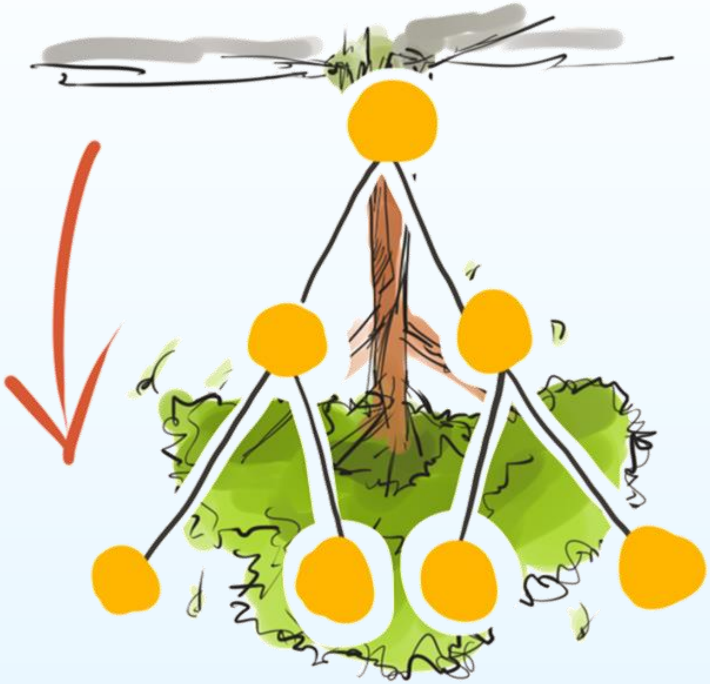


# Lecture 4:
# Basic Complexity Analysis

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.

# Outline

- Mathematic Background
  - Logarithms and Exponents
  - Series
  - Recurrence Relations
- Complexity Analysis
  - Asymptotic Notations
  - Recurrence Relations

# Efficiency of Algorithms

- In many situations, you will often have a selection of among possible algorithms or data structures, or even compare them

- It is not possible to simply say that

  "algorithm $A$ is faster than algorithm $B$" => quality

  Why?
  - System dependence: execution time, memory space, compiler, …
  - Application dependence: input data

- An alternative comparison is based on the quantity metric by looking at relatively rates of growth in time requirements as the size of problem increases
  - Use mathematics and elementary calculus

# L'Hôpital โลปิตาล

If you are attempting to determine

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

but both $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$, it follows

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f^{(1)}(n)}{g^{(1)}(n)}$$

Repeat as necessary …

Where $f^{(k)}(n)$ is the $k^{th}$ derivative

# Logarithms and Exponents

If $n = e^m$, we define $m = \ln(n)$

พหุนาม

Exponents grow faster than any non-constant polynomial

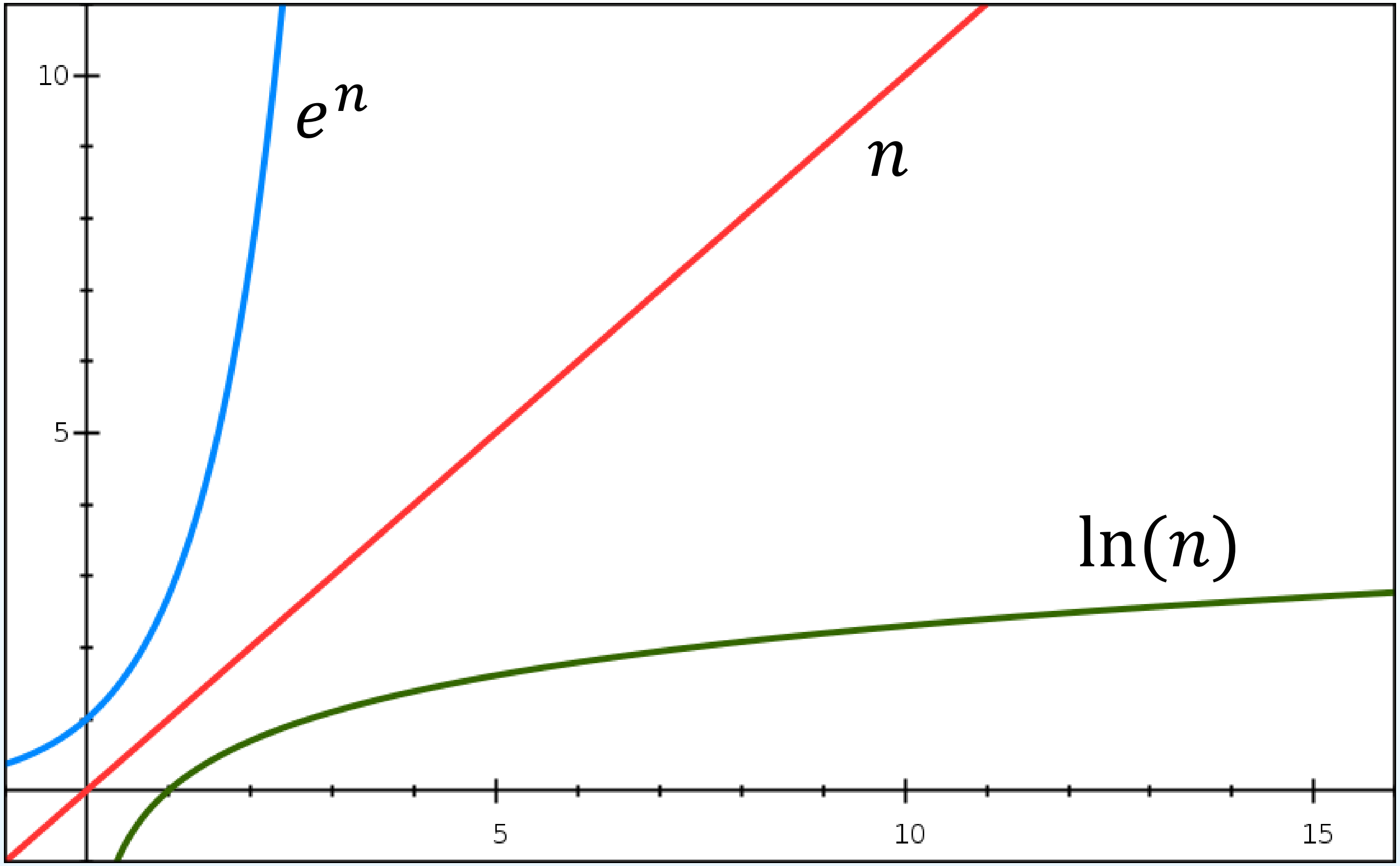
$$\lim_{n\to\infty} \frac{e^n}{n^d} = \infty \text{ for any } d > 0$$

Thus, their inverses (i.e., logarithms) grow slower than any polynomial

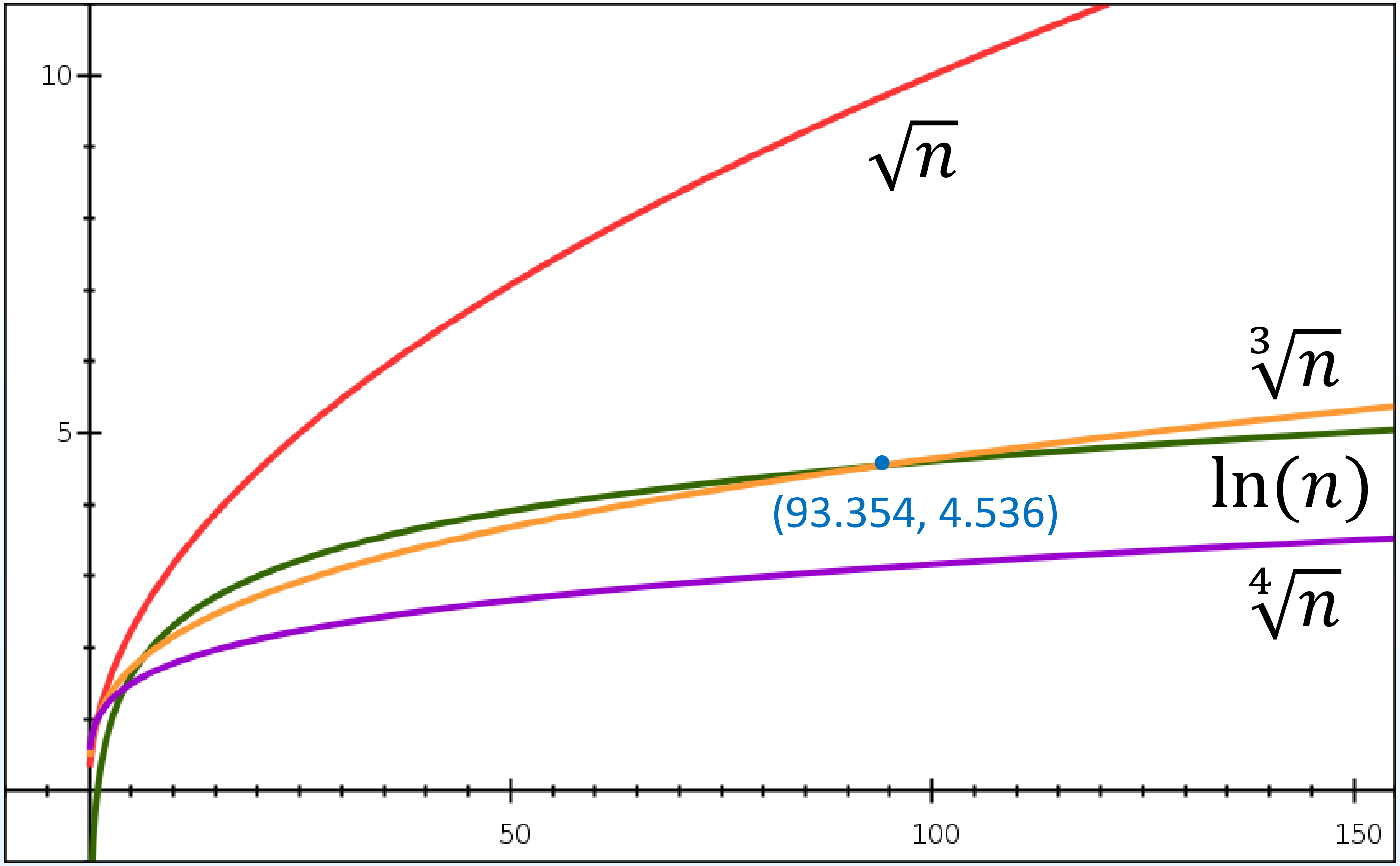
$$\lim_{n\to\infty} \frac{\ln(n)}{n^d} = 0$$

$n^x, x > 0$

# Example: Logarithms and Exponents



# Example: Logarithms and Exponents




after the point (5503.66, 8.61), $\ln(n)$ will grow slower than $\sqrt[4]{n}$

# Logarithm with Different Bases

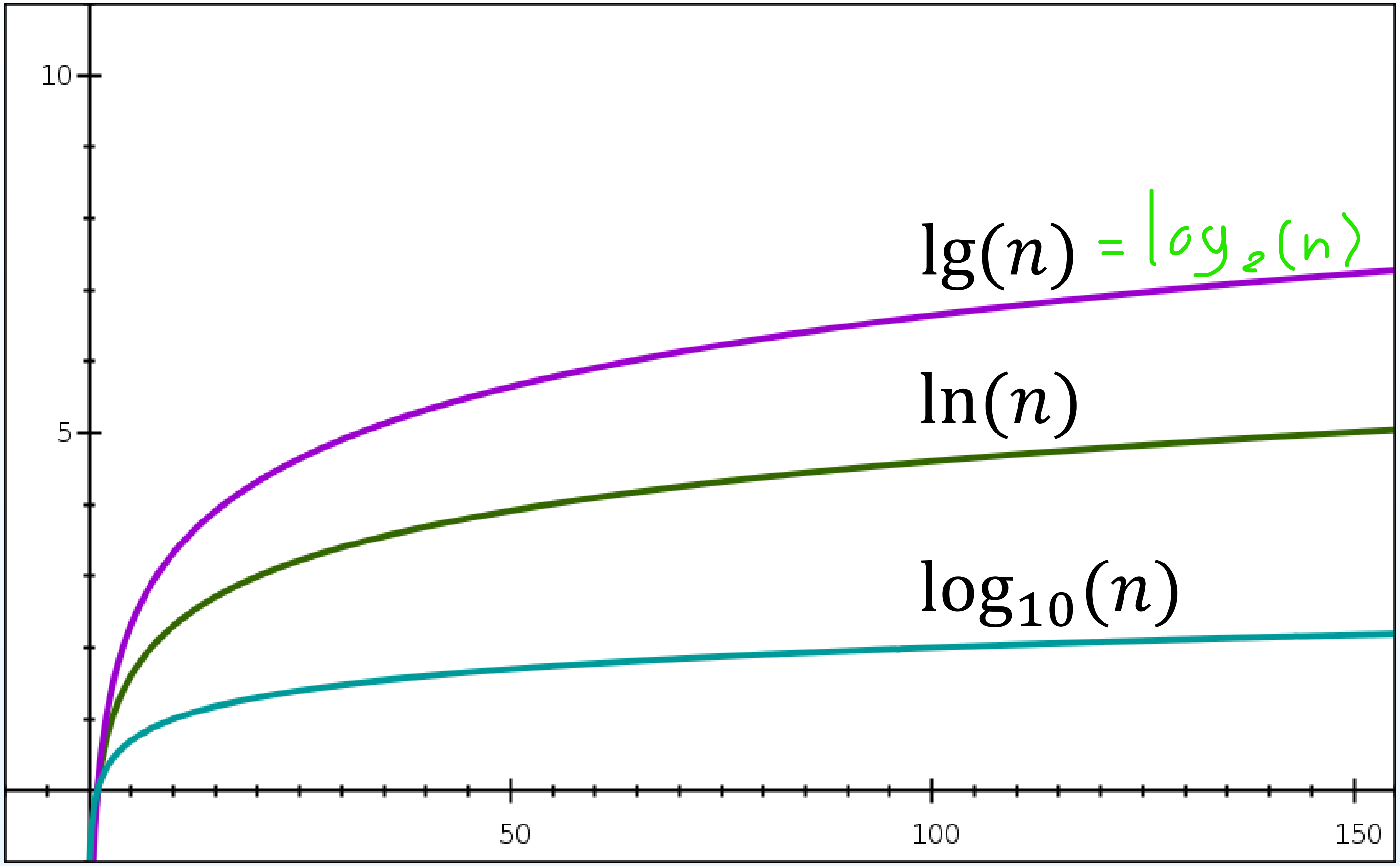
You have seen the formula

$$\log_b(n) = \frac{\log_x(n)}{\log_x(b)} = \frac{\ln(n)}{\ln(b)}$$

constant

So that all logarithms are scalar multiples of each others

# Logarithm with Different Bases




Note: the base-2 logarithm $\log_2(n)$ is written as $\lg(n)$

# Some Properties of Logarithms

- $\log_b(nm) = \log_b(n) + \log_b(m)$
- $\log_b\left(\frac{n}{m}\right) = \log_b(n) - \log_b(m)$
- $\log_b(n^m) = m\log_b(n)$
- $b^{\log_b(n)} = n$
- $n^{\log_b(m)} = m^{\log_b(n)}$
- $\log_b \log_b(n) < \log_b(n) < n$ for all $n > 0$

# Arithmetic Series

Each term in an arithmetic series is increased by a constant value (usually 1):

$$1 + 2 + 3 + \cdots + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Proof1: Adding the series twice

$$\begin{aligned}
S_n &= 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n \\
S_n &= n + (n-1) + (n-2) + \cdots + 3 + 2 + 1 \\
2S_n &= (n+1) + (n+1) + (n+1) + \cdots + (n+1) + (n+1) + (n+1) \\
S_n &= \tfrac{1}{2} n(n+1)
\end{aligned}$$

Proof2: By induction

- Basic step: The statement is true for $n = 1$
- Inductive hypothesis: Assume the statement is true for $1 < i \leq n$
- Inductive step: Based on the hypothesis, the statement is also true for $n + 1$

# Quickly Algorithm Analysis

Consider the following code fragment:

```
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
    printf("Hello\n");
```

How many times is `printf()` executed?

| i | j | times |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 1,2 | 2 |
| 3 | 1,2,3 | 3 |
| … | | |
| n | 1,2,3,…,n | n |

arithmetic series

$$= \frac{n(n+1)}{2}$$

# Other Polynomial Series

We could repeat the proven process, after all:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

# Geometric Series

A series for which the ratio of each two consecutive terms ${}^{a_{i+1}}/_{a_i}$ is a constant $|r| < 1$

$$1 + r + r^2 + \cdots + r^n = \sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

Proof:

$$\begin{aligned}
S_n &= 1 + \ r \ + r^2 + \cdots + r^n \\
rS_n &= r + r^2 + r^3 + \cdots + r^{n+1} \\
S_n - rS_n &= (1 + \ r \ + r^2 + \cdots + r^n) - (r + r^2 + r^3 + \cdots + r^{n+1}) \\
(1 - r)S_n &= 1 - r^{n+1} \\
S_n &= \frac{1 - r^{n+1}}{1 - r}
\end{aligned}$$

# Geometric Series

A series for which the ratio of each two consecutive terms ${}^{a_{i+1}}/_{a_i}$ is a constant $|r| < 1$

$$1 + r + r^2 + \cdots + r^n = \sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

The sum converges as $n \to \infty$

$$1 + r + r^2 + \cdots = \sum_{i=0}^{\infty} r^i = \frac{1}{1 - r}$$

# Recurrence Relations

- Sequences may be defined $\boxed{\text{explicitly}}$ ตรงๆ
  - For example, the harmonic sequence $x_n = \frac{1}{n}$, we have $1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \cdots$

- A **recurrence relationship** is a means of defining a sequence based on previous values in the sequence
  - Such definitions of sequences are said to be $\boxed{\text{recursive}}$
  - For example,

    the odd number sequence: $x_n = x_{n-1} + 2$ where $x_1 = 1$

    the Fibonacci sequence: $x_n = x_{n-1} + x_{n-2}$ where $x_0 = 0, x_1 = 1$

# Recurrence Relations

- In some cases, given the recurrence relation, we can find the explicit formula (closed form)
  - For example,

    the odd number sequence: $x_n = x_{n-1} + 2$ where $x_1 = 1$

    its closed form is given by $x_n = 2n - 1$

    the Fibonacci sequence: $x_n = x_{n-1} + x_{n-2}$ where $x_0 = 0, x_1 = 1$

    its closed form is given by $x_n = \frac{\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n}{2^n\sqrt{5}}$

# Recurrence Relations

- We may use a functional form for a recurrence relation:

| **Mathematic** | **Function** |
|---|---|
| $x_1 = 1$ | $f(1) = 1$ |
| $x_n = x_{n-1} + 2$ | $f(n) = f(n-1) + 2$ |
| $x_n = x_{n-1} + x_{n-2}$ | $f(n) = f(n-1) + f(n-2)$ |

# Weighted Averages

Given $n$ objects $x_1, x_2, x_3, \ldots, x_n$, the average is

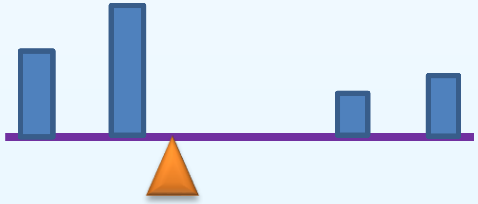
$$\frac{x_1 + x_2 + x_3 + \cdots + x_n}{n}$$

If we are given a sequence of coefficients $c_1, c_2, c_3, \ldots, c_n$ where

$$c_1 + c_2 + c_3 + \cdots + c_n = 1$$

then we refer to

$$c_1 x_1 + c_2 x_2 + c_3 x_3 + \cdots + c_n x_n$$



as a weighted average

For an average, $c_1 = c_2 = c_3 = \cdots = c_n = \frac{1}{n}$

# Combinations

Given $n$ distinct items, in how many ways can you choose $k$ of these?

The number of ways such items can be chosen is written

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

where $\binom{n}{k}$ is read as "$n$ choose $k$"

# Combinations

You have also seen this in expanding polynomials:

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

For example,

$$(x+y)^4 = \sum_{k=0}^{4} \binom{4}{k} x^k y^{4-k}$$

The coefficients of Pascal's triangle:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

$$= \binom{4}{0} y^4 + \binom{4}{1} xy^3 + \binom{4}{2} x^2y^2 + \binom{4}{3} x^3y + \binom{4}{4} x^4$$

$$= y^4 + 4xy^3 + 6x^2y^2 + 4x^3y + x^4$$

# Outline

- Mathematic Background
  - Logarithms and Exponents
  - Series
  - Recurrence Relations
- Complexity Analysis
  - Asymptotic Notations
  - Recurrence Relations

# Algorithm Analysis

In an algorithm analysis, you always have to know as the size of an algorithm's input grows

- Time: How much longer does it run?
- Space: How much memory does it use?

**How do you answer these questions?**

For now, we will focus on time only.

# Problems with Timing

- Why not just code the algorithm and time it?
  - Hardware: processor, memory, etc.
  - OS, programming language, libraries, compiler/interpreter
  - Programs running in the background
  - Choice of input, number of inputs
- Timing does not really evaluate the algorithm but merely evaluates a specific implementation
- At the core of CS, a backbone of theory & mathematics
  - Examine the algorithm itself, not the implementation
  - Reason about performance as a function of $n$
  - Mathematically proven things about performance
- Yet, timing has its place
  - In real world, we do want to know whether implementation $A$ runs faster than implementation $B$ on data set $C$, e.g., Benchmarking

# Evaluations

Evaluate an algorithm

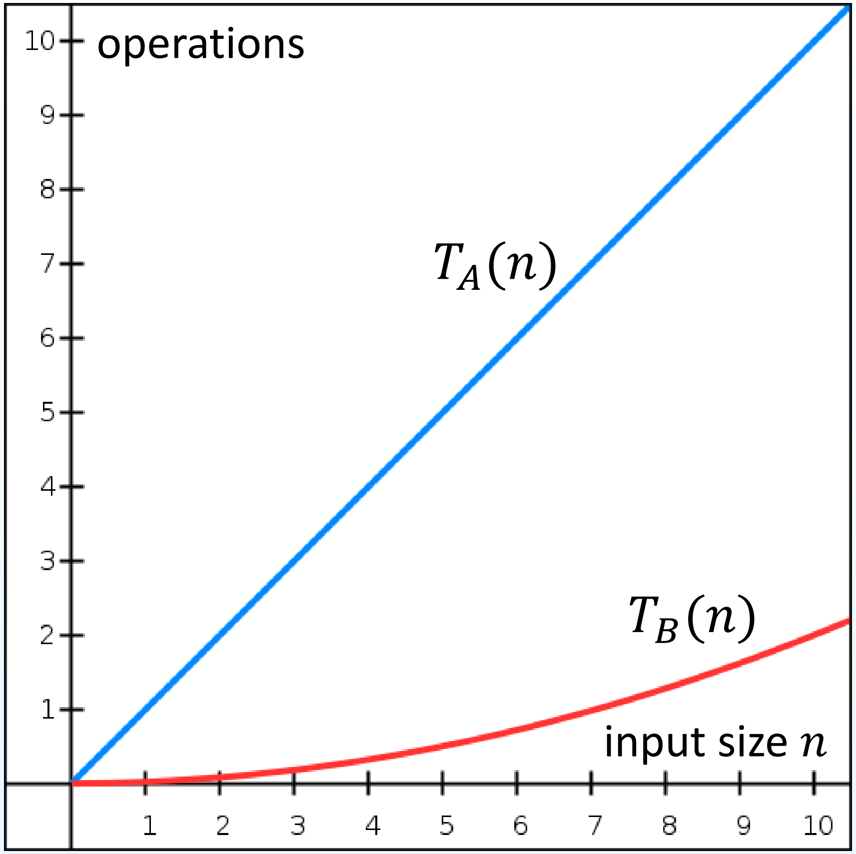
➡ Use asymptotic analysis

Evaluate an implementation

➡ Use timing

# Motivation for Algorithm Analysis

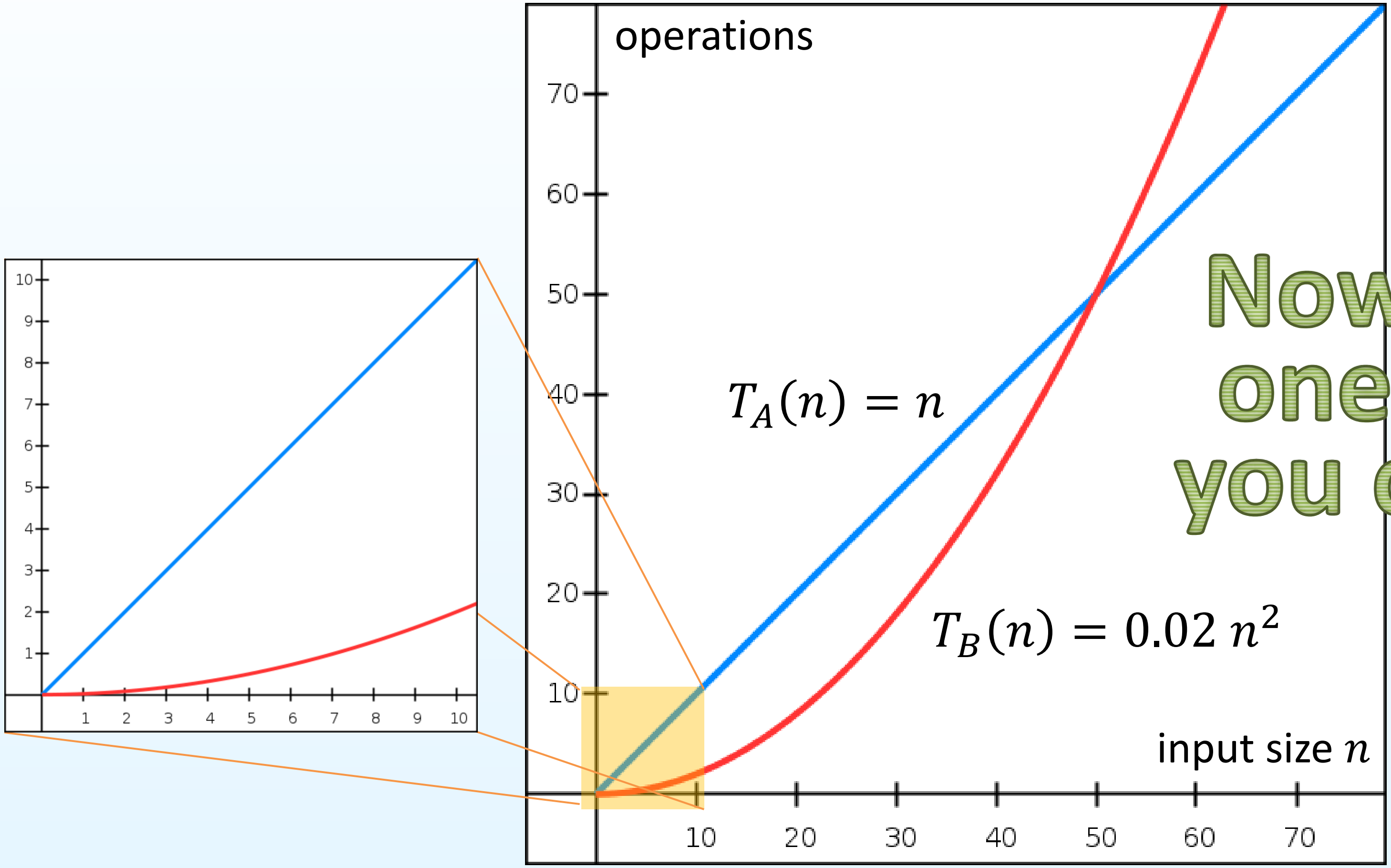
- Suppose you are given two algorithms $A$ and $B$ for solving the problem
- The running times (operations) $T_A(n)$ and $T_B(n)$ of $A$ and $B$ as a function of input size $n$ are given




**Which is better?**

# Motivation for Algorithm Analysis

- For large $n$, the running times of $A$ and $B$ are:




Now which one would you choose?

# Goals of Algorithm Analysis

- Concentrate on large inputs
  - Some algorithms only work fine for small inputs
- Be independent of hardware, OS, language, etc.
- Be general, not specific in some test cases

# Assumptions in Analyzing Code

- Basic operations take a unit (constant) running time, e.g.,
  - Arithmetic + - × / กำหนดค่า etc
  - Assignment
  - Comparing two simple values
  - Accessing array with an index

- Other operations are summations or products
  - Consecutive statements are summed
  - Loops are (cost of loop body)×(number of loops)

# Examples: Analyzing Code

## What are the running times for the following codes?

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

Annotations: `i=0` → $1$; `i<n` → $n$; `i++` → $n$; `x` → $n$; `x+1` → $n$

$$\approx 1 + 4n$$

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

Annotations: `i=0; i<n; i++` → $1 + 2n$; inner loop → $n(1 + 4n)$

$$\approx 1 + 2n + n(1 + 4n)$$
$$\approx 1 + 3n + 4n^2$$

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

Annotations: `i=0; i<n; i++` → $1 + 2n$; inner loop → $n + 4(1 + 2 + \cdots + n)$

$$\approx 1 + 2n + n + \frac{4n(n+1)}{2}$$
$$\approx 1 + 5n + 2n^2$$

# No Need to be so Exact

- Constant coefficients do not matter

  For example: Given $T_A(n) = n^2$ and $T_B(n) = 10n^2$,
  which has the faster growth rate? โตเท่ากัน

$$\lim_{n\to\infty} \frac{n^2}{10n^2} = \lim_{n\to\infty} \frac{1}{10} = 0.1$$

a constant

- Lower-order terms are less important

  For example: Given $T_A(n) = n^2$ and $T_B(n) = 10n^2 + 5n + 2$,
  which has the faster growth rate?

$$\lim_{n\to\infty} \frac{n^2}{10n^2 + 5n + 2} = \lim_{n\to\infty} \frac{2n}{20n + 5} = \lim_{n\to\infty} \frac{1}{10} = 0.1$$

**"We will focus on the dominant term only"**

# Worst-Case Analysis

- In general, we are interested in three types of performance
  - Best-case
  - Average-case *ทำยาก
  - Worst-case
- When determining worst-case, we tend to be pessimistic
  - If there is a conditional, count the branch that runs the slowest
  - This will give a loose bound on how slow the algorithm may run

# Algorithmic Complexity

How the running time of an algorithm increases with the size of the input *in the limit* (**growth rate**), as the size of the input increase without bound

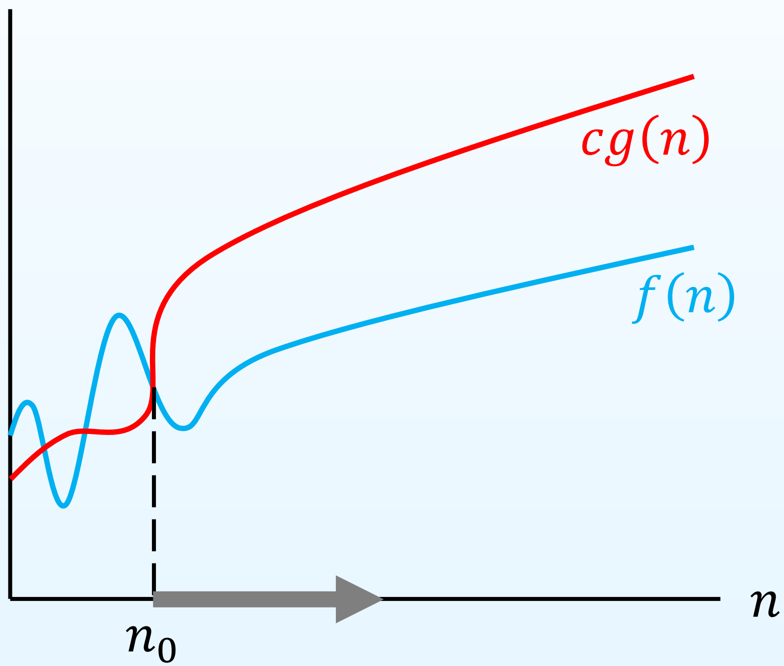
# Asymptotic Notation

Notations are used to describe the asymptotic running time (**complexity**) of an algorithm, defined as **functions** whose domains are the set of natural numbers $N = \{0, 1, 2, \dots\}$

# Asymptotic: Big-Oh Notation

Given two functions $f(n)$ and $g(n)$ for inputs $n$, we say

"$f(n)$ is in $O(g(n))$ iff there exist positive constants $c$ and $n_0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$"




**Proof:**

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0, c \;\; ; c \neq \infty$$

$g(n)$ is an asymptotic upper bound for $f(n)$

# Examples: Big-Oh

Are the following statements TRUE or FALSE?

- $4 + 3n$ is in $O(n)$

$$\lim_{n\to\infty} \frac{4+3n}{n} = 3$$

TRUE

- $n + 2\ln(n)$ is in $O(\ln(n))$

$$\lim_{n\to\infty} \frac{n+2\ln(n)}{\ln(n)} = \lim_{n\to\infty} \frac{1+\frac{2}{n}}{\frac{1}{n}} = \lim_{n\to\infty} (n+2) = \infty$$

FALSE

- $n^{50}$ is in $O(2^n)$

$$\lim_{n\to\infty} \frac{n^{50}}{2^n} = \lim_{n\to\infty} \frac{50n^{49}}{\ln(2)\,2^n} = \lim_{n\to\infty} \frac{50\cdot 49n^{48}}{\ln^2(2)\,2^n}$$

$$= \cdots = \lim_{n\to\infty} \frac{50!}{\ln^{50}(2)\,2^n} = 0$$

TRUE

# Big-Oh Common Comparisons

Increasing running time

| Big-Oh | Description |
|---|---|
| $O(1)$ | Constant (or $O(k)$ for constant $k$) |
| $O(\log\log n)$ | Log log |
| $O(\log n)$ | Logarithmic |
| $O(\log^2 n)$ | Log squared |
| $O(n)$ | Linear |
| $O(n\log n)$ | n log n |
| $O(n^2)$ | Quadratic |
| $O(n^3)$ | Cubic |
| $O(n^k)$ | Polynomial (where $k$ is constant) |
| $O(k^n)$ | Exponential (where constant $k > 1$) |

# Comment on Notation

- $4 + 3n$ is in $O(n)$? ✓
- $4 + 3n$ is in $O(n \log n)$?
- $4 + 3n$ is in $O(n^2)$?
- $4 + 3n$ is in $O(n^3)$?
- $4 + 3n$ is in $O(n^k)$, for all $k \geq 1$?
- $4 + 3n$ is in $O(k^n)$, for all $k > 1$?

Choose $O(\cdot)$ with the least running time as possible!

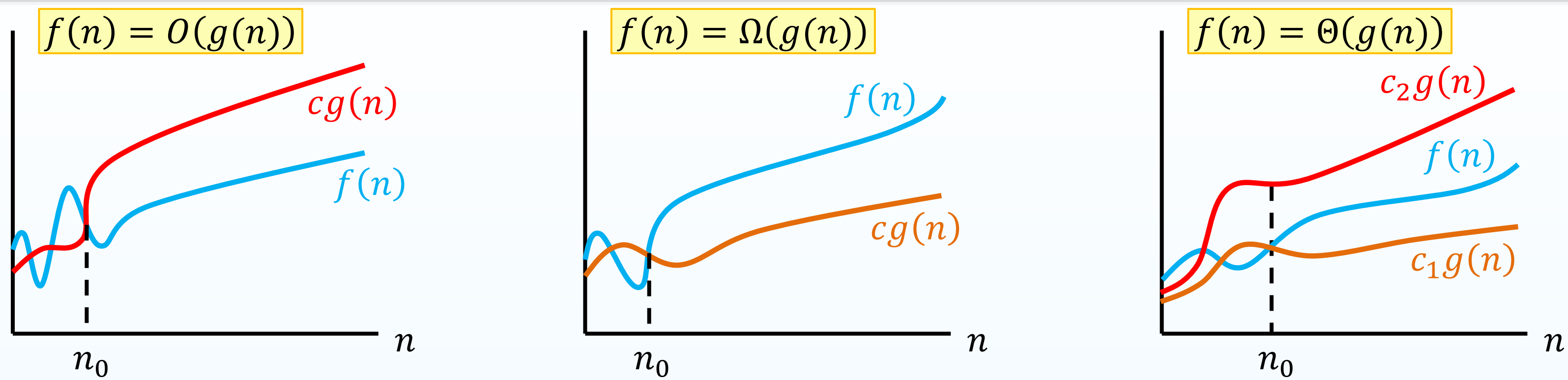
# Comment on Notation

- We say "$3n^2 + 17$ is in $O(n^2)$"
- We may also say/write it as

$$3n^2 + 17 \text{ is } O(n^2)$$
$$3n^2 + 17 = O(n^2)$$
$$3n^2 + 17 \in O(n^2)$$

- But '=' does not mean an equality, so that we would never say $O(n^2) = 3n^2 + 17$

# Asymptotic Notations




- **Big-Oh: upper bound**

  $f(n)$ is in $O(g(n))$ iff there exist positive constants $c$ and $n_0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$

- **Big-Omega: lower bound**

  $f(n)$ is in $\Omega(g(n))$ iff there exist positive constants $c$ and $n_0$ such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$

- **Big-Theta: tight bound** อ่านว่า เท่า

  $f(n)$ is in $\Theta(g(n))$ iff there exist positive constants $c_1$, $c_2$, and $n_0$ such that $0 \le c_1g(n) \le f(n) \le c_2g(n)$ for all $n \ge n_0$

# Asymptotic Notations

Less common notations

- Little-oh: like Big-Oh but strictly less than

  $f(n)$ is in $o(g(n))$ iff for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$

  For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$

- Little-omega: like Big-Omega but strictly greater than

  $f(n)$ is in $\omega(g(n))$ iff for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$

  For example, $2n^2 = \omega(n)$, but $2n^2 \neq \omega(n^2)$

# Asymptotic Notations

$$f(n) = o(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = O(g(n)) \qquad 0 \le \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \Theta(g(n)) \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \Omega(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

$$f(n) = \omega(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

# Example: Big-Oh Analysis

Compute the sum of `n` integers stored in the array `a`:

**Code Fragment:**

```
1: int sum_array(int a[], int n) {
2:   int sum = 0, i;
3:
4:   for (i=0; i<n; i++)
5:     sum += a[i];
6:   return sum;
7: }
```

- Lines 2 and 6 take constant time, i.e., $O(1)$
- Lines 4 and 5 perform `n` iterations, i.e., $O(n)$
- So that the running time is $O(1+n) \Rightarrow O(n)$
- Actually, $\Theta(n)$ since all `n` integers are exactly accessed

# Example: Big-Oh Analysis

Find the value v in the array a of n integers

**Code Fragment:**

```
1: int find(int a[], int n, int v) {
2:   int i;
3:
4:   for (i=0; i<n; i++)
5:     if (a[i] == v)
6:       return 1;
7:   return -1;
8: }
```

Lines 4-6 are the dominant costs of the running time

- Worst-case: v is the last element ➡ $O(n)$
- Best-case: if you are lucky, v is the first element ➡ $\Omega(1)$
- Average-case: the probability of v stored in each position ➡ …

# Example: Big-Oh Analysis

Again, compute the sum of `n` integers stored in the array `a`:

**Code Fragment:**

```
1: int sum_array(int a[], int n) {
2:   if (n == 1)
3:     return a[n-1];
4:   else
5:     return a[n-1] + sum_array(a, n-1);
6: }
```

- Lines 2-3 take constant $O(1)$
- Let $T(n)$ be the running time of summing all `n` integers
- Then, in lines 4-5, the cost is $O(1) + T(n-1)$ +
- So that we got the recurrence relation:

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ T(n-1) + O(1) & \text{otherwise} \end{cases}$$

- How can we find $O(T(n))$? $O(n)$

# Recurrence Relations

- Substitution method
  - Expand the recurrence and express it as a summation of terms dependent only on $n$ and the initial conditions
- Recursion tree
  - Visualize what happens when a recurrence is iterated
- Master theorem
  - Provide a cookbook method for solving recurrences of the specific form

# Example: Sum of n Elements in Array

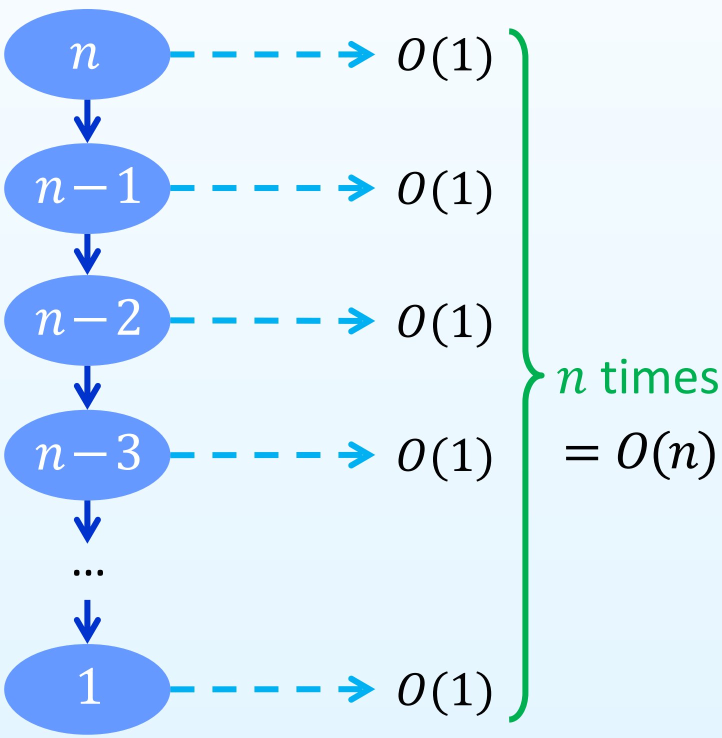
$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ T(n-1) + O(1) & \text{otherwise} \end{cases}$$

branching workload

- Substitution method

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ &= T(n-2) + O(1) + O(1) \\ &= T(n-3) + O(1) + O(1) + O(1) \\ &\quad \cdots \\ &= \boxed{T(1)} + O(1) + \cdots + O(1) \\ &= \underbrace{O(1) + O(1) + \cdots + O(1)}_{n \text{ times}} \\ &= O(n) \end{aligned}$$

- Recursion tree



# Example: Towers of Hanoi

```
#include <stdio.h>

void toh(int n, char from, char to, char aug) {
  if (n == 1)
    printf("Move %d from %c to %c\n", n, from, to);   → base case
  else {
    toh(n-1, from, aug, to);                          → branching
    printf("Move %d from %c to %c\n", n, from, to);   → workload
    toh(n-1, aug, to, from);                          → branching
  }
}

int main(void) {
  int n = 0;                  → O(1)

  printf("Enter n: ");        → O(1)
  scanf("%d", &n);            → O(1)
  toh(n, 'A', 'B', 'C');      → O(T(n))
  return 0;                   → O(1)
}
```

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ 2T(n-1) + O(1) & \text{otherwise} \end{cases}$$

# Recurrence Relation: Tower of Hanoi

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ 2T(n-1) + O(1) & \text{otherwise} \end{cases}$$

- Substitution method

(n-1) levels

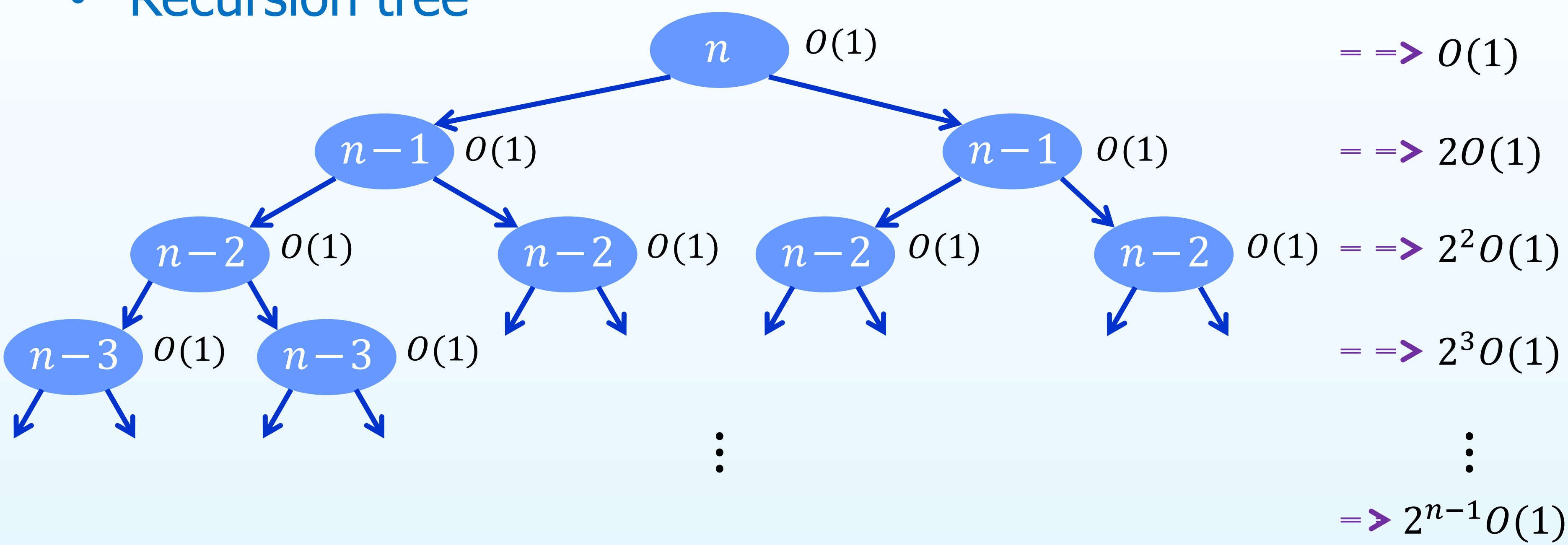
$$\begin{aligned}
T(n) &= 2T(n-1) + O(1) \\
&= 2\big(2T(n-2) + O(1)\big) + O(1) = 2^2T(n-2) + 2O(1) + O(1) \\
&= 2^2\big(2T(n-3) + O(1)\big) + 2O(1) + O(1) = 2^3T(n-3) + 2^2O(1) + 2O(1) + O(1) \\
&= 2^4T(n-4) + 2^3O(1) + 2^2O(1) + 2O(1) + O(1) \\
&\quad \dots \\
&= 2^{n-1}T(1) + 2^{n-2}O(1) + \dots + 2^2O(1) + 2O(1) + O(1) \\
&= 2^{n-1}O(1) + 2^{n-2}O(1) + \dots + 2^2O(1) + 2^1O(1) + 2^0O(1) \\
&= O(1)\sum_{i=0}^{n-1} 2^i \\
&= (2^n - 1)O(1) = O(2^n - 1) = O(2^n)
\end{aligned}$$

| | n | n-1 | … | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 1 | 1 | 1 | 1 | 1 | |
| = | 1 | 0 | 0 | 0 | 0 | 0 | 0 | - 1 |

# Recurrence Relation: Tower of Hanoi

$$T(n) = \begin{cases} O(1) & \text{if } n = 1, \\ 2T(n-1) + O(1) & \text{otherwise} \end{cases}$$

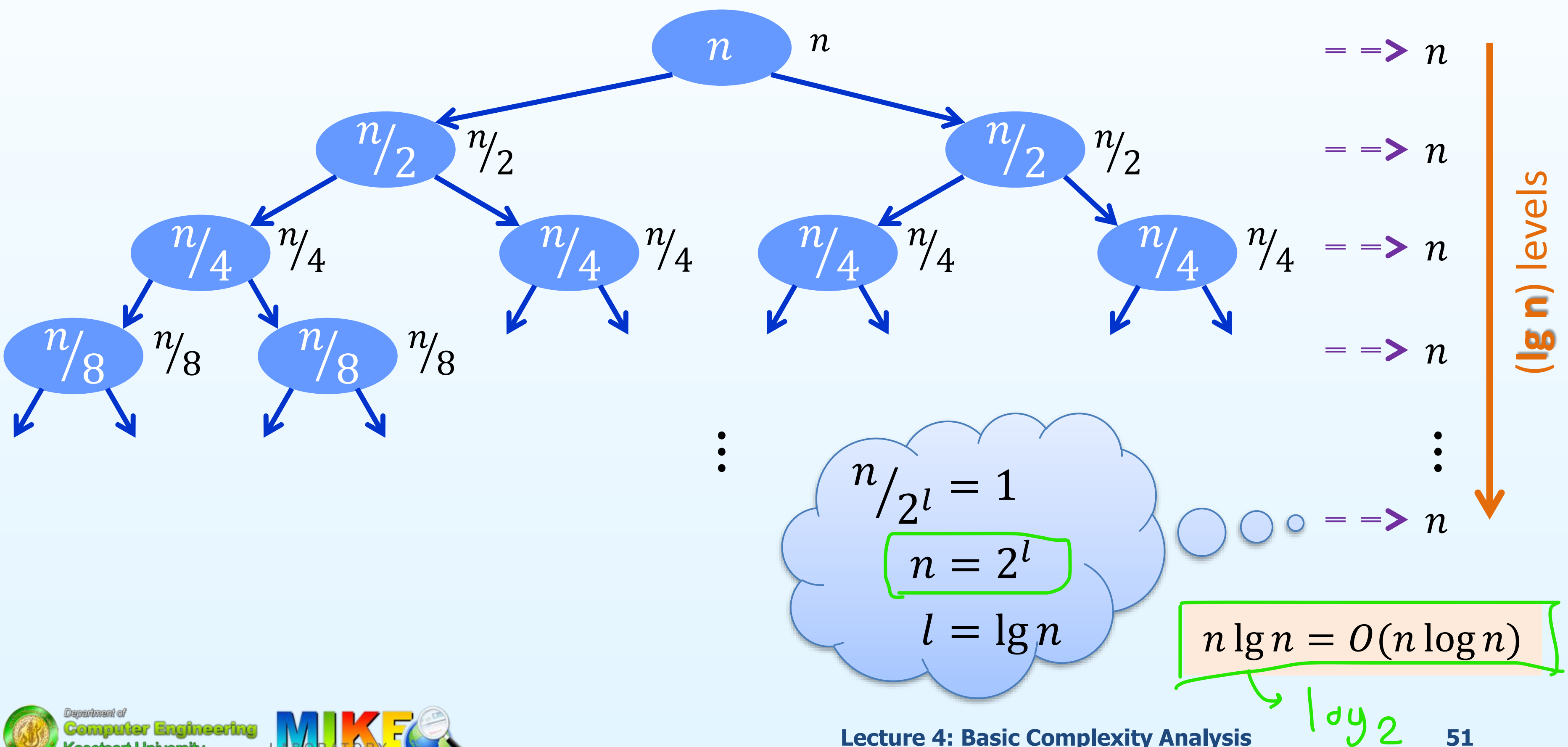
- Recursion tree




$$O(1)\sum_{i=0}^{n-1} 2^i = O(2^n)$$

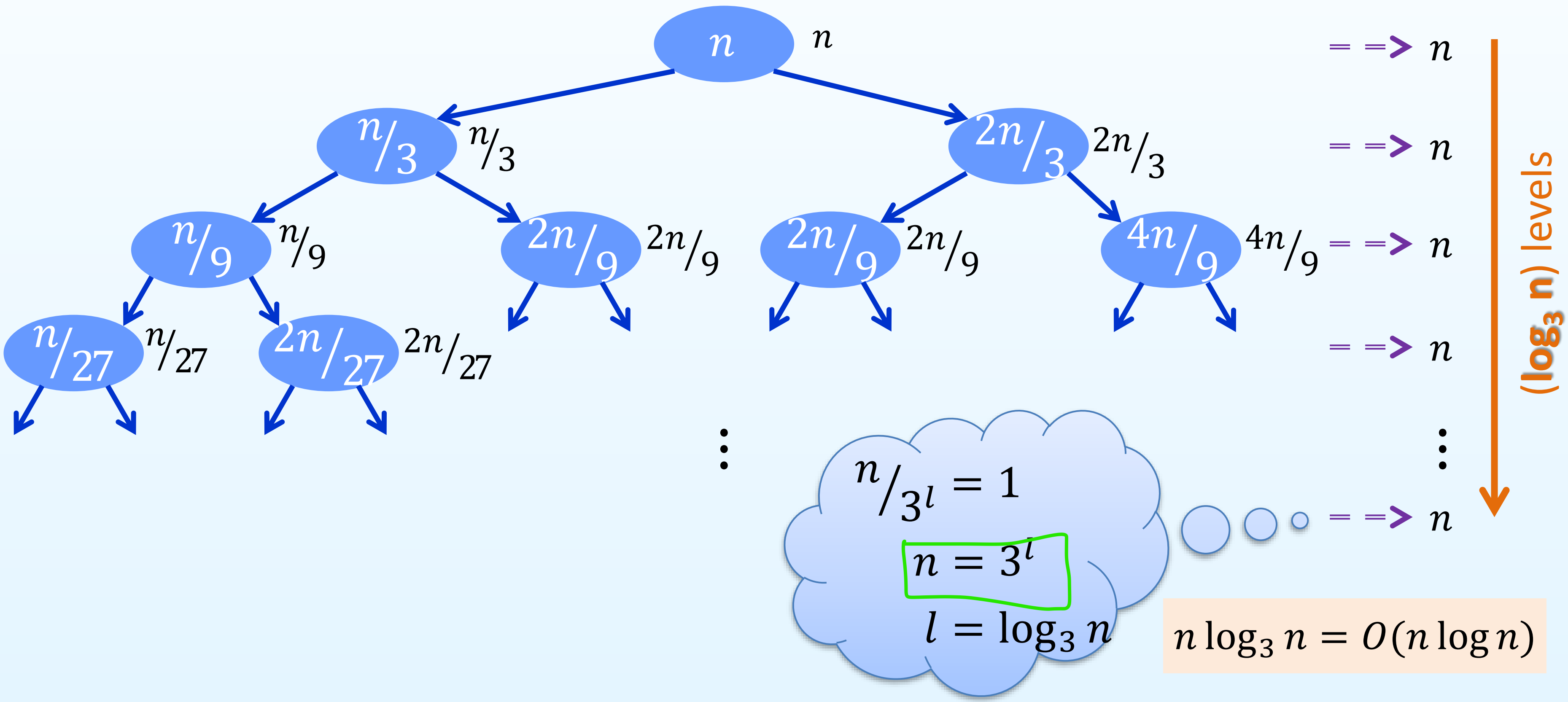
# Example: Recursion Tree

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$



# Example: Recursion Tree

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$



# Recurrence Relations

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \qquad ; a \geq 1, b > 1$$

- Master theorem
  - If $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$ for some constant $\varepsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$
  - If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$
  - If $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$ for some constant $\varepsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$

# Example: Master Theorem

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

- We have $a = 9,\ b = 3,\ f(n) = n$
- Thus $n^{\log_b a} = n^{\log_3 9} = n^2$
- Since $f(n) = n = O\left(n^{\log_3 9 - \varepsilon}\right)$, where $0 < \varepsilon \leq 1$, we can apply the case 1
- So that $T(n) = \Theta\left(n^{\log_3 9}\right) = \Theta(n^2)$

# Example: Master Theorem

$$T(n) = T\left(\frac{n}{3}\right) + 1$$

- We have $a = 1,\ b = 3,\ f(n) = 1$
- Thus $n^{\log_b a} = n^{\log_3 1} = 1$
- Since $f(n) = 1 = \Theta\left(n^{\log_3 1}\right)$, we can apply the case 2
- So that $T(n) = \Theta(n^{\log_3 1} \log n) = \Theta(\log n)$

# Example: Master Theorem

$$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$$

- We have $a = 3,\ b = 4,\ f(n) = n\log n$
- Thus $n^{\log_b a} = n^{\log_4 3} \approx n^{0.793}$
- Since $f(n) = n\log n = \Omega\left(n^{\log_3 4+\varepsilon}\right)$, where $0 < \varepsilon \le 0.207$, the case 3 will apply if we can show $af\left(\frac{n}{b}\right) \le cf(n), c < 1$
- $3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) = \left(\frac{3}{4}\right)n(\log n - \log 4) \le \left(\frac{3}{4}\right)n\log n$ for $c = \frac{3}{4}$
- So that $T(n) = \Theta(f(n)) = \Theta(n\log n)$

# Any Question?