)
        sum += 1;
```

Outline

- ❑ Big-Omega Notation
- ❑ Big-Theta Notation
- ❑ Simplifying Rules
- ❑ **Other Discussions**

Other Control Statements

❑ `while` loop

- Analyze like a `for` loop.

❑ `if` statement

- Take greater complexity of `then/else` clauses.

❑ `switch` statement

- Take complexity of the most expensive case.

❑ Subroutine (function) call

- Take complexity of the subroutine.

Multiple Parameters

❑ When the input size consists of multiple parameters

- e.g., 2D-array ($n \times m$ matrix), its size parameters are n and m.
- Describe the complexity with respect to n and m.
  - e.g., $T(n, m)$ and $S(n, m)$

❑ Example

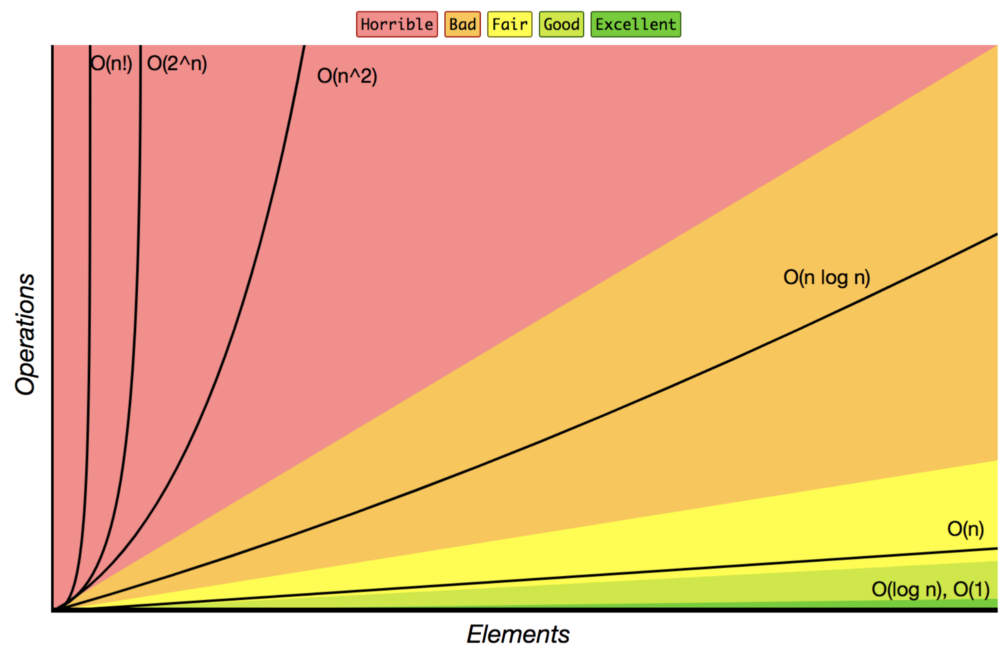
- Time complexity: $T(n, m) = \Theta(n \times m)$
- Space complexity: $S(n, m) = \Theta(n \times m)$

```
sum = 0;
for(i = 1; i <= n; i++)
    for(j = 1; j <= m; j++)
        sum += A[i][j];
```

Complexity Category

- Complexities that frequently appear are categorized as follows:

Base Func.	Name	Scalability
1	Contant	Good
$\log n$	Logarithmic	↑
n	Linear	
$n \log n$	Log-linear	
n^2	Quadratic	
n^3	Cubic	
n^p	Polynomial	
2^n	Exponential	↓
$n!$	Factorial	Poor



What You Need To Know

❑ Asymptotic Notations

- Prove claims using the definitions of O, Ω, and Θ.

❑ Simplifying Rules

- Quickly analyze complexities using the simplifying rules

❑ Other Discussions

- Analysis with control statements and multiple parameters
- Understand which complexities are good for scalability

In Next Lecture

❑ Concept of recursion

- What is recursion?
- Why do we need recursion?

❑ How to design and analyze recursion

- Divide and conquere
- Mathematical induction
- Recursive complexity

Thank You