Lecture #1
Algorithm Analysis

Algorithm

JBNU Spring 2021
Jinhong Jung

In Previous Lecture

What is algorithm?

- To describe the sequntial process for solving a problem using a modern computer(s)
  - Input ⇒ algorithm ⇒ output
- Must clarify the input and output of your problem

Algorithm should be

- Easy-to-understand
  - Able to implement an algorithm using a programming language
- Correctly designed & efficient
  - Must produce correct answers
  - Should be practically efficient for large size of input

How can we measure the efficiency?

In This Lecture

Efficiency of algorithms

- What is the efficiency? How to measure the efficiency?
  - Empirically and theoretically

Computational complexities

- Why should we take care about the size of input?

Best, average, and worst cases

- Which case is important for algorithm analysis?

Asymptotic analysis

- How to express complexities in simple & uniform ways?
  - While considering the large size of input at the same time
- Concept of Big-O, Omega, and Theta bounds

Outline

Background on Algorithm Analysis

Asymptotic Analysis

Efficiency Of Algorithm

A problem can be solved by many algorithms

- **Problem**: What if the number n is added n times?
  - **Input**: the number n
  - **Output**: a number that n is added n times

Algorithm A	Algorithm B	Algorithm C

`sum ← 0`
`for i in range(0, n):`
`    sum ← sum + n`
	`sum ← 0`
	`for i in range(0, n):`
	`    for j in range(0, n):`
	`        sum ← sum + 1`
		`sum ← n × n`

- **Q: Which algorithm should we use among them?**
  - A: The fastest and lightest one (i.e., the best)
- **Q: How can we know which of them is the best?**
  - **A1**: Let's directly run and compare them on a computer!
  - **A2:** Algorithm C since it takes $O(1)$ time and space

Time & Space Costs

Choice of data structure or algorithm can make the difference between programs

- Running in a few seconds or many days
- Consuming a little memory or huge memory space

A solution is said to be efficient

- If it solves the problem within its resource constrains

Resource	Time	Space
Empirical	Wall-clock time	Memory usage
Theoretical	Time complexity	Space complexity

The (time | space) cost of a solution

- The amount of resources that the solution consumes

Measure efficiency ⇔ Measure time & space costs

How To Measure Efficiency (1)

Empirical Measurement

- e.g., measure the runtime of a program
- e.g., check the maximum memory usage (i.e., VmPeak)

Empirical Time Cost (wall-clock time)

```
start_time = tic

Run an algorithm to be measured

run_time = toc - start_time
```

Empirical Space Cost (memory usage)

```
cat /proc/$pid/status
--------------------------
VmPeak: 67380632 kB <<
VmSize:     6552 kB
VmData: 67376304 kB
VmStk:       132 kB
```

- Pros: Easy-to-check & practically intuitive
- Cons
  - Varied by environment (HW, OS, PL, ...) and how to implement
  - Hard to know the tendency of performance for the size of input
- **Can we know the efficiency without directly running it?**

How To Measure Efficiency (2)

Theoretical Measurement

- **Complexity analysis** in terms of time & space
  - **Time complexity = the number of basic operations**
    - e.g., the number of additions or multiplications
  - **Space complexity = the amount of data to be stored or used**
    - e.g., the size of an array where the input data are stored

- In general, the computational complexities of an algorithm depend on the size n of input data
  - $T(n)$: time complexity (function) for given n input data
  - $S(n)$: space complexity for given n input data
    - Most of the times, it is proportional to the input size (for data structures)

Basic Operations

Basic operation runs in constant time C regardless of the input size

- Add/subtraction ($+$ or $-$) & division/multiplication ($/$ or $\times$)
  - For $a + b$ or $a \times b$, its # of operations is a constant (i.e., 1)

- Assignment ($=$ or $\leftarrow$)
  - For $c = 10$, its # of operations is a constant (i.e., 1)
  - For $c \leftarrow a + b$, its # of operations is a constant (i.e., 2)

- Comparison ($<$ or $>$)
  - For $c > b$, its # of operations is a constant (i.e., 1)

- …… ……

Example Of Complexity Analysis

Problem: What if the number n is added n times?

- Let's analyze the time complexity of each algorithm

Algorithm A	Algorithm B	Algorithm C

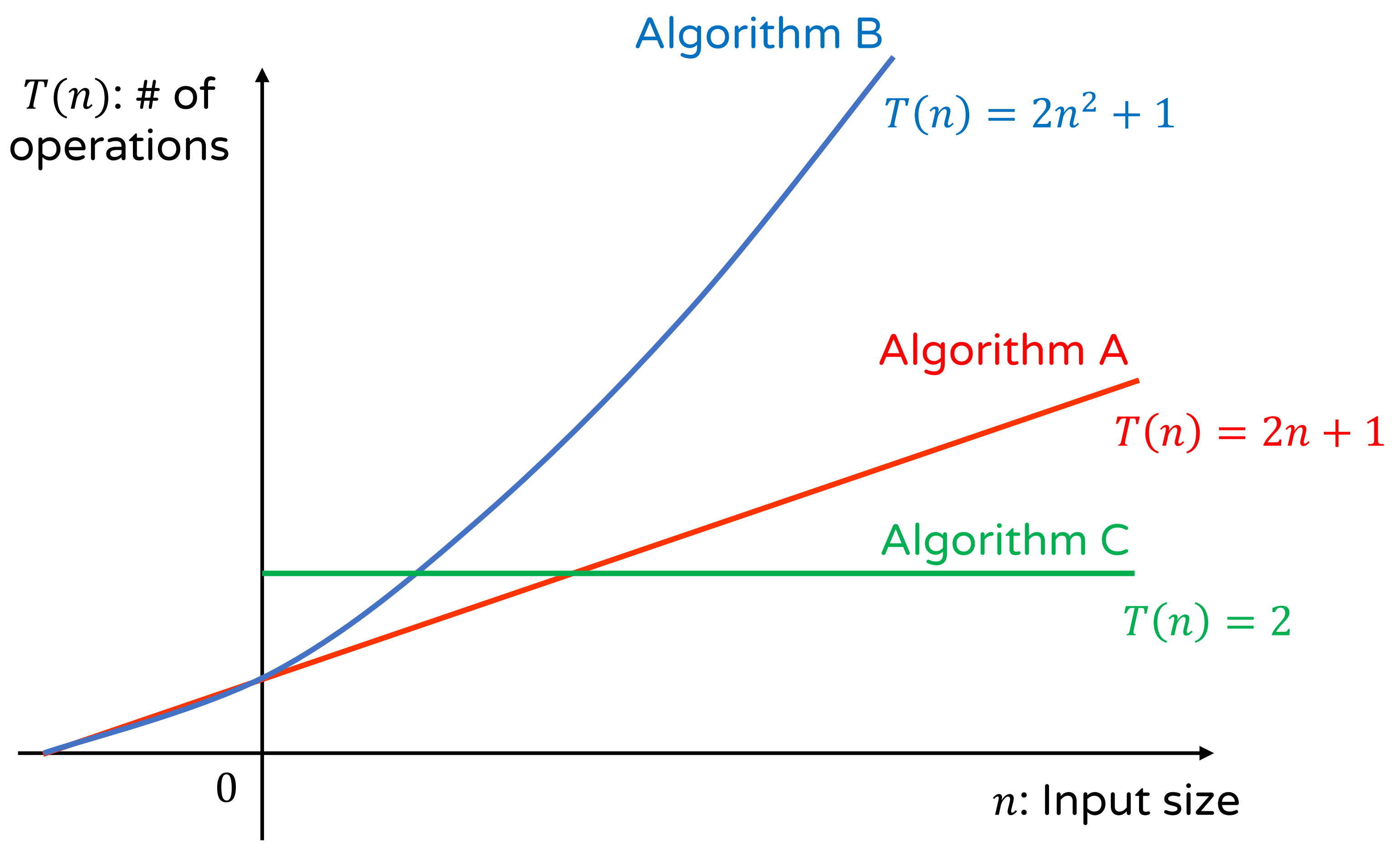
`sum ← 0`
`for i in range(0, n):`
`    sum ← sum + n`
	`sum ← 0`
	`for i in range(0, n):`
	`    for j in range(0, n):`
	`        sum ← sum + 1`
		`sum ← n × n`

- Count the number of basic operations := $T(n)$

	Algorithm A	Algorithm B	Algorithm C
Assignments	$n + 1$	$n \times n + 1$	1
Additions	n	$n \times n$	
Multiplications			1
Total	$2n + 1$	$2n^2 + 1$	2

Performance Tendency

Let's represent the # of operations as a graph

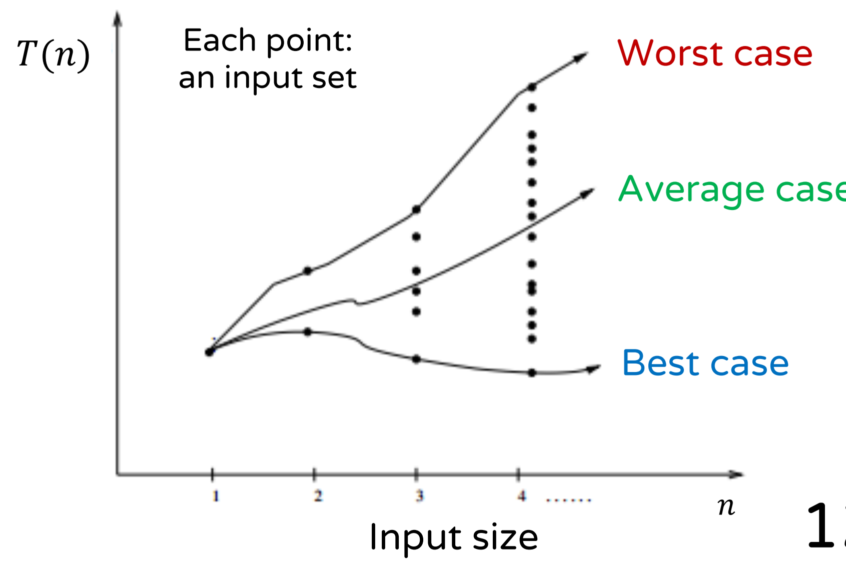


Best, Average, & Worst Cases

Complexities can be different according to inputs

- Best case: input sets consuming **the least resources**
  - Easy to obtain, but hard to judge its general performance
- Average case: input sets exhibiting **the average cost**
  - Can indicate precise performance, but hard to calculate in general
- Worst case: input sets consuming **the largest resources**
  - Easy to obtain, but can be loosely estimated (when it is rare)
  - Guarantee that the algorithm for all inputs takes time/space less than or equal to the worst case

Most of the times, **we should do analysis for the worst case**



Example For Cases

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

    return -1 # when the array doesn't have the key
```

- **Best case**: $T(n) = 1$ when the array has the querying key at the first
- **Worst case**: $T(n) = n$ when the array has it at the end or no the key
- **Average case**: $T(n) = \frac{n+1}{2}$, i.e., the expectation for all possible cases

$$\frac{1}{n}\times 1 + \frac{1}{n}\times 2 + \cdots + \underset{\uparrow}{\frac{1}{n}}\times \underset{\uparrow}{i} + \cdots + \frac{1}{n}\times n = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\times\frac{n(n+1)}{2} = \frac{n+1}{2}$$

P(the key is at index i)

# of operations searching for index i

Summary

A solution is said to be efficient

- If it solves the problem within its resource constrains

How to measure the efficiency?

- Empirical measurement
  - Hard to know the tendency of performance for the size of input
- Computational complexity analysis
  - $T(n)$: time complexity = the number of basic operations
  - $S(n)$: space complexity = the amount of data to be used

Best, average, and worst cases

- Should do analysis for **the worst case**
- **Guarantee that the algorithm for all inputs** takes time/space less than or equal to the worst case

Outline

Background on Algorithm Analysis

Asymptotic Analysis ☜

- Big-O notation
- Big-Omega notation
- Big-Theta notation

Motivation

Q. Which of the following is faster?

- Algorithm A: # of operations is 2^n, i.e., $T_A(n) = 2^n$
- Algorithm B: # of operations is n^{10}, i.e., $T_B(n) = n^{10}$
  - If $n = 10$, $T_A(10) = 2^{10} = 1024 \ll T_B(10) = 10^{10} = 10$ billion (10^9)
  - If $n = 60$, $T_A(60) = 2^{60} \approx 1.15 \times 10^{18} > T_B(60) = 60^{10} \approx 6.05 \times 10^{17}$
  - If $n = 100$, $T_A(100) = 2^{100} \approx 10^{30} \gg T_B(100) = 100^{10} = 10^{20}$

If the input size n becomes extremely large, then Alg. B is faster “**eventually**” than Alg. A

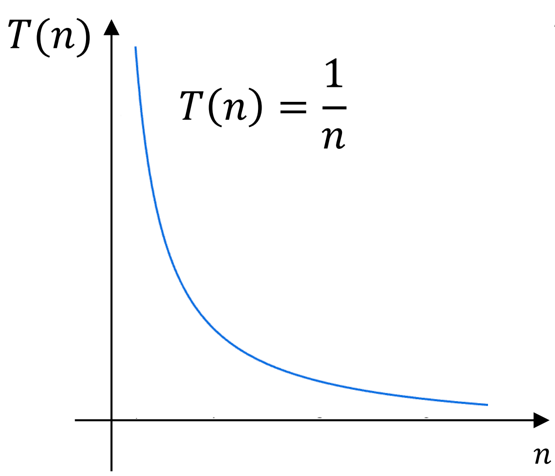
Asymptotic analysis (점근적 분석)

- Aim to analyze the efficiency of an algorithm when the input size becomes very large

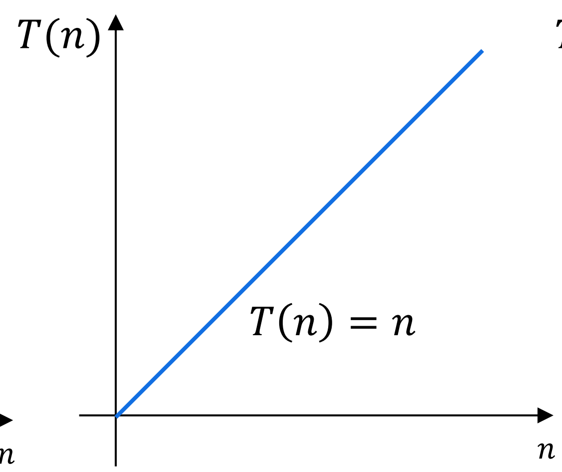
Asymptotic Analysis

To analyze how a complexity function of the input size n changes as n becomes large

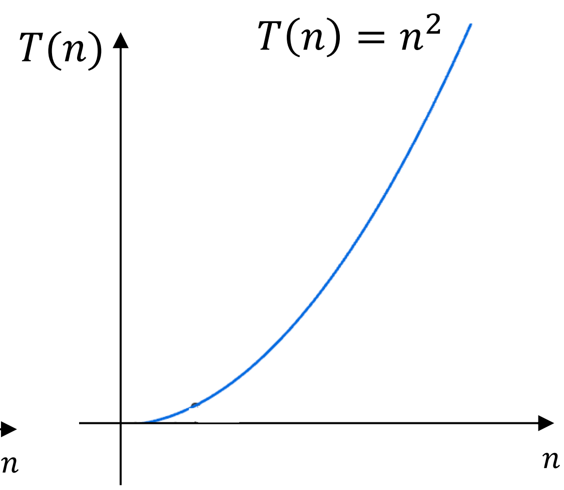
- **Asymptotic**: to approach an infinity point (i.e., $n \to \infty$)
- As $n \to \infty$, how the function changes is called **asymptotic (or limiting/tail) behavior**



As $n \to \infty$,
it converges to 0



As $n \to \infty$,
it keeps increasing
"linearly"



As $n \to \infty$,
it keeps increasing
"quadratically"

Why Need To Consider ∞?

Q. What if the complexity function consists of multiple sub-functions?

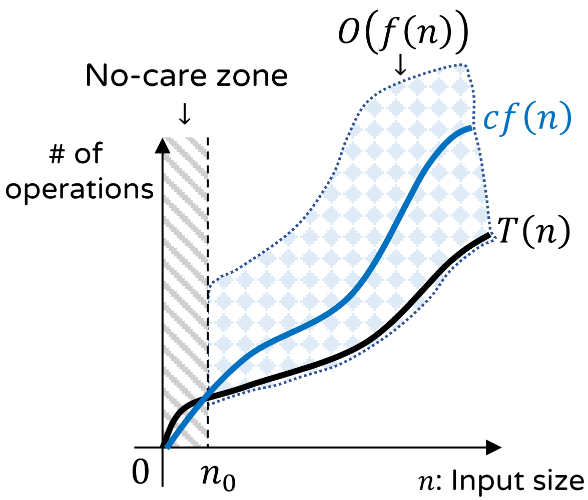
- Example: $T(n) = n^2 + n + 1$
  - $n = 1$ $T(n) = 1 + 1 + 1 = 3$ (33.3% for n^2)
  - $n = 10$ $T(n) = 100 + 10 + 1 = 111$ (90% for n^2)
  - $n = 100$ $T(n) = 10000 + 100 + 1 = 10101$ (99% for n^2)
  - $n = 1{,}000$ $T(n) = 1000000 + 1000 + 1 = 1001001$ (99.9% for n^2)
- In other words, $T(n)$ is proportional to n^2 as $n \to \infty$
  - n^2 is considered as a **dominating factor** having the largest exponent in general

- Using asymptotic analysis, **it's possible to know how the complicated function behaves** eventually

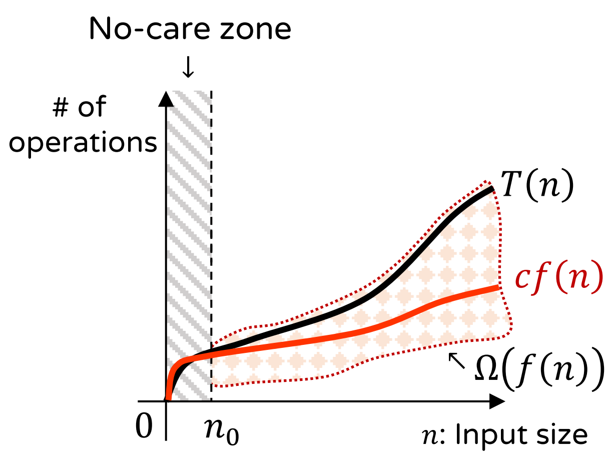
Asymptotic Bounding

Q. Then, how can we simply describe the limiting behavior of an arbitrary complexity function?

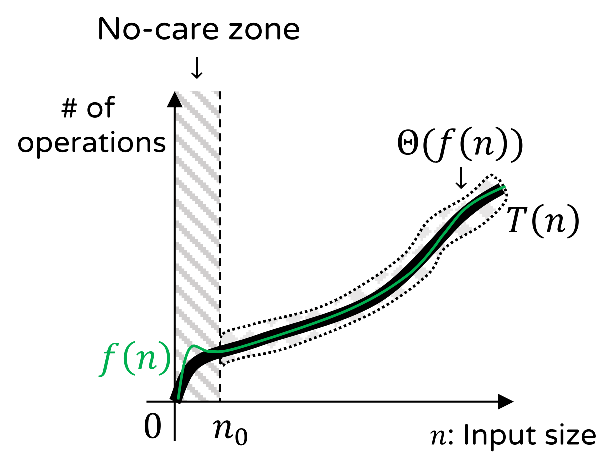
- Big-O (O, upper bound)
- Big-Omega (Ω, lower bound)
- Big-Theta (Θ, exact bound)



Big-O (O, upper bound)



Big-Omega (Ω, lower bound)



Big=Theta (Θ, exact bound)

⚠ These are pictorial examples that help you visually understand the concept when n_0 is fixed (i.e., it's not that strict)

Big-O Notation

$T(n) \in O(f(n))$

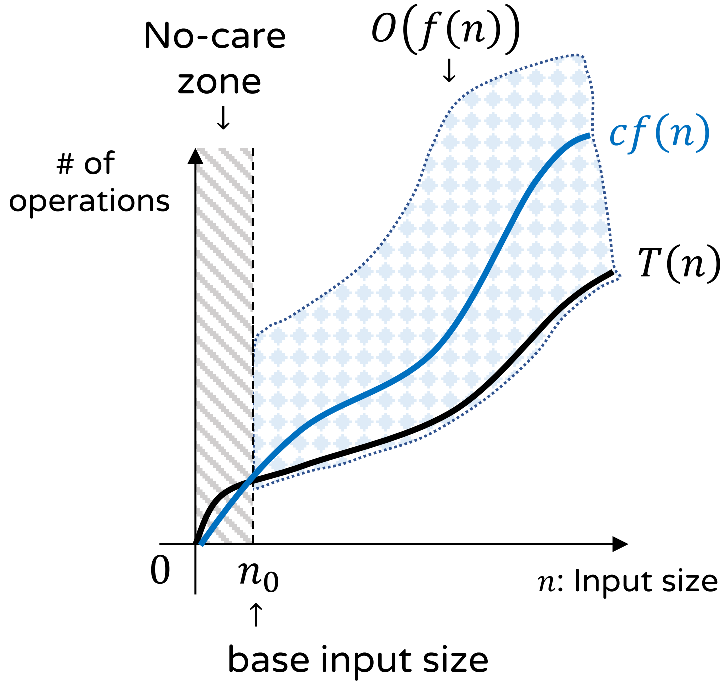
Definition of $T(n) = O(f(n))$ [as $n \to \infty$]

$T(n)$ is in the set $O(f(n))$

if there exist two positive constants c and n_0

such that $T(n) \leq cf(n)$ for all $n \geq n_0$

$\Rightarrow$ Indicating the input size is large enough



Usage of big-O notation

- The algorithm is in $O(f(n))$ for [best | average | worst] case

Interpretation of big-O notation

- When the input size is large enough, it always executes in less than $cf(n)$ for [best | average | worst] case

- $T(n)$ grows asymptotically no faster than $f(n)$ as upper bound

Big-O Examples (1)

Claim) $T(n) = 5n^2 = O(n^2)$

- **Proof)** Intuitively pick c and n_0 so that $c = 6$ and $n_0 = 1$; then, for all $n \geq n_0 = 1$, $T(n) = 5n^2 \leq cn^2 = 6n^2$
  - In this proof, $c = 6$ & $n_0 = 1$ is one of numerous answer candidates
  - Any c and n_0 can be an answer if they satisfy the definition
    - e.g., $c = 7$ & $n_0 = 1$

Claim) $T(n) = 4 = O(1)$

- Proof) Suppose $c = 10$ and $n_0 = 1$;
  Then, for all $n \geq n_0 = 1$, $T(n) = 4 \leq c \times 1 = 10$

- Say “it takes constant time” in this case

Big-O Examples (2)

Claim) $T(n) = 3n^2 + 100 = O(n^2)$

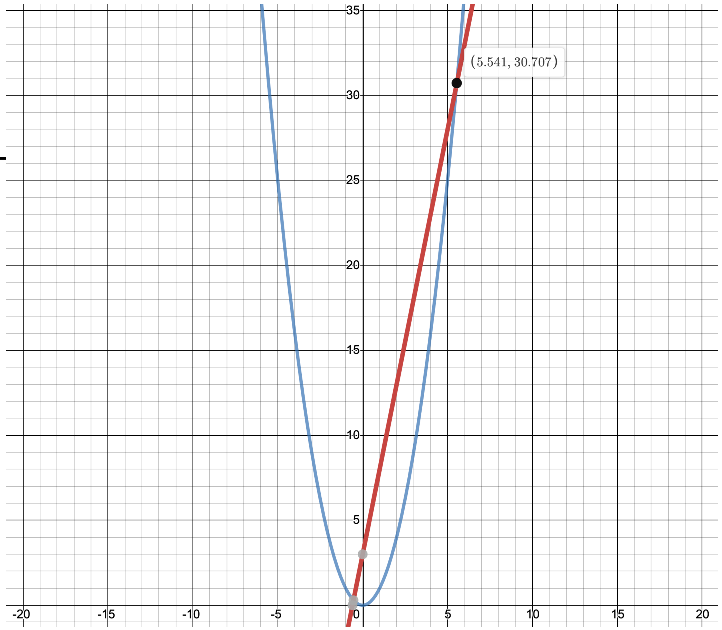
- Proof 1)
  - Suppose $n_0 = 100$ and $c = 5$; for all $n \geq 100$, $3n^2 + 100 \leq cn^2 = 5n^2$
- Proof 2)
  - $3n^2 + 100 \leq 3n^2 + 100n^2 = 103n^2 \Rightarrow c = 103$ for all $n \geq n_0 \geq 1$
    - Any $n_0 \geq 1$ is good in this case (e.g., $n_0 = 1$ or $n_0 = 2$)
- Proof 3)
  - First, let $c = 13$; then, $3n^2 + 100 \leq 13n^2 \Leftrightarrow 100 \leq 10n^2 \Leftrightarrow 10 \leq n^2$
  - This indicates $n \geq \sqrt{10} \approx 3.162 \Rightarrow n_0 = 4$
  - Then, for all $n \geq 4$, $3n^2 + 100 \leq 13n^2$

If a polynomial has the term of largest degree $\leq n^r$, then it is $O(n^r)$

Big-O Examples (3)

Claim) $T(n) = 5n + 3 = O(n^2)$

- Proof)
  - Suppose $c = 1$
  - Then, $5n + 3 \le n^2$ for all $n \ge n_0 = 6$



This problem implies that

- Big-O bound can be either of **strict** or **loose** upper bound
  - By which base function $f(n)$ is used
- Example: $T(n) = 3n^2$
  - $T(n) = 3n^2 = \{O(n^2), O(n^3), O(n^4), \cdots\}$
  - If a problem says like "estimate Big-O bound **as tight as possible**", you should write it like $T(n) = O(n^2)$
    - Do likewise for Big-Omega bound as well

Big-Omega Notation

$T(n) \in \Omega(f(n))$

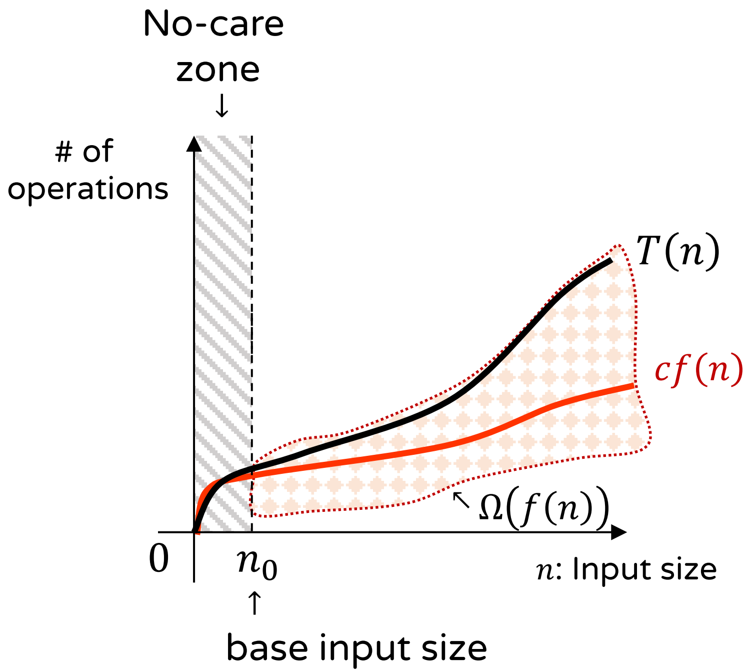
Definition of $T(n) = \Omega(f(n))$ [as $n \to \infty$]

$T(n)$ is in the set $\Omega(f(n))$

if there exist two positive constants c and n_0

such that $T(n) \geq cf(n)$ for all $n \geq n_0$

⇒ Indicating the input size is large enough



Usage of big-Omega notation

- The algorithm is in $\Omega(f(n))$ for [best | average | worst] case

Interpretation of big-Omega notation

- When the input size is large enough, it always requires **more than** $cf(n)$ for [best | average | worst] case

- $T(n)$ grows asymptotically faster than $f(n)$ as lower bound

Big-Omega Examples

Claim) $T(n) = 5n^2 = \Omega(n^2)$

- Proof)
  - Suppose $c = 4$ and $n_0 = 1$; then, $5n^2 \geq 4n^2$ for all $n \geq n_0 = 1$

Claim) $T(n) = 5n^2 + 3 = \Omega(n^2)$

- Proof)
  - Let $c = 1$; then, $5n^2 + 3 \geq n^2 \Leftrightarrow 4n^2 \geq -3$ for all natural numbers n
  - Thus, any $n_0 > 0$ can be good, e.g., $n_0 = 1$

Claim) $T(n) = 5n^3 + 3 = \Omega(n^2)$

- Proof)
  - Let $c = 1$; then, $5n^3 + 3 \geq n^2$ for all n; thus, any $n_0 > 0$, e.g., $n_0 = 1$
- This is also an example of a loose lower bound
  - If a polynomial has the term of largest degree $\geq n^r$, then it's $\Omega(n^r)$

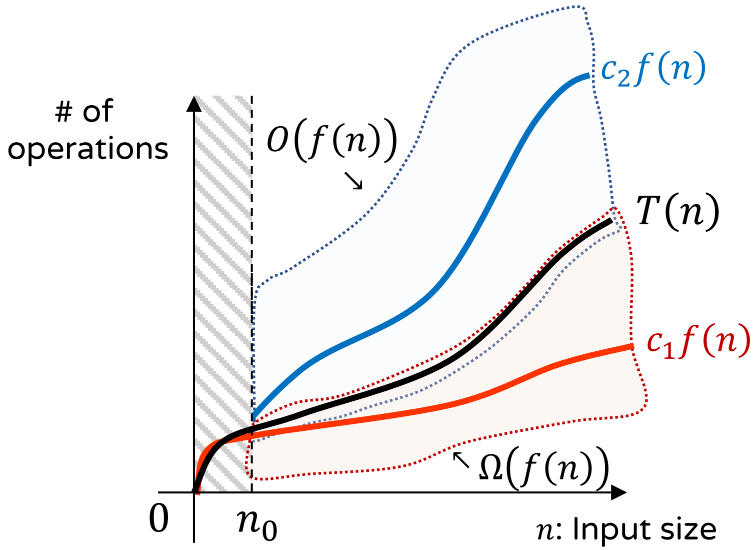
Big-Theta Notation

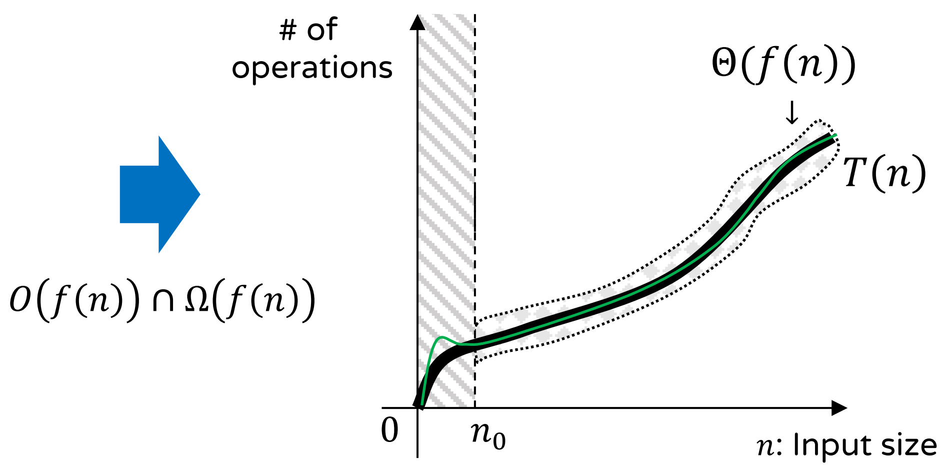
Definition of $T(n) = \Theta(f(n))$ [as $n \to \infty$]

- $T(n)$ is in the set $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

  if there exist two positive constants c and n_0

  such that $c_1 f(n) \le T(n) \le c_2 f(n)$ for all $n \ge n_0$





$T(n)$ 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, $5n^2 = \Theta(n^2)$ by its definition

Claim) $T(n) = 5n^2 + 3 = \Theta(n^2)$

- **Proof)**
  - $5n^2 + 3 = O(n^2)$ and $5n^2 + 3 = \Omega(n^2)$; thus, $5n^2 + 3 = \Theta(n^2)$

Interesting fact for Big-Theta

- Note that
  - If a polynomial has the term of largest degree $\leq n^r$, then it's $O(n^r)$
  - If a polynomial has the term of largest degree $\geq n^r$, then it's $\Omega(n^r)$
- Implying that if a polynomial's largest degree term is n^r, then it's $\Theta(n^r)$!

Example With Big Bounds (1)

Problem: What if the number n is added n times?

- Let's analyze the time complexity of each algorithm

Algorithm A	Algorithm B	Algorithm C

`sum ← 0`
`for i in range(0, n):`
`    sum ← sum + n`
	`sum ← 0`
	`for i in range(0, n):`
	`    for j in range(0, n):`
	`        sum ← sum + 1`
		`sum ← n × n`

- Count the number of basic operations := $T(n)$

	Algorithm A	Algorithm B	Algorithm C
Assignments	$n+1$	$n \times n + 1$	1
Additions	n	$n \times n$	
Multiplications			1
Total	$2n+1$	$2n^2+1$	2
Big Bounds	$O(n) = \Omega(n) = \Theta(n)$	$O(n^2) = \Omega(n^2) = \Theta(n^2)$	$O(1) = \Omega(1) = \Theta(1)$

Example With Big Bounds (2)

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

    return -1 # when the array doesn't have the 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**: $T(n) = \frac{n+1}{2} = O(n) = \Omega(n) = \Theta(n)$

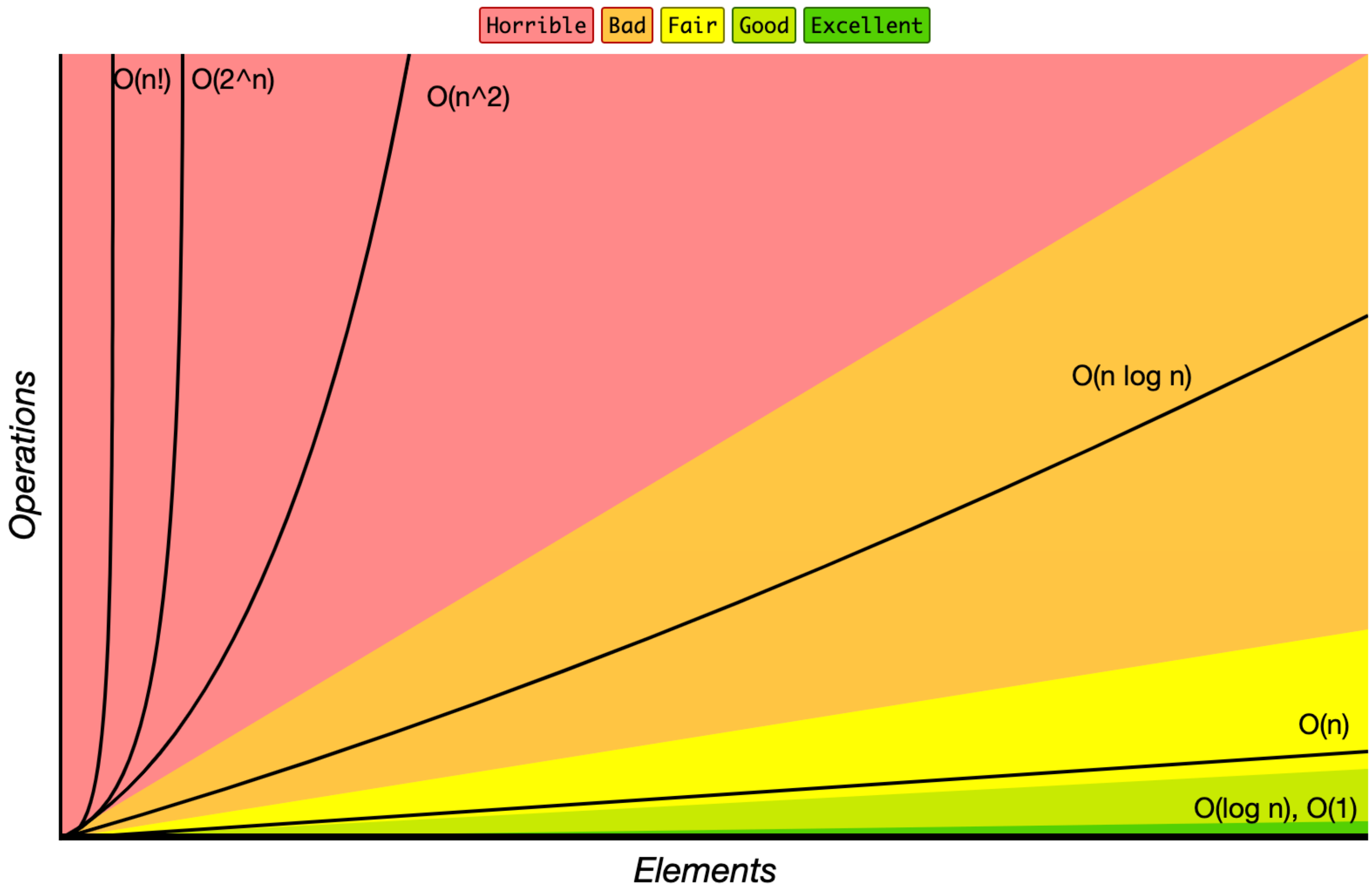
At Least Get Big-O Bound

Mostly, we use Big-O notation a lot to analyze the efficiency of an algorithm

- If possible, try to obtain Big-Theta at the first
- But it's hard to obtain Big-Omega for some problems
  - Implying that it's hard to get Big-Theta too
- If impossible, Big-O bound is enough
  - Since we can still guess that the algorithm's efficiency is unlikely to be worsen than the upper bound
  - Must obtain the Big-O bound **as tight as possible** for **the worst (input) case**

Big-O Complexity Chart

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$$



Reference: https://www.bigocheatsheet.com/

Summary

Big-O implies an upper bound

- It always executes in less than the upper bound
- $O(n^2)$: $T(n)$ grows asymptotically no faster than n^2

Big-Omega implies a lower bound

- It always requires more than than the lower bound
- $\Omega(n^2)$: $T(n)$ grows asymptotically faster than n^2

Big-Theta implies an exact bound

- It performs operations as much as the exact bound
- $\Theta(n^2)$: $T(n)$ grows asymptotically as fast as n^2

Get Big-Theta bound if possible; otherwise, obtain Big-O bound for worst case as tight as possible

Outline

Background on Algorithm Analysis

Asymptotic Analysis

- Big-O notation
- Big-Omega notation
- Big-Theta notation
- Little-o notation
- Little-omega notation

Little-o Notation

Describe the growthrate of $f(n)$ is asymptotically less than $g(n)$

- $5n = o(n^2)$, the growthrate of $5n$ is less than that of n^2
- $0.5n^2 \neq o(n^2)$ since the growthrate of $0.5n^2$ = that of n^2

Definition of $T(n) = o\big(g(n)\big)$

$$o\big(g(n)\big) = \left\{ f(n) \middle| \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \right\}$$

- Set of all functions $f(n)$ overwhelmed by $g(n)$
  - Interprested as loose upper bound

- If an alg. is $o(n \log n)$, its speed cannot exceed $n \log n$
  - Tighter and stricter than $O(n \log n)$ (can prevent information loss)

Little-omega Notation

Describe the growthrate of $f(n)$ is asymptotically greater than $g(n)$

- $n^2 = \omega(n)$, the growthrate of n^2 is greater than that of n
- $2n^2 \neq \omega(n^2)$ since the growthrate of $2n^2$ = that of n^2

Definition of $T(n) = \omega\big(g(n)\big)$

$$\omega\big(g(n)\big) = \left\{ f(n) \middle| \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \right\}$$

- Set of all functions $f(n)$ overwhelming $g(n)$
  - Interprested as loose lower bound

- If an alg. is $\omega(n)$, its speed **must** exceed n
  - Tighter and stricter than $\Omega(n)$ (can prevent information loss)

Example of Little-o and Little-ω

Claim. $n^2 - 5 = o(n^3)$

- Proof)
  - $f(n) = n^2 - 5$ and $g(n) = n^3$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2-5}{n^3} = 0$; thus, $n^2 - 5 = o(n^3)$ by its definition

Claim. $\frac{n^3}{4} = \omega(n^2)$

- Proof)
  - $f(n) = \frac{n^3}{4}$ and $g(n) = n^2$
  - $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{1}{4}\frac{n^3}{n^2} = \infty$; thus, $\frac{n^3}{4} = \omega(n^2)$ by its definition

What You Need To Know

Efficiency of algorithms

- What is the efficiency? How to measure the efficiency?

Computational complexities

- Why should we take care about the size of input?

Best, average, and worst cases

- Which case is important for algorithm analysis?

Asymptotic analysis

- How to express complexities in simple & uniform ways?
  - While considering the large size of input at the same time
- Concept of Big-O, Omega, and Theta bounds
  - + little-o and little-ω

In Next Lecture(s)

Recursive Algorithm & Reccurence

How to analyze a recursive complexity function

- Substitute method
- Mathmetical induction
- Master theorem

Thank You