CMP418: Data Structures and Algorithms 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

Chapter 2 Outline

- 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

The Efficiency Analysis Framework

- Analysis of algorithm is 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 basic operation 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 basic operation

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^3	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}		

$$log_2 n = log_2 b \ \ log_b n$$

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

How to Derive Worst, Best and Average-Cases Efficiencies

- Worst-case (usually): $\boldsymbol{C_{worst}(n)}$ Maximum over input of size of size n
- Best-case (Bogus): $\boldsymbol{C_{best}(n)}$ 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): $\boldsymbol{C_{avg}(n)}$ 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 inputs
  - 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
$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
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 ($0 \le p \le 1$)
  - 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

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

$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1 - p)$$

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

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 **basic operation** 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.
- For such algorithms, 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

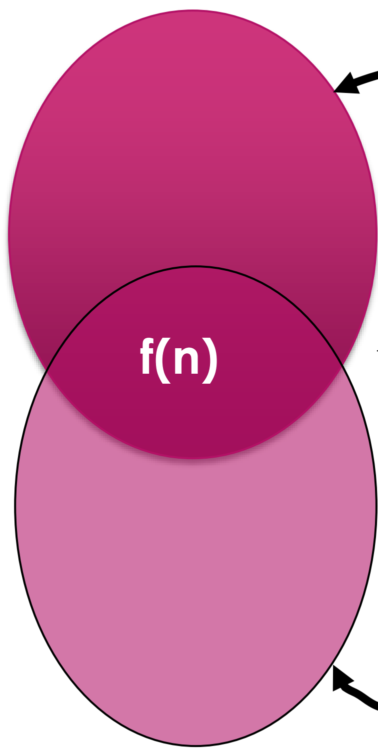
Asymptotic Notations and Basic Efficiency Classes

- 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
- Its is a way of describing the characteristics of a function in the limit

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

- 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



(≥) $\Omega(g(n))$: functions that grow at least as fast as g(n)

(=) $\Theta(g(n))$: functions that grow at the same rate as g(n)

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

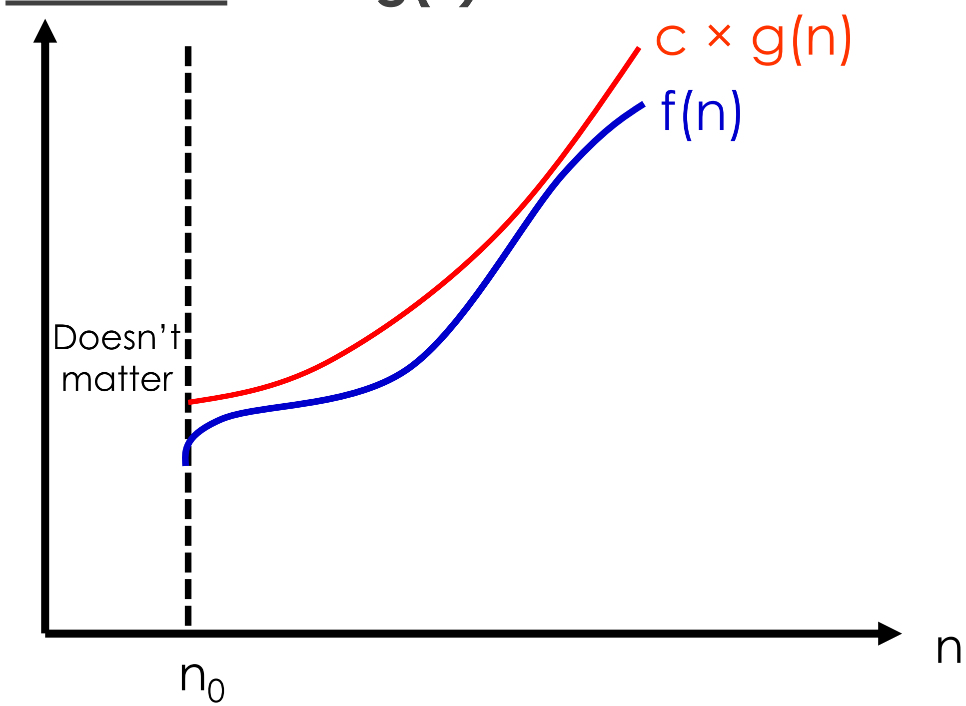
Asymptotic O(big oh)-Notation

- **Definition:** A function t(n) is said to be in ***O(g(n))***, denoted ***f(n) є O(g(n))***, if ***f(n)*** is bounded above by some positive ***constant*** multiple of ***g(n)*** for sufficiently large ***n.*** If we can find +ve constants ***c*** and n_0 such that: $f(n) \leq c \times g(n)$ *for all* $n \geq n_0$
- ***O(g(n)):*** Set of functions that grow ***no faster*** than ***g(n).***
  - **f(n)** є **O(g(n))**

Example:

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

5n+20 is O(n)

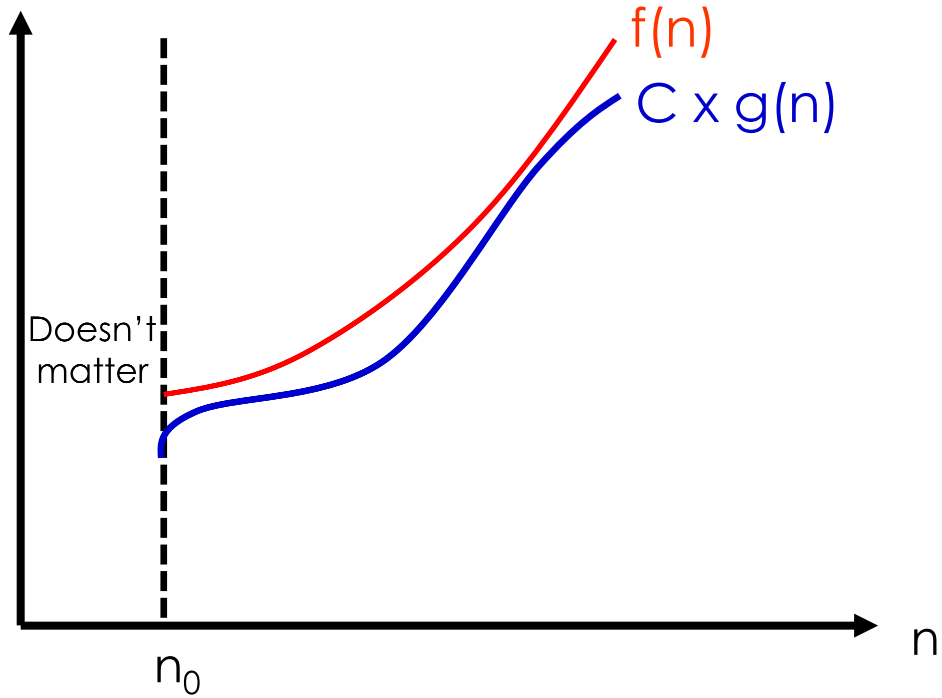


Try this ☺

- 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 t(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 sufficiently large ***n.*** If we can find +ve constants ***c*** and n_0 such that: $f(n) \geq c \times g(n)$ ***for all*** $n \geq n_0$
- ***Ω(g(n)):*** Set of functions that grow ***no faster*** than ***f(n).***
  - **f(n) ϵΩ(g(n))**

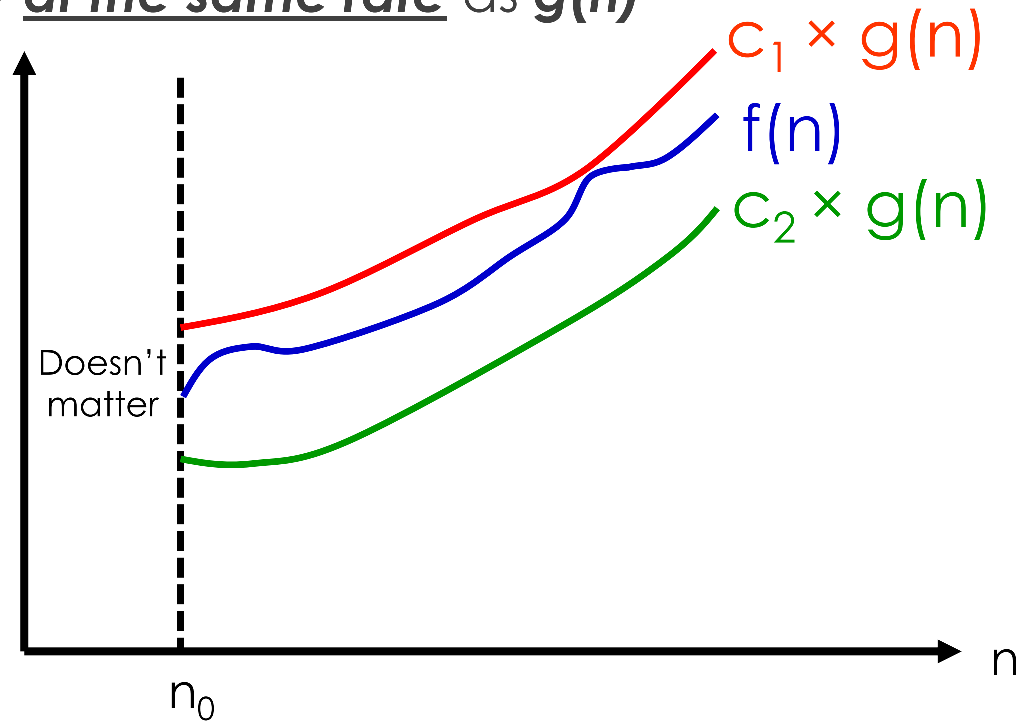


Try this ☺

- 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. $c_2 \times g(n) \leq f(n) \leq c_1 \times g(n)\ \forall\ n \geq n_0$
- ***Θ(g(n)):*** Set of functions that grow ***at the same rate*** as ***g(n)***
  - ***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

- 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 $n \ge \max\{\mathbf{n_{01}}, \mathbf{n_{02}}\}$
  - $f_1(n) + f_2(n) \le \underline{\max\{\mathbf{c1, c2}\}}\, g_1(n) + \underline{\max\{\mathbf{c1, c2}\}}\, g_2(n)$ for $n \ge \max\{ n_{01}, n_{02} \}$
  - $f_1(n) + f_2(n) \le \underline{2\max\{c_1, c_2\}} \max\{g_1(n), g_2(n)\}$, for $n \ge \max\{n_{01}, n_{02}\}$
  - $f_k(n) \le c_k \times g_k(n)$ for $n \ge n_k$

Basic Asymptotic Efficiency Classes

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

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. $\log_a \dfrac{x}{y} = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \dfrac{\log_b x}{\log_b a} = \log_a b \log_b x$

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, $l \le u$); $\sum_{i=1}^{n} 1 = n$

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

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

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 ***basic operation***
3. Determine ***worst, average, and best cases for*** *input of size n*
4. Set up a sum for the **number of times** the basic operation is executed
5. Simplify the sum using **standard formulas** and rules to establish its order of growth

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 Algorithms

1. **Input size: Array** A[*0,…, n-1*]
2. **basic operation:** if A[*i*] = A[*j*]
3. **Worst case:** $\sum_{j=i+1}^{n-1} 1$
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**

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

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

$$\begin{array}{l}\textbf{if } A[i] > maxval \\ \qquad maxval \leftarrow A[i]\end{array}$$

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

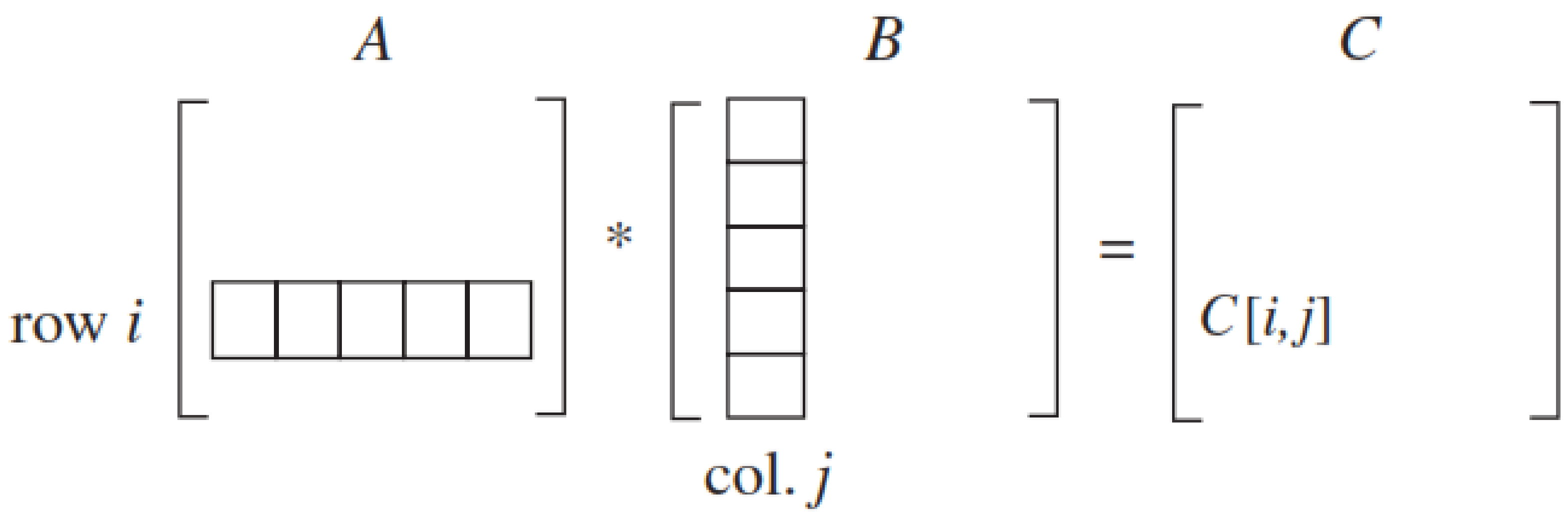
$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

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 $n \times n$ matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
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

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

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

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

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

Analysis of Binary Algorithms

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

Thank You