Lecture #1
Algorithm Analysis (1)

Algorithm

JBNU

Jinhong Jung

In This Lecture

Algorithm efficiency

- What is the efficiency? Why should we care about it?
- How to measure the efficiency? What is the complexity?

Best, average, and worst cases

- Which case is important for algorithm analysis?

Asymptotic analysis and notations

- How to express complexities in simple & uniform ways?
  - While considering the large size of input at the same time
- Concept of asymptotic notations – Big-O notation

Outline

- Motivation to Algorithm Analysis
- How to Measure Efficiency
- Best, Average, and Worst Cases
- Asymptotic Analysis
- Asymptotic Notations

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?
  - A: The fastest and lightest one (i.e., the most efficient).
- Q. How to know which is the most efficient? ⇒ Today's topic!

Time & Space Costs

- 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 or space cost of a solution
  - The amount of time or space that the solution consumes
- Measure efficiency ⇔ Measure time & space costs

Outline

- ❑ Motivation to Algorithm Analysis
- ❑ How to Measure Efficiency
- ❑ Best, Average, and Worst Cases
- ❑ Asymptotic Analysis
- ❑ Asymptotic Notations

How To Measure Efficiency (1)

❑ Empirical Measurement

- e.g., measure the runtime of a program
- e.g., check the maximum memory usage

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
```

- Pros: easy-to-check
- Cons
    - Varied by environment (HW, OS, PL, …) and implementation
    - Hard to know the tendency of performance for the size of input

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 memory space to be used**
    - e.g., the size of an array where the input data are stored

- In general, the 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
    - Mostly, $S(n)$ is linearly proportional to the input size for data structures or algorithms

Basic Operations

- **Run in constant time** regardless of the input size
  - Add/subtraction (+ or −) & division/multiplication (/ or ×)
    - For $a + b$ or $a \times b$, # of operations is a constant (i.e., 1)
  - Assignment (= or ←)
    - For $c = 10$, # of operations is a constant (i.e., 1)
    - For $c \leftarrow a + b$, # of operations is a constant (i.e., 2)
  - Comparison (< or >)
    - For $c > b$, # 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

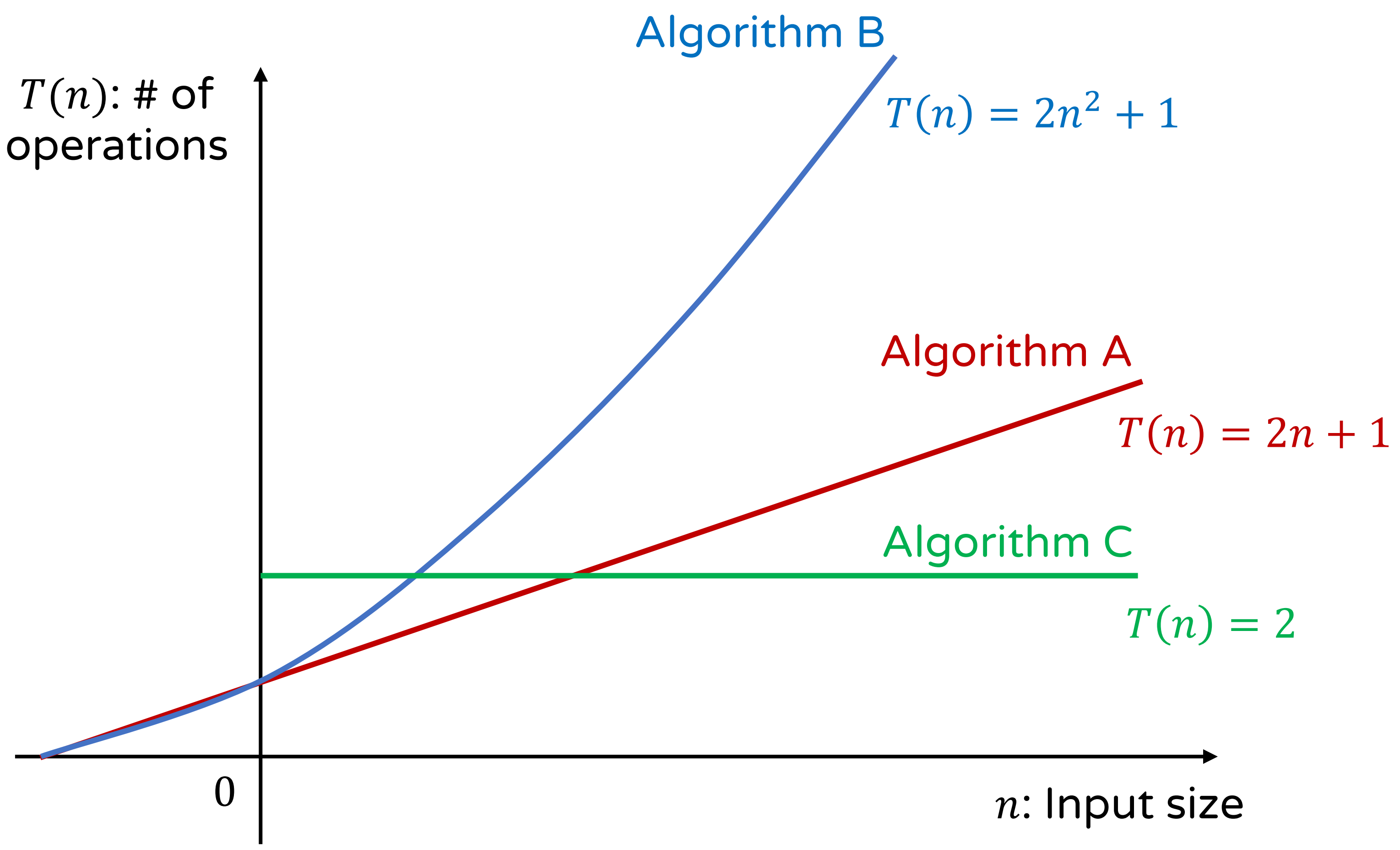
`sum ← 0`
`for i in range(0, n):`
`    sum ← sum + n`
	`sum ← 0`
	`for i in range(0, n):`
	`    for j in range(0, n):`
	`        sum ← sum + 1`
		`sum ← n × n`

  - 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



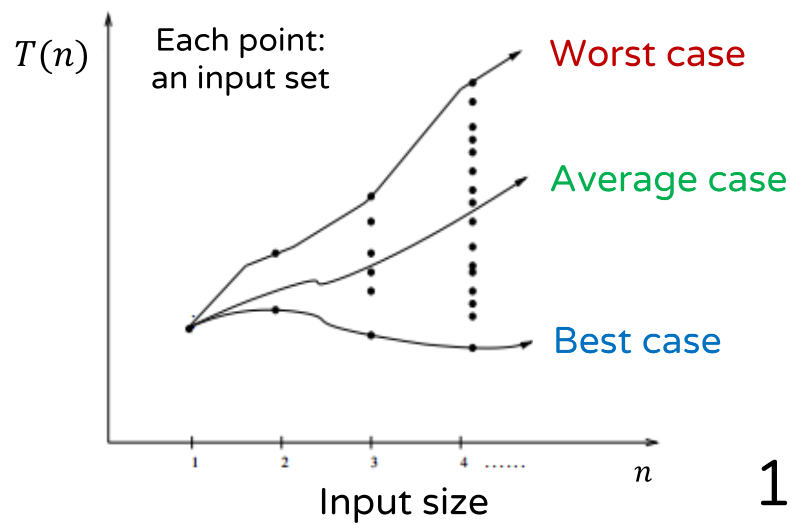
Outline

- Motivation to Algorithm Analysis
- How to Measure Efficiency
- **Best, Average, and Worst Cases**
- Asymptotic Analysis
- Asymptotic Notations

Best, Average, & Worst Cases

- **Complexities can be different according to inputs**
  - Best case: input sets consuming the least resources
    - Easy to imagine, 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 imagine, 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

At least **we should**
do analysis for the worst case



Example Of 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
  - Best case: $T(n) = 1$
    - The array has the querying key at the first
  - Worst case: $T(n) = n$
    - The array has it at the end or no the key
  - Average case: $T(n) = (n+1)/2$
    - The expectation for all possible cases

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

    throw "out-of-key"
```

$$T(n) = \frac{1}{n}\times 1 + \frac{1}{n}\times 2 + \cdots + \underset{\substack{\uparrow \\ P(\text{the key is at index } i)}}{\frac{1}{n}} \times \underset{\substack{\uparrow \\ \text{\# of operations searching for index } i}}{i} + \cdots + \frac{1}{n}\times n = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\times\frac{n(n+1)}{2} = \frac{n+1}{2}$$

Outline

- ❑ Motivation to Algorithm Analysis
- ❑ How to Measure Efficiency
- ❑ Best, Average, and Worst Cases
- ❑ Asymptotic Analysis
- ❑ Asymptotic Notations

Motivation To Asymptotic Analysis

❑ 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., $T_B(n) = n^{10}$

	$n = 10$	$n = 60$	$n = 100$
Algorithm A	$2^{10} = 1024$	$2^{60} \approx 1.15\times 10^{18}$	$2^{100} \approx 10^{30}$
Algorithm B	10^{10}	$60^{10} \approx 6.05\times 10^{17}$	$100^{10} = 10^{20}$
Faster?	Algorithm A	Algorithm B	Algorithm B

- If the input size n becomes extremely large, then Alg. B is faster "eventually" than Alg. A.

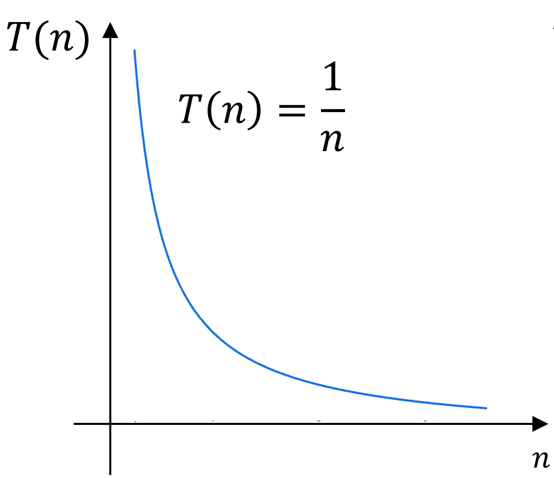
❑ Asymptotic analysis

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

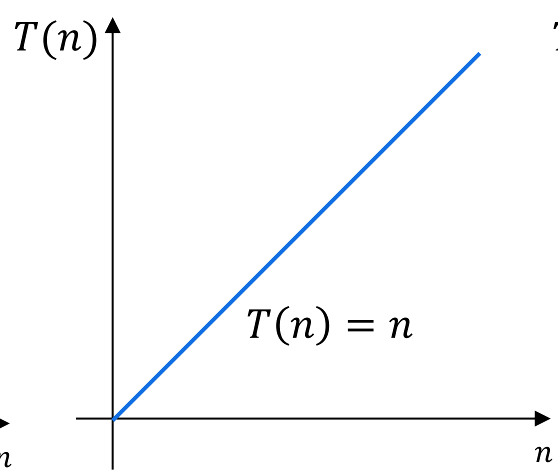
Asymptotic Analysis

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

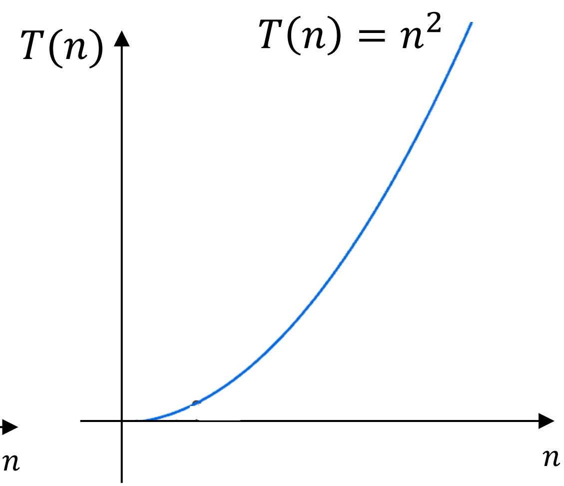
- **Asymptotic**: to approach an infinity point (i.e., $n \to \infty$)
- As $n \to \infty$, how the function changes is called **asymptotic (or limiting/tail) behavior**



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 ∞? (1)

❑ What if the complexity consists of multiple terms?

- 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 called a **dominant factor** having the largest exponent

Why Need To Consider ∞? (2)

- **What if the dominant factor is $5n^2$, $10n^2$, or $100n^2$?**
  - The term n^2 dominates its coefficient as $n \to \infty$.
  - Eventually, they show the similar tail behavior of n^2.

- **How can we simply describe the limiting behavior of an arbitrary complexity function?**
  - $T(n) = n^2$
  - $T(n) = n^2 + n + 1$
  - $T(n) = 3n^2 - 2n + 100$
  - $T(n) = 100n^2$

All of them have the tail behavior of n^2.
Can we represent them in one category?

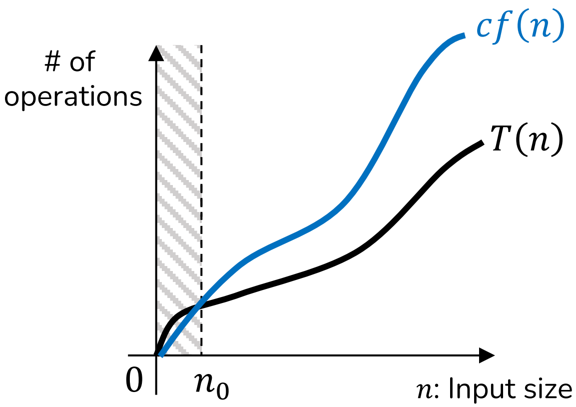
Outline

- ❑ Motivation to Algorithm Analysis
- ❑ How to Measure Efficiency
- ❑ Best, Average, and Worst Cases
- ❑ Asymptotic Analysis
- ❑ Asymptotic Notations

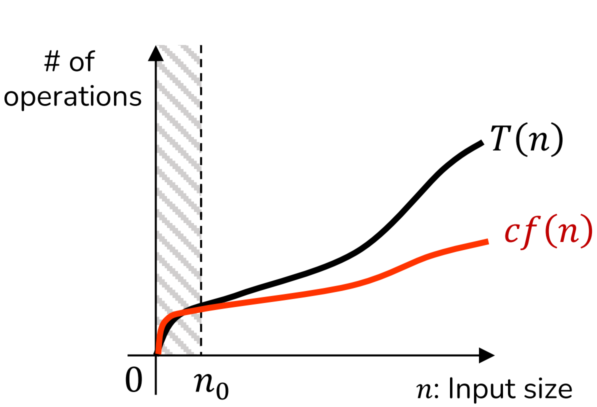
Asymptotic Notations

❑ **Simple way to represent the limiting behaviors of an arbitrary complexity function**

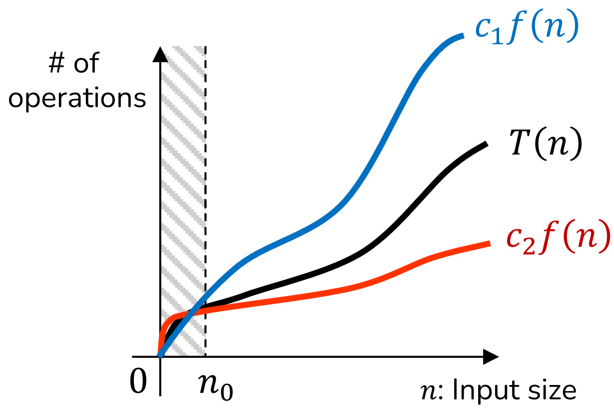
- Big-O notation
- Big-Omega notation
- Big-Theta notation



Big-O



Big-Omega



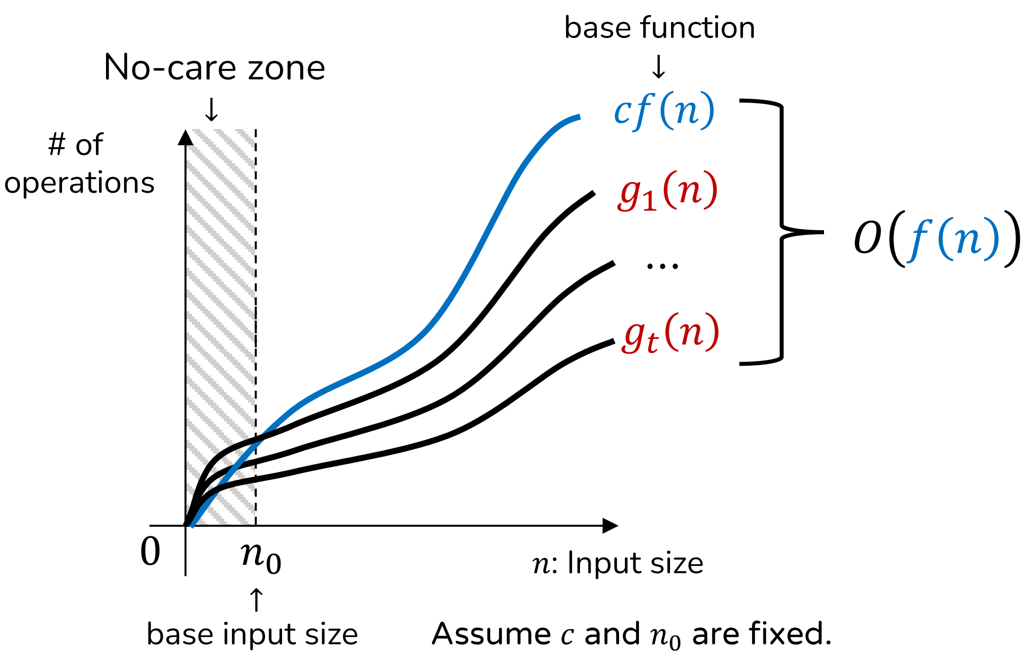
Big-Theta

Big-O Notation (1)

❑ Definition of $O(f(n))$

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

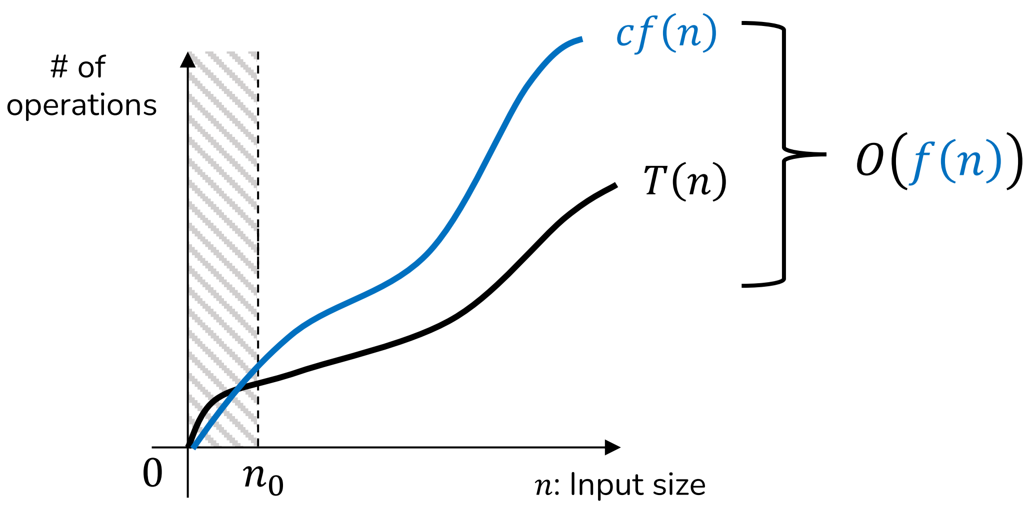
- Set of functions $\le cf(n)$ for large input size n



Big-O Notation (2)

Interpretation of $T(n) = O(f(n))$

- The time complexity $T(n)$ of the algorithm is in $O(f(n))$ for [best | average | worst] case.
  - When the input size is large enough, it always executes in less than or equal to $cf(n)$ for the 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)
    - $3n^2 + 100 \le 3n^2 + 100n^2 = 103n^2 \Rightarrow c = 103$ for all $n \ge n_0 \ge 1$.
      - Any $n_0 \ge 1$ is good in this case (e.g., $n_0 = 1$ or $n_0 = 2$)
  - Proof 2)
    - First, let $c = 13$; then, $3n^2 + 100 \le 13n^2 \Leftrightarrow 100 \le 10n^2 \Leftrightarrow 10 \le n^2$.
    - This indicates $n \ge \sqrt{10} \approx 3.162 \Rightarrow n_0 = 4$.
    - Then, for all $n \ge 4$, $3n^2 + 100 \le 13n^2$.
- If a polynomial has the term of largest degree $\le n^r$, then it is $O(n^r)$.

Big-O Examples (3)

- **Claim)** $T(n) = 5n + 3 = O(n^2)$
  - **Proof)** Suppose $c = 1$; then, $5n + 3 \leq n^2$ for all $n \geq n_0 = 6$.

- **As above, Big-O notation can be either of strict or loose upper bound**
  - By which base function $f(n)$ is targeted,
  - i.e., $T(n) = 5n + 3 = \{O(n), O(n^2), O(n^3), O(n^4), \cdots\}$.
    - If a problem says like "estimate Big-O notation **as tight as possible**", you should write it like $T(n) = O(n)$.
    - Do likewise for Big-Omega notation!

Big-O Examples (4)

- How can we simply describe the limiting behavior of an arbitrary complexity function?
  - $T(n) = n^2$
  - $T(n) = n^2 + n + 1$
  - $T(n) = 3n^2 - 2n + 100$
  - $T(n) = 100n^2$

- The above functions are all in $O(n^2)$!
  - As $n \to \infty$, each function shows the same limiting behavior of n^2, saying **they have the same performance.**

What You Need To Know

❑ Algorithm efficiency

- To compare the performance of algorithms
- Measured by time or space costs empirically & theoretically

❑ Best, average, and worst cases

- Do analysis for worst case to guarantee the performance

❑ Asymptotic analysis and notations

- Aim to analyze the efficiency of an algorithm when $n \to \infty$
- $O\big(f(n)\big)$ = a set of functions $\leq\ cf(n)$ for large input size n
  - $T(n) = O\big(f(n)\big)$ means $T(n)$ grows asymptotically no faster than $f(n)$ as upper bound.

In Next Lecture

- **Other asymptotic notations**
  - Big-Omega and Big-Theta notations
- **Simplifying rules on the notations**
- **More examples of asymptotic analysis**
- **How to analyze algorithms with multiple parameters**

Thank You