CMP418: Algorithm and Complexity Analysis (3 units)

Lecture 2: Fundamentals of the Analysis of Algorithm Efficiency

MR. M. YUSUF

Chapter 2 Outline

- The Efficiency Analysis Framework
- Asymptotic Notations and Basic Efficiency Classes
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
- Example: Computing the ***nth Fibonacci Number***
- Empirical Analysis of Algorithms
- Algorithm Visualization

The Efficiency Analysis Framework

- Analysis of algorithm is the theoretical study of computer program ***performance*** *(*processing speed*)* and resource usage (communication, primary and secondary memory).
- Analysis of algorithm's efficiency can be achieve in two resources
  - ***Time efficiency, also called time complexity,*** indicates how fast an algorithm in question runs ~ *performance*
  - ***Space efficiency, also called space complexity,*** *refers to the amount of memory units required by the algorithm in addition* to the space needed for its input and output ~ *resource usage.*
- *Other evaluable criteria of algorithm other than* ***performance*** *includes;* Correctness (Accuracy and precision), Simplicity(Ease), Maintainability (Continuity), Cost of programming time, Robustness, Stability, Features, Security and Scalability.

The Efficiency Analysis Framework…

- The efficiency analysis framework is not complete until the following questions are answered.
  - How to measure an Input’s Size
  - How to state Units for Measuring Running Time
  - How to Compute Orders of Growth
  - How to derive Worst, Best and Average Cases Efficiencies

Measuring an Input's Size

- We can investigate an algorithm's efficiency as a function of some parameter *n* indicating the algorithm's input size.
- The choice of Input size depends on the problem as shown in the examples;
  - Example 1: what is the **input size** for sorting *n* numbers say in a polynomial?
    - it will be the polynomial's degree or the number of its coefficients
  - Example 2: what is the **input size** for multiplying two $n \times n$ matrices?
    - The first and more frequently used is the matrix order *n*
  - Example 3: What is the **input's size** for a spell-checking algorithm?
    - If the algorithm examines individual characters of its input
      - we should measure the size by the number of characters
    - if it works by processing words
      - we should count their number in the input

How to State Units for Measuring Running Time

- When we measure the ***running time of a program implementing the algorithm*** in milliseconds, seconds, etc.
  - Drawbacks – so much dependence on ***extraneous factors*** *like;*
    - Speed of particular computer.
    - Quality of the program implementation of the algorithm.
    - Compiler used in generating the machine code.
    - Difficulty of clocking the actual running time of the program.
- Since we are after a measure of an *algorithm's efficiency,*
  - *we would like to* have a ***metric*** that does not ***depend*** on these ***extraneous factors***.
- Soln 1: Count the number of times each algorithm's operation is executed
  - Difficult and unnecessary
- Soln 2: Count the number of times an algorithm's ***"Basic Operation"*** is executed

Measuring Running Time base Basic Operation

- **Basic operation (Bop):** is the operation contributing the **most** to the **total running time**, and compute the number of times the Bop is executed.
- Usually the most ***time-consuming operation*** in the algorithm's ***innermost loop.***

Problem	Input size Measure	Basic operation
Search for a key in a list of ***n*** items	# of items in the list	Key comparison
Multiplication of two ***n*×*n*** matrices	Matrix dimensions or # of elements	Multiplication of two numbers
Typical graph problem	# of vertices and/or edges	Visiting a vertex or traversing an edge

Theoretical Analysis of Time Efficiency

- Let **C_{op}** = execution time of an algorithm's ***Bop*** on a particular computer,
- let **C*(n)*** be the number of times this operation needs to be executed for this algorithm.
- The we can estimate running time efficiency ***T(n)*** of a program implementing this algorithm on that computer by;

$$T(n) \approx C_{op}\, C(n).$$

Running time

Execution time for Bop

Number of times basic op. is executed

- Where ***n*** is the input size

What is the Orders of Growth

S/N	n	$log_2 n$	n	$n\ log_2 n$	n^2	n^2	n^2	$n!$
1	10	3.3	10	3.3×10	10^2	10^3	10^3	$3.6×10^6$
2	10^2	6.6	10^2	$6.6×10^2$	10^4	10^6	$1.3×10^{30}$	$9.3×10^{157}$
3	10^3	10	10^3	$10×10^3$	10^6	10^9		
4	10^4	13	10^4	$13×10^4$	10^8	10^{12}		
5	10^5	17	10^5	$17×10^5$	10^{10}	10^{15}		
6	10^6	20	10^6	$20×10^6$	10^{12}	10^{18}		

$$log_2 2n = log_2 2 + log_2 n = 1 + log_2 n$$

How to Derive Worst, Best and Average-Cases Efficiencies

- Worst-case (usually): $C_{worst}(n)$ Maximum over input of size of size n
- Best-case (Bogus): $C_{best}(n)$ Minimum over input of size of size n
  - We don't worry about this since some slower algorithm works faster on some inputs
- Average-case (Sometimes): $C_{avg}(n)$ expected time over input of size n
  - But we don't know the statistical distribution of the inputs
  - So we make assumption of the statistical distribution
    - like all inputs are equally likely possibly uniform distribution
  - NOT the average of worst and best cases

Sequential Search Algorithm

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n-1]$ and a search key K
//Output: The index of the first element in A that matches K
//   or -1 if there are no matching elements

```
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

How to Derive Average-Cases Efficiencies of Seq. Search

- Two assumptions:
  - Probability of successful search is p (0 ≤ p ≤ 1)
  - Search key can be at any index with equal prob. (uniform distribution)

$C_{avg}(n)$ = Expected # of comparisons for success + Expected # of comparisons if k is not in the list

$$\begin{aligned} C_{avg}(n) &= [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p). \end{aligned}$$

Recapitulation of the Analysis Framework

- **Time** and **space** efficiencies are measured as functions of the algorithm's input size.
- **Time efficiency** is measured by counting the number of times the algorithm's Bop is executed.
- **Space efficiency** is measured by counting the number of extra memory units consumed by the algorithm.
- Efficiencies of some algorithms may differ significantly for inputs of the same size.
- We need to distinguish between the **worst-case**, **average-case**, and **best-case** efficiencies.
- The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.

What next?

- The Efficiency Analysis Framework
- Asymptotic Notations and Basic Efficiency Classes
- General Plan for Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
- Example: Computing the ***nth Fibonacci Number***
- Empirical Analysis of Algorithms
- Algorithm Visualization

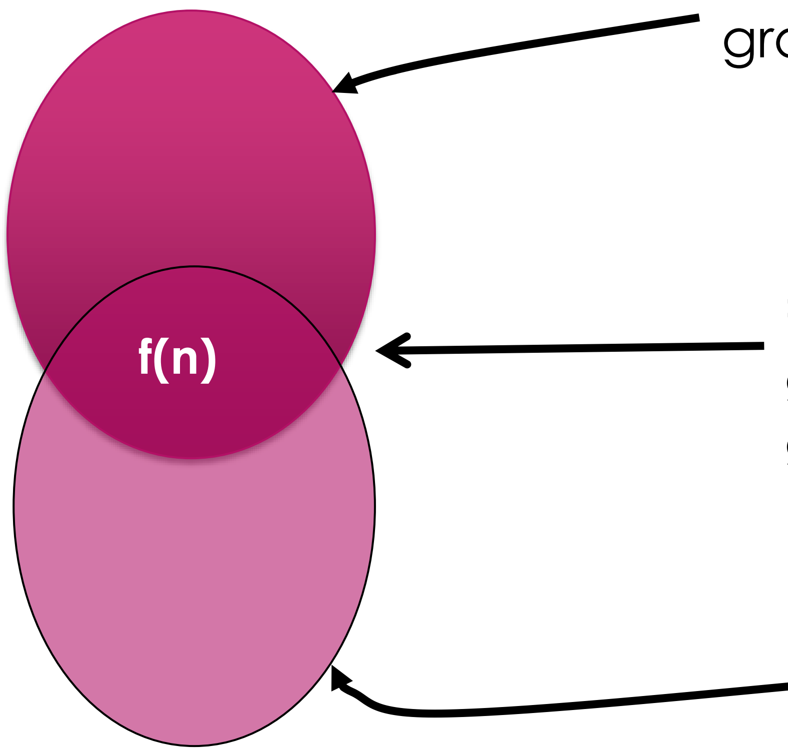
Asymptotic Notations and Basic Efficiency Classes

- Asymptotic other of growth is a way of comparing functions that ***ignores constant*** factors and ***small input sizes***
- It is a way of comparing size and functionality of a function

$$\boldsymbol{O} \approx \leq, \quad \Omega \approx \geq, \quad \Theta \approx =, \quad \mathrm{o} \approx <, \quad \omega \approx >,$$

- We can define asymptotic Order of growth in two **methods**:
- **Method 1:** Using **Theorem**
- **Method 2:** Using **definitions** of O, Ω, and Θ notations.

Asymptotic Notations and Basic Efficiency Classes



≥ Ω(g(n)): functions that grow at least as fast as g(n)

= Θ(g(n)): functions that grow at the same rate as g(n)

≤ O(g(n)): functions that grow no faster that g(n)

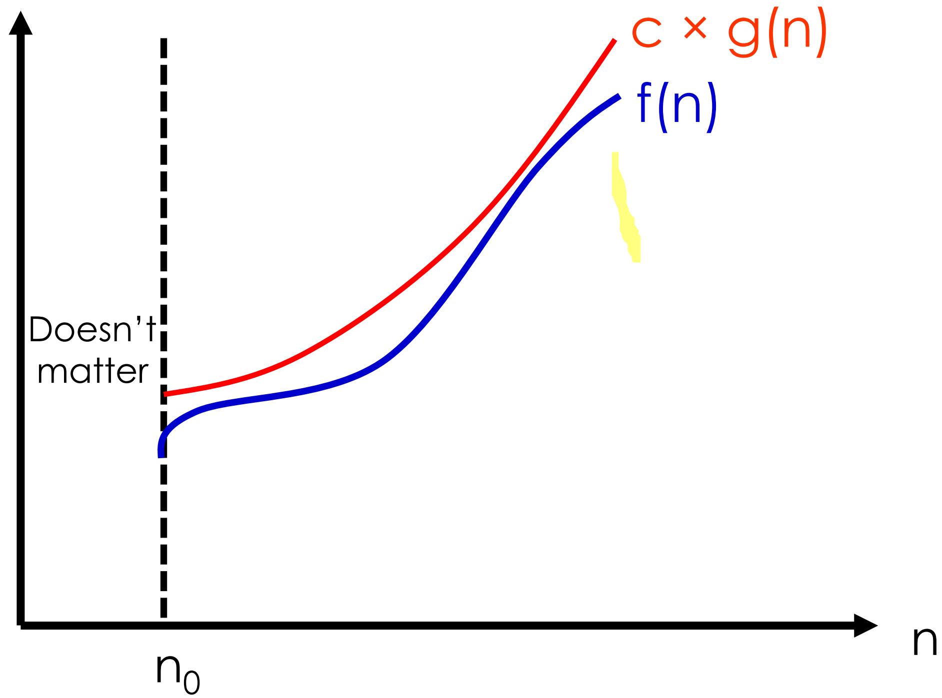
Asymptotic O(big oh)-Notation

- **Definition:** A function ***f(n)*** is said to be in ***O(g(n))***, denoted ***f(n) ϵ O(g(n))***, if ***f(n)*** is bounded above by some positive ***constant (c)*** multiple of ***g(n)*** for sufficiently large ***n***. If we can find +ve constants ***c*** and n_0 such that: $f(n) \le c \times g(n)$ *for all* $n \ge n_0$
- ***Then O(g(n))*** are set of functions that grow ***no faster*** than ***g(n)***. *Written as;*
  - **f(n)** ϵ **O(g(n))**

Example:

$10n+5$ is $O(n^2)$

$5n+20$ is $O(n)$

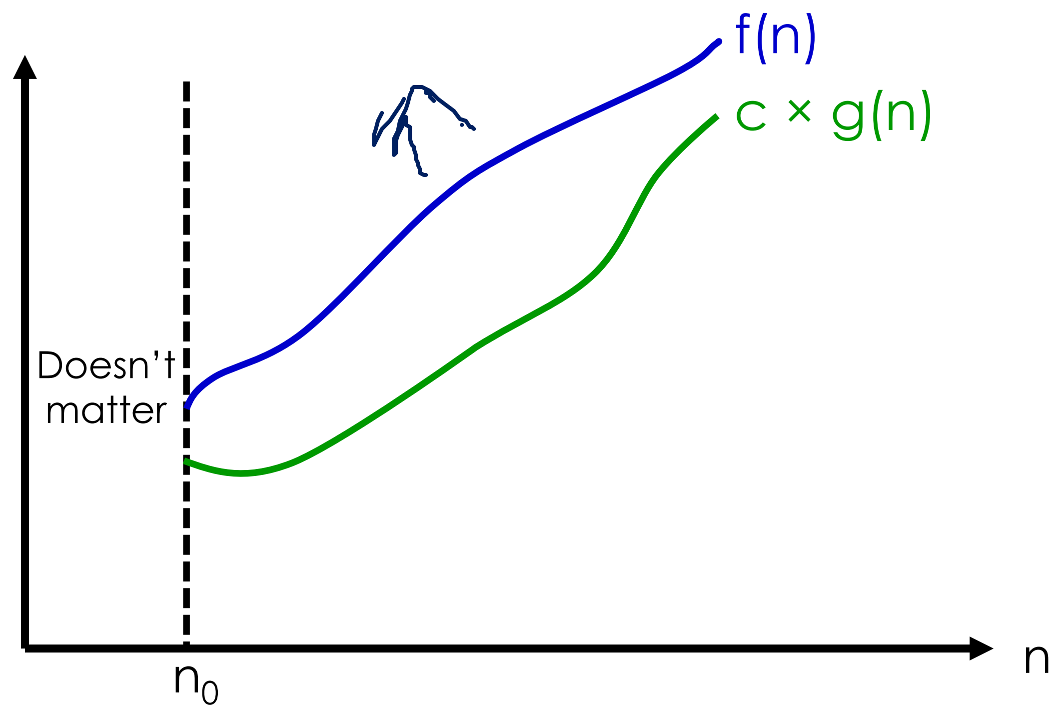


Try this ☺

- Is $100n+5 \in O(n^2)$?
- Is $2^{n+1} \in O(2^n)$?
- Is $2^{2n} \in O(2^n)$?
- Is $\frac{1}{2}n(n-1) \in O(n^2)$?

Asymptotic Ω(big omega)-Notation

- **Definition:** A function ***f(n)*** is said to be in ***Ω(g(n))*** denoted ***f(n) ϵ Ω(g(n))***, if ***f(n)*** is bounded below by some positive constant multiple of ***g(n)*** for all sufficiently large ***n.*** If we can find +ve constants ***c*** and n_0 such that ***f(n) ≥ c × g(n)*** for all $n \geq n_0$
- ***Then Ω(g(n)) are*** Set of functions that grow <u>at least as fast</u> as ***g(n). Written as;***
  - ***f(n) ϵ Ω(g(n))***

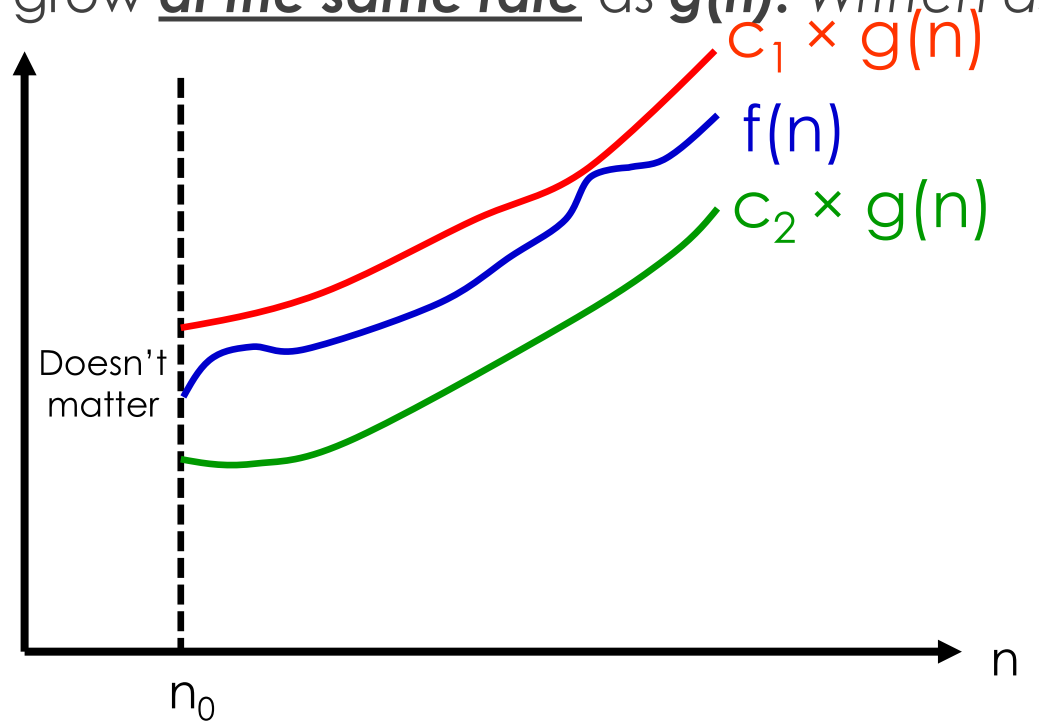


Try this ☺

- Is $n^3 \in \Omega(n^2)$?
- Is $100n+5 \in \Omega(n^2)$?
- Is $\frac{1}{2}n(n-1) \in \Omega(n^2)$?
- Is $\frac{1}{4}n(n+1) \in \Omega(n^3)$?

Asymptotic Θ(big theta)-Notation

- **Definition:** A function ***f(n)*** is said to be in ***Θ(g(n))*** denoted ***f(n) є Θ(g(n))***, if ***f(n)*** is bounded both above and below by some positive constant multiples of ***g(n)*** for all sufficiently large ***n***. If we can find +ve constants c_1, c_2, and n_0 such that. $c_2 \times g(n) \le f(n) \le c_1 \times g(n)\ \forall\ n \ge n_0$
- ***Then Θ(g(n)):*** Set of functions that grow ***at the same rate*** as ***g(n)***. *Written as;*
  - ***f(n) ∈ Θ(g(n))***



Try this ☺

- Is $\frac{1}{2}n(n-1) \in \Theta(n^2)$?
- Is $n^2+\sin(n) \in \Theta(n^2)$?
- Is $an^2+bn+c \in \Theta(n^2)$ for $a > 0$?
- Is $(n+a)^b\ \Theta(n^b)$ for $b > 0$?

Asymptotic Notations and Basic Efficiency Classes Generalization

- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then $f_1(n) + f_2(n) \in (\max\{g_1(n), g_2(n)\})$
- Analogous assertions are true for Ω and Θ notations.
- **Implication:** if sorting makes no more than $\mathbf{n^2}$ comparisons and then binary search makes no more than $\boldsymbol{log_2 n}$ comparisons, then efficiency is given by $O(\max\{n^2, \log_2 n\}) = O(n^2)$
- $f_1(n) \le c_1 g_1(n)$ for $n \ge \mathbf{n_{01}}$ and $f_2(n) \le c_2 g_2(n)$ for $n \ge \mathbf{n_{02}}$
  - $f_1(n) + f_2(n) \le \mathbf{c_1} g_1(n) + \mathbf{c_2} g_2(n)$ for $n \ge \max\{\mathbf{n_{01}}, \mathbf{n_{02}}\}$
  - $f_1(n) + f_2(n) \le \underline{\max\{c_1, c_2\}} * \max\{g_1(n), g_2(n)\}$, for $n \ge \max\{n_{01}, n_{02}\}$
  - …
  - $f_k(n) \le c_k \times g_k(n)$ for $n \ge n_k$

M. Yusuf

Using Limits for Comparing Orders of Growth

- We compare the orders of growth of two specific functions by computing the limit of the ratio of two functions in question.

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

- The first two solution 0 and c ≈ f(n) ϵ O(g(n))
- The second case c ≈ f(n) ∈ Θ(g(n))
- The third case c and ∞ ≈ f(n) ϵ Ω(g(n))

calculus techniques for computing limits

- L'Hôpital's rule. $$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\lim_{n\to\infty}\frac{t'(n)}{g'(n)}$$

- Stirling's formula $n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$ for large values of n.

Calculus Techniques For Computing Limits

Example 1. Compare the order of growth of $\frac{1}{2}n(n-1)$ and n^2.

Solution

Using L'H$\hat{o}$pital's rule $= \lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \lim\limits_{n\to\infty} \frac{f\prime(n)}{g\prime(n)} = \lim\limits_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2}$

$$= \frac{1}{2}\lim_{n\to\infty} \frac{n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2}{n^2} - \frac{n}{n^2} = \frac{1}{2}\lim_{n\to\infty} 1 - \frac{1}{n} = \frac{1}{2}$$

Since the limit is equal to a positive constant, the functions have the same order of growth, or, symbolically we say $\frac{1}{2}n(n-1) \epsilon\, \theta(n^2)$

Basic Asymptotic Efficiency Classes

Class	Notation	Example
constant	1	May be in best cases, hashing (on average)
logarithmic	$log_2 n$	Binary search (worst and average cases)
linear	n	Sequential search (worst and average cases)
linearithmic	$n \times log_2 n$	Divide and conquer algorithms, e.g., merge sort
quadratic	n^2	Two embedded loops, e.g., selection sort
cubic	n^3	Three embedded loops, e.g., matrix multiplication
exponential	2^n	All subsets of n-elements set Gaussian elimination
factorial	n!	All permutations of an n-elements set, combinatorial problems

What next?

- The Efficiency Analysis Framework
- Asymptotic Notations and Basic Efficiency Classes
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
- Example: Computing the ***nth Fibonacci Number***
- Empirical Analysis of Algorithms
- Algorithm Visualization

General Plan for Analysis of Nonrecursive Algorithms

1. Decide on parameter *n indicating* ***input size***
2. Identify algorithm's ***Bop***
3. Determine ***worst, average, and best cases for*** *input* of size *n*
4. Set up a sum for the **number of times** the Bop is executed
5. Simplify the sum using **standard formulas** and rules to establish its order of growth

Properties of Logarithms

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_a x^y = y \log_a x$
4. $\log_a xy = \log_a x + \log_a y$
5. $\log_a \frac{x}{y} = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \frac{\log_b x}{\log_b a} = \log_a b \log_b x$

Important Summation Formulae

1. $\sum_{i=l}^{u} 1 = \underbrace{1+1+\cdots+1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \leq u$); $\sum_{i=1}^{n} 1 = n$

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

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

Important Summation Formulae…

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \ (a \neq 1); \quad \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772\ldots$ (Euler's constant)

8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Sum Manipulation Rules

1. $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$

2. $\sum_{i=l}^{u}(a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i$

3. $\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i$, where $l \le m < u$

4. $\sum_{i=l}^{u}(a_i - a_{i-1}) = a_u - a_{l-1}$

Analysis of Unique Elements Algorithms

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise

for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

Solution for Analysis of Unique Elements Algorithm

1. **Input size: Array** A[*0,…, n-1*]
2. **Bop:** if A[*i*] = A[*j*]
3. **Worst case:** When A not distinct
4. **Set up a sum:** $C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$
5. **Establish order of growth**

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$$

- The complexity class of unique element algorithm is ***quadratic***

Solution for Analysis of Unique Elements Algorithms

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

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

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

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

Analysis of Maximum Element Algorithms

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

Solution for Analysis of Maximum Element Algorithms

$$\textbf{if } A[i] > maxval$$
$$\quad maxval \leftarrow A[i]$$

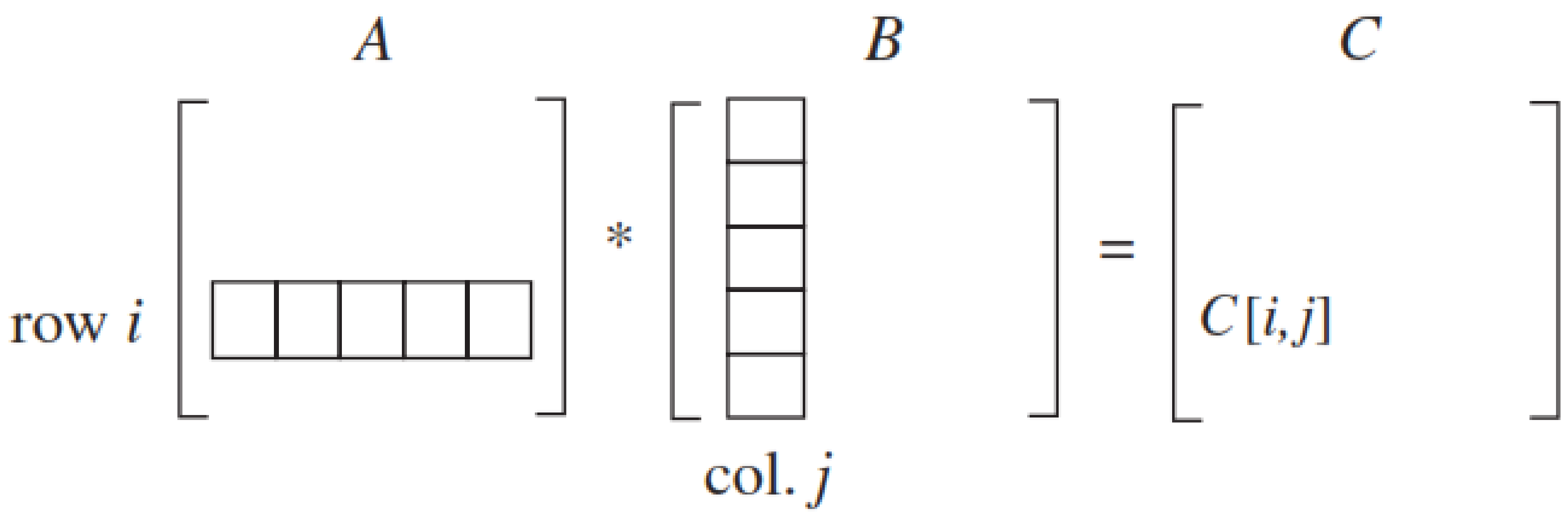
$$C(n) = \sum_{i=1}^{n-1} 1.$$

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Analysis of Matrix Multiplication Algorithms

```
ALGORITHM  MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])
    //Multiplies two square matrices of order n by the definition-based algorithm
    //Input: Two n × n matrices A and B
    //Output: Matrix C = AB
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            C[i, j] ← 0.0
            for k ← 0 to n − 1 do
                C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
    return C
```

Solution for Analysis of Matrix Multiplication Algorithms…



where $C[i, j] = A[i, 0]B[0, j] + \cdots + A[i, k]B[k, j] + \cdots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \le i, j \le n-1$.

Solution for Analysis of Matrix Multiplication Algorithms

$$\sum_{k=0}^{n-1} 1,$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

$$T(n) \approx c_m M(n) = c_m n^3,$$

What next?

- The Efficiency Analysis Framework
- Asymptotic Notations and Basic Efficiency Classes
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
- Example: Computing the ***nth Fibonacci Number***
- Empirical Analysis of Algorithms
- Algorithm Visualization

Mathematical Analysis of Recursive Algorithms

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Mathematical Analysis of Recursive Algorithms

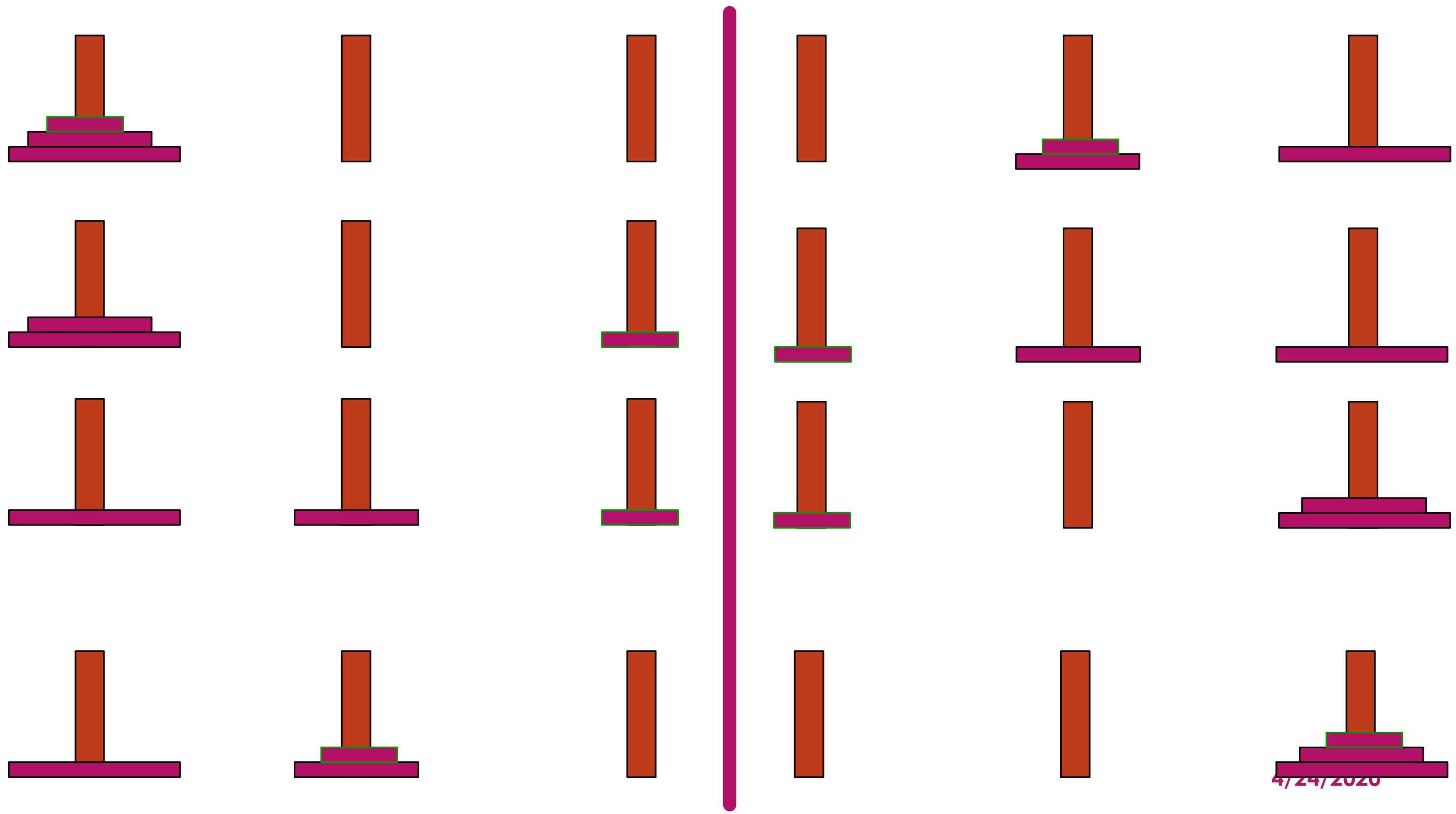
1. Decide on **input size** parameter
2. Identify the **Bop**
3. Does $C_{worst}(n)$ depends also on **input type?**
4. Set up a ***recurrence relation***
5. Solve the recurrence or, at least establish the order of growth of its solution

Analysis of Recursive Algorithm: Tower of Hanoi

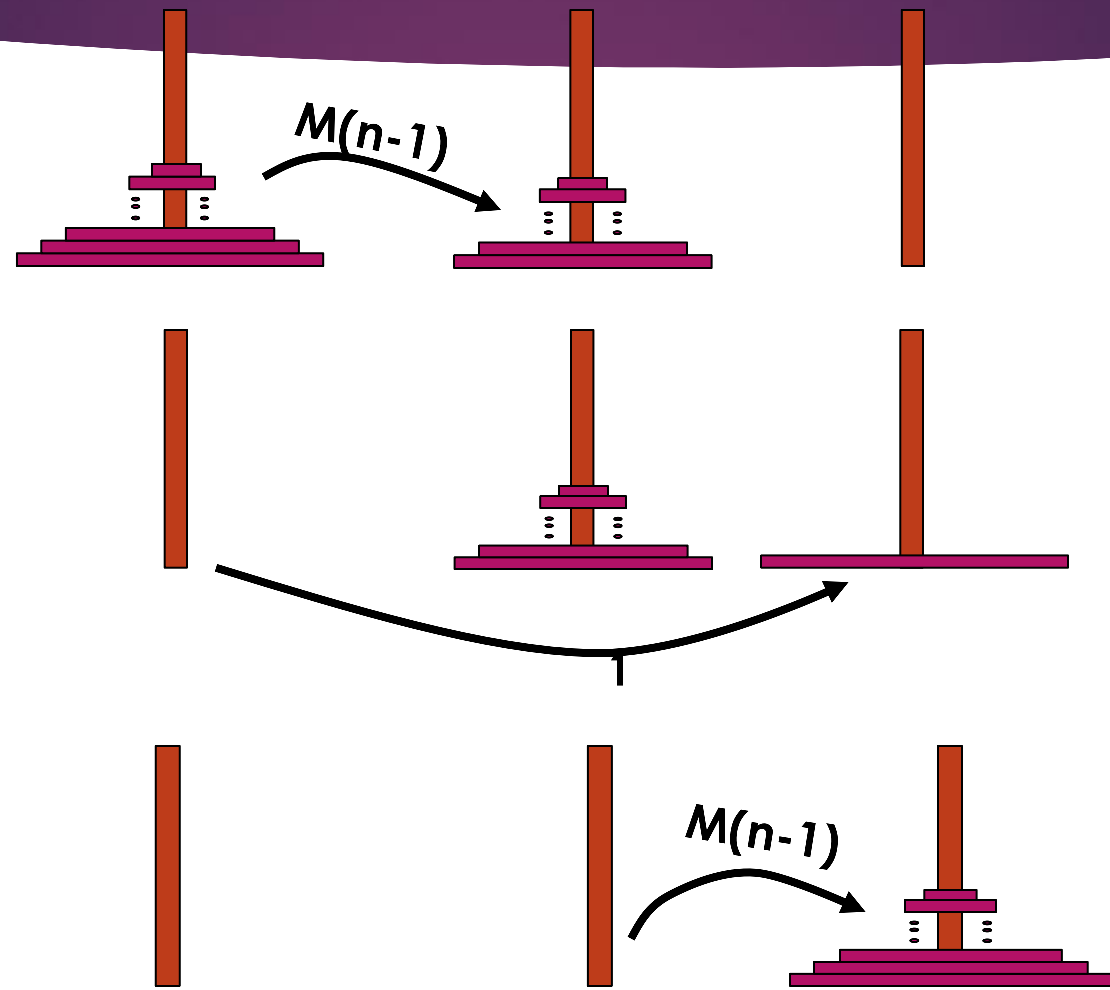
The goal is to **move all the disks to the third peg,** using the second one as an auxiliary, if necessary.

We can **move only one disk** at a time, and

it is **forbidden to place a larger disk on top of a smaller one.**



Analysis of Recursive Algorithm: Tower of Hanoi



$M(n) = M(n-1) + 1 + M(n-1)$
$M(1) = 1$

Analysis of Recursive Algorithm: Tower of Hanoi

$M(n) = M(n\text{-}1) + 1 + M(n\text{-}1)$ for $n > 1$

$M(1) = 1$

[Using Backward substitution…] *that is sub $M(n\text{-}1) = 2M(n\text{-}2)+1$*

$M(n) = 2M(n\text{-}1) + 1$

$M(1) = 2[\mathbf{2M(n\text{-}2)+1}] + 1 = 2^2M(n\text{-}2)+2+1$

$M(2) = 2^2[2M(n\text{-}3)+1] + 2 + 1 = 2^3M(n\text{-}3)+ 2^2+2+1$

$M(3) = 2^3[2M(n\text{-}4)+1] + 2^2 + 2 + 1 = 2^4M(n\text{-}4)+ 2^3 + 2^2 + 2 + 1$

… … …

$M(n) = 2^n[2M(\mathbf{n\text{-}(n+1)})+1] + 2^{n\text{-}1} + 2^{n\text{-}2} + 2^{n\text{-}3} + \ldots\ldots + 2^2 + 2 + 1$

$M(n) = 2^{n+1}M(\mathbf{\cancel{n\text{-}n}\text{-}1)}) + 2^n + 2^{n\text{-}1} + 2^{n\text{-}2} + 2^{n\text{-}3} + \ldots\ldots + 2^2 + 2 + 1$

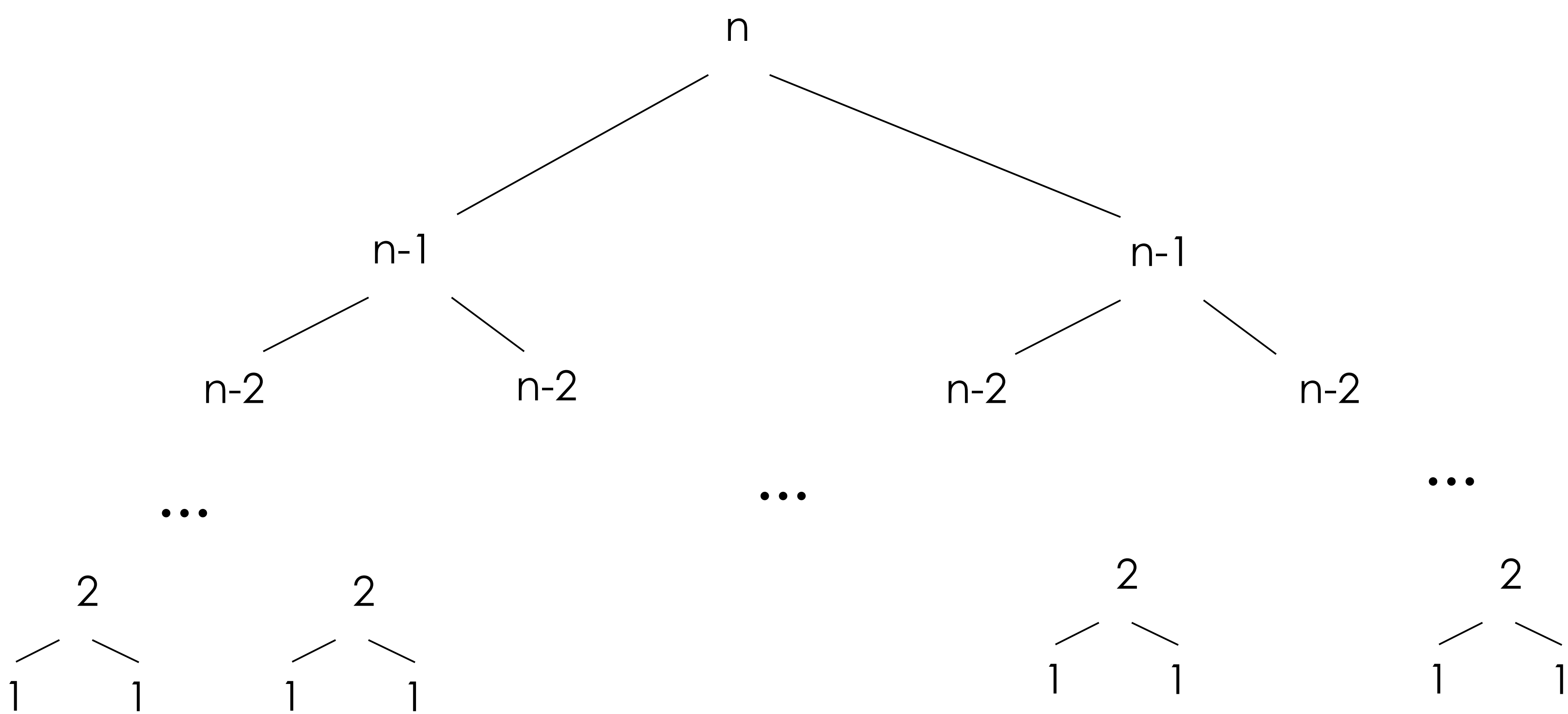
$M(n) = 2^{n+1}M(\text{-}1) + 2^n + 2^{n\text{-}1} + 2^{n\text{-}2} + 2^{n\text{-}3} + \ldots\ldots + 2^2 + 2 + 1$

$M(n) = \sum_{i=0}^{n} 2^i = 2^n\text{-}1$ therefore $\boldsymbol{M(n) \in \Theta(2^n)}$

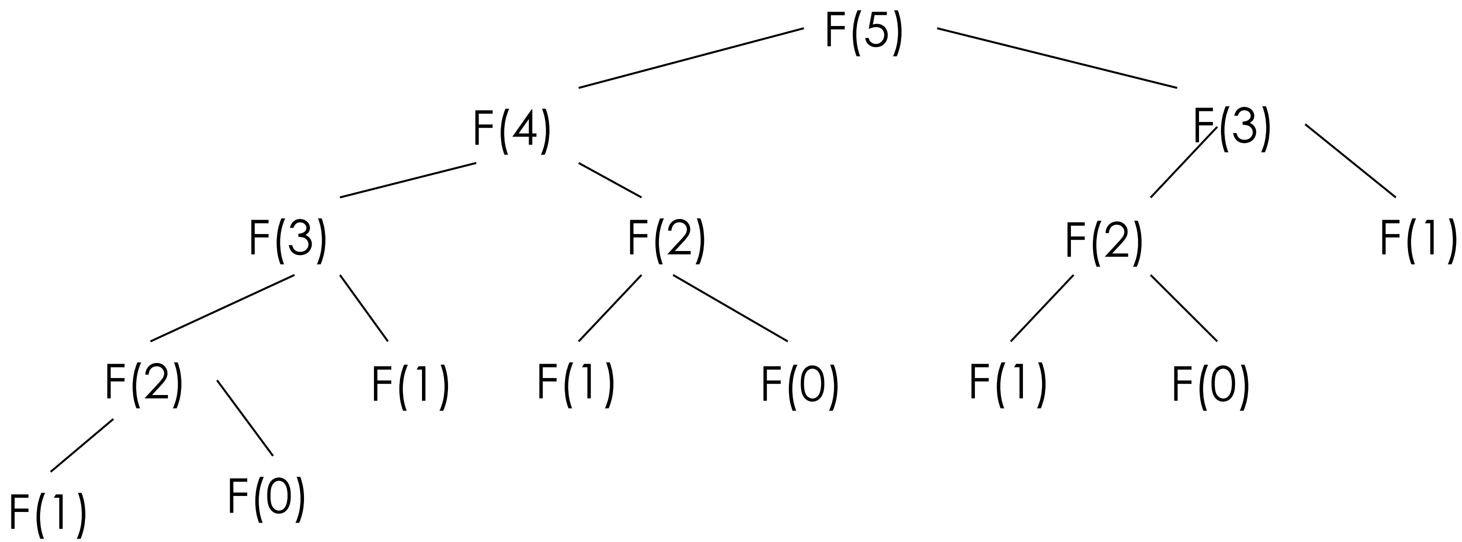
What next?

- The Efficiency Analysis Framework
- Asymptotic Notations and Basic Efficiency Classes
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
- Example: Computing the ***nth Fibonacci Number***
- Empirical Analysis of Algorithms
- Algorithm Visualization

Mathematical Analysis of Recursive Algorithms



Mathematical Analysis of *n*th Fibonacci



Only n-1 additions, Θ(n)!

```
ALGORITHM Fib(n)
  F[0] <- 0
  F[1] <- 1
  for i <- 2 to n do
            F[i] <- F[i-1]+F[i-2]
  return F[n]
```

```
ALGORITHM Fib(n)
f <- 0      fnext <- 1
for i <- 2 to n do
        tmp <- fnext
        fnext <- fnext+f
        f <- tmp
return fnext
```

What next?

- The Efficiency Analysis Framework
- Asymptotic Notations and Basic Efficiency Classes
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
- Example: Computing the ***nth Fibonacci Number***
- Empirical Analysis of Algorithms
- Algorithm Visualization

Empirical Analysis of Algorithms

- Empirical Analysis
  - Advantages
    - Applicable to **any** algorithm
  - Disadvantages
    - Machine and input **dependency**
- Mathematical Analysis
  - Advantages
    - Machine and input independence
  - Disadvantages
    - Average case analysis is **hard**

When to Consider Empirical Analysis of Algorithm

- To **check the accuracy** of a theoretical assertion about the algorithm's efficiency,
- To **compare the efficiency** of several algorithms for solving the same problem or different implementations of the same algorithm,
- To **develop a hypothesis** about the algorithm's efficiency class,
- To **ascertain the efficiency** of the program implementing the algorithm on a particular machine.

Obviously, **an experiment's design** should depend on the question the experimenter seeks to answer.

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment’s **purpose.**
2. Decide on the **efficiency metric *M*** to be measured and the measurement unit(an operation count vs. a time unit).
3. Decide on **characteristics** of the input sample (its range, size, and so on).
4. Prepare a program **implementing** the algorithm (or algorithms) for the experimentation.
5. Generate a **sample** of inputs.
6. **Run** the algorithm (or algorithms) on the sample’s inputs and record the data observed.
7. **Analyze** the data obtained.

How to Perform Empirical Analysis of Algorithm

- The **goal** of the experiment should influence, if not dictate, how the **algorithm's efficiency** is to be measured.
  1. To insert a counter (or counters) into a program implementing the algorithm to count the number of **times** the **algorithm's Bop** is executed.
  2. To **measure time** of the program implementing the algorithm in question.
     a. Use a **system's command**,
     b. Use **code fragment** by asking for the system time right before the fragment's start (t_{start}) and just after its completion (t_{finish}), and then computing the difference between the two (t_{finish} - t_{start}).
  3. To use a sample representing a "typical" ***see next slide***
  4. Data can be presented numerically in a **table** or graphically in a ***scatterplot.***
     - ***Advantages: Opportunity for easy manipulation***

How to Generate input in Empirical Analysis of Algorithm

- Input should adhere to some systematical pattern like(1,000, 2,000, 3,000, …, 10,000) or (500, 1,000, 2,000, 4,000, …, 128,000)
  - Advantage: impact is easier to analyze.
  - Disadvantage: the algorithm under investigation exhibits atypical behavior on the sample chosen.
    - E.g. **fast even** samples and **slow Odd** samples = misleading.
- Better is to generate random sizes within desired range
  - Pseudorandom number generators
  - *linear congruential method*

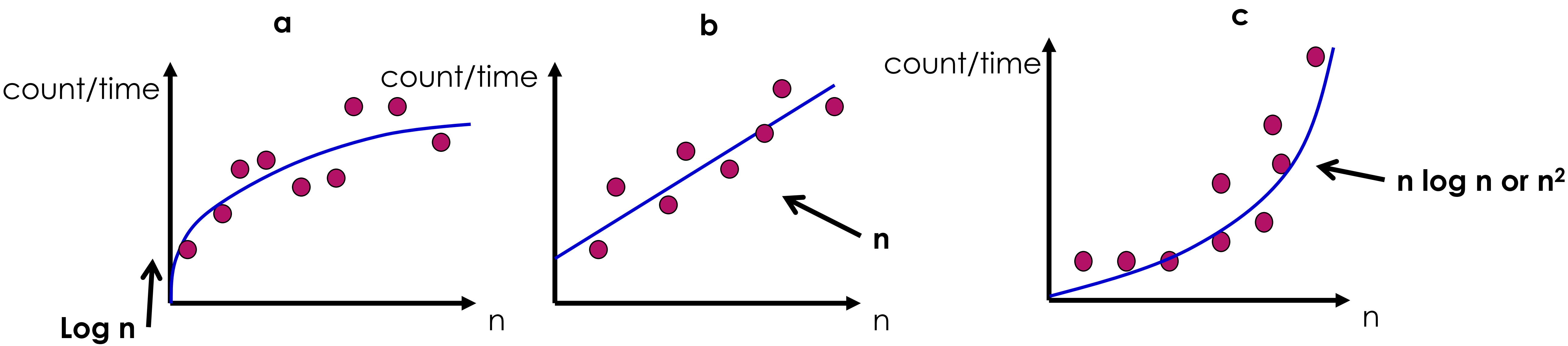
What next?

- The Efficiency Analysis Framework
- Asymptotic Notations and Basic Efficiency Classes
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
- Example: Computing the ***nth Fibonacci Number***
- Empirical Analysis of Algorithms
- Algorithm Visualization

Visualization of Empirical Analysis of Algorithm

Tabulation

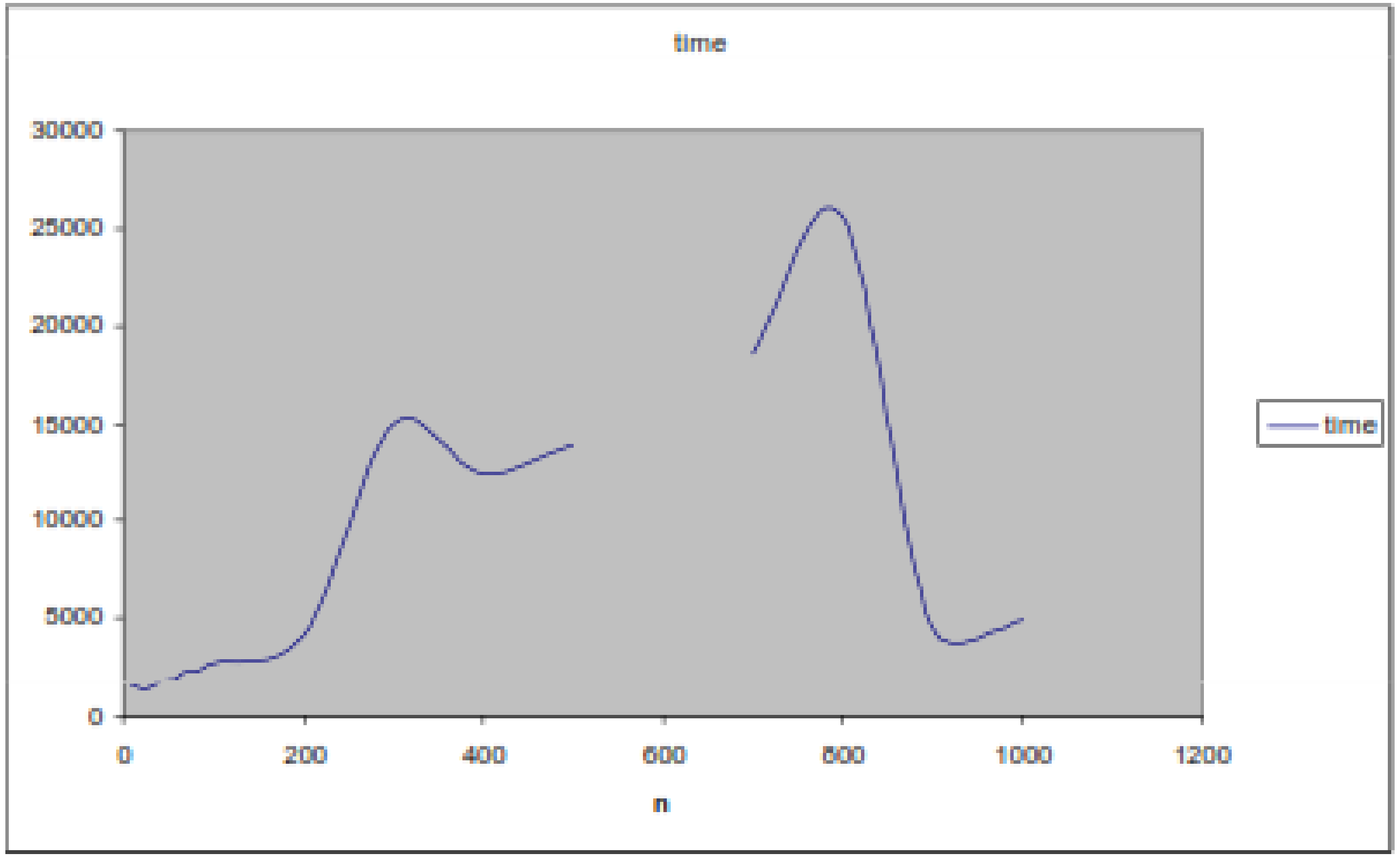
n	f(n)	g(n)	f(n)/g(n)



Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions

Example of Empirical Analysis Result

n	time
0	1572954
10	2013
20	2237
30	2520
40	3288
50	3871
60	3439
70	6520
80	5774
90	6260
100	4615
200	7587
300	9999
400	12696
500	15607
600	29191
700	18299
800	21851
900	5026
1000	5399



Summary

- Time and Space Efficiency Analysis
- C(n): Count of # of times the Bop is executed for input of size n
- C(n) may depend on type of input and then we need worst, average, and best case analysis
- order of growth (O, Ω, Θ) is all that matters: logarithmic, linear, linearithmic, quadratic, cubic, and exponential
- Input size, Bop, worst case?, sum or recurrence,
- We run on computers for empirical analysis
- Empirical analysis can be used to test any algorithm by Machine and input dependency
- Mathematical Analysis is machine and input independence but difficult to achieve

Thank You