1

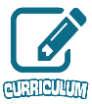
DATA STRUCTURES AND ALGORITHMS - (CIE 205)

LEC_I ALGORITHMS COMPLEXITY ANALYSIS

FALL (2024-2025)

COURSE DESCRIPTION

This course introduces formal techniques to support the design and analysis of algorithms, focusing on both the underlying mathematical theory and practical considerations of efficiency. Introduced design approaches will be supported by some common data structures such as arrays, linked lists, stacks, and queues, with a focus on some advanced ones such as trees, balanced search trees, graphs, and heaps. Topics include mathematics foundation for algorithms complexity estimation, different algorithms design techniques such as: Brute force, divide-and-conquer, speed for time tradeoff, and greedy algorithms, In addition to some famous algorithms such as shortest path algorithms, and string matching algorithms



2

COURSE LEARNING OBJECTIVES

- Use **arrays** and **linked lists** to implement and apply linear lists and the related operations (**insertion, deletion and traversal**).
- Implement and use special linear lists (**stacks and queues**) and the related basic operations (**push and pop; Enqueue and Dequeue**).
- Implement and use simple **sorting algorithms**.
- Apply searching algorithms including **sequential search, binary search and hashing** techniques.
- Apply different **String Matching** algorithms.
- Use the **graph** data structure and apply common **graph algorithms.**
- Implement and use the non-linear list **tree** data structures (**Binary trees, Binary Search Trees**) and the related operations (insert, delete and traverse).
- Compare and Select or improve an appropriate solution to a problem.
- Develop teamwork discipline through working in teams.

3

ASSESSMENT STRATEGY

Grading:	
Quizzes	12 %
Lab Assignments	5%
Course Project	15 %
Lab Exam	8 %
Midterm Exam	20 %
Final Exam	40 %

4

Agenda

- **What is an Algorithm?**
- **Data Structures Definition**
- **The analysis framework**
  - Order of growth
  - Worst case, best case and average case efficiencies
  - Importance of analysis
- **Asymptotic notations and basic efficiency classes**

1- sequence
2- Computational
3- procedure
4 - strategy

What is an Algorithm?

- An **Algorithm** is:
  - A sequence of **well-defined instructions** for solving a problem.
  - A computational procedure that takes some values as input and produces some values as output.
  - A procedure for getting answers to a specific problem
  - A problem solving strategy even if computers are not involved.

Problem solving strategy

1. Know standered set
2. design analyze.

Why do you need to study algorithms?

1- • **Practical reasons**
  - Know a standard set of important algorithms from different areas of computing.
  - Be able to design new algorithms and analyze their efficiency.

2- • **Theoretical reasons**
  - The study of algorithms was recognized as the cornerstone of computer science.
  - Studying algorithms helps in developing analytical skills.

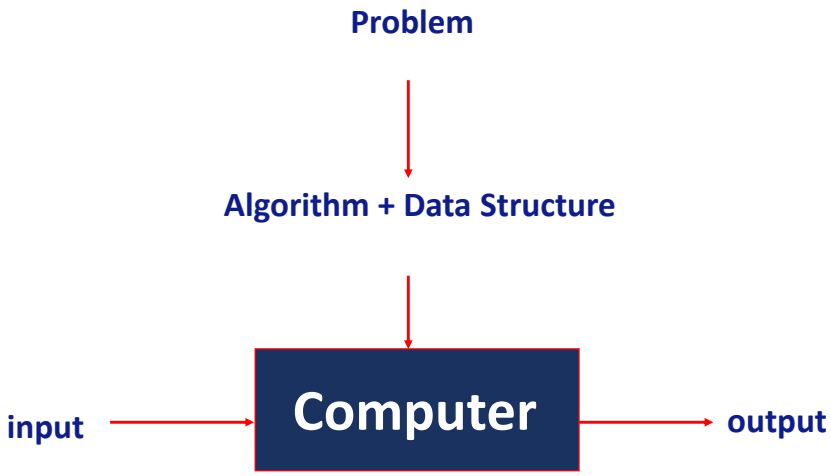
7

Representation of the logical relationship existing between individual elements of data.

Data Structure Definition

- Data structure is **representation of the logical relationship** existing between individual elements of data.
- ***In other words***, a data structure is ***a way of organizing all data items*** that considers not only the elements stored but also their relationship to each other.

8



9

Program = algorithm + Data Structure

- Algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, ***we should select an appropriate data structure for that algorithm.***

10

Agenda

Sequence of well defined instructions to solve Probl

- **What is an Algorithm?**
- **Data Structures Definition**

Representation of the logical Relationship existing between elements of data.

- **The analysis framework**
  - Order of growth
  - Worst case, best case and average case efficiencies
  - Importance of analysis
- **Asymptotic notations and basic efficiency classes**

11

Analysis of Algorithms

- **How good is the algorithm?**
  - **Efficiency**
    - Time efficiency
    - Space efficiency

12

1- input's size

2- Running time:

Analysis Framework

1- • **Measuring an input's size**

– All algorithms run longer on larger input

– An algorithm's efficiency is a function of "n" which is the algorithm's input size

– **Example:**

• Sorting an array : the input size is the array length

2- • **Measuring running time**

– Counting the number of times the algorithm's "**basic operation**" is executed on inputs of size "n"

13

Analysis Framework

Time efficiency: is analyzed by determining the number of repetitions of the ***basic operation*** as a function of ***input size***

✂ ***Basic operation***: the operation that contributes the most towards the running time of the algorithm

Time = t.n → number of basic operations

the time for basic operation

input size

$$T(n) \approx c_{op} C(n)$$

running time

execution time for basic operation or cost

Number of times basic operation is executed

14

Example

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of *n* items	n	comparing
Multiplication of two matrices	n x n	multiple

15

Importance of analysis

how fast

- It gives an idea about **how fast the algorithm**

$$T(n) \approx c_{op} C(n)$$

basic operations

- If the number of basic operations

$$C(n) = \tfrac{1}{2}\, n\,(n-1) = \tfrac{1}{2}\, n^2 - \tfrac{1}{2}\, n \approx \tfrac{1}{2}\, n^2$$

How much longer if the algorithm doubles its input size?

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

$\frac{T_2}{T_1} = \frac{4n^2}{n^2}$ $\quad T_2 = 4\, T_1$

$$\frac{T(2n)}{T(n)} \approx \frac{C_{OP} C(2n)}{C_{OP} C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}(n)^2} = 4$$

✂ **Increasing input size increases the complexity**

16

Importance of analysis

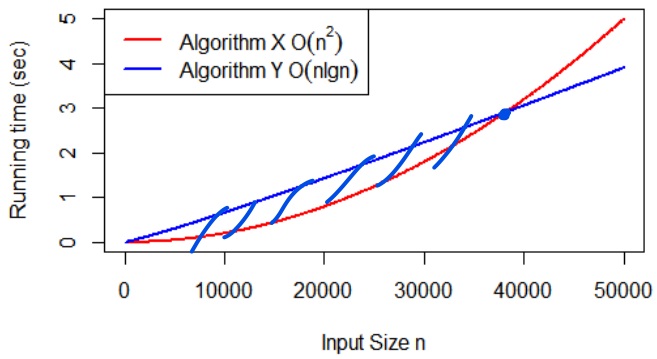
- Assume:
  - Algorithm X takes time $2n^2$ (written by best programmer) running on machine with **1000** MIPS (Million instructions per second)
  - Algorithm Y takes time $50n \lg n$ (written by worst programmer) running on machine with **10** MIPS
- Running time for 10^6 numbers
  - Algorithm X takes 2000 seconds
  - Algorithm Y takes 30 seconds

Complexity makes a huge difference!

17

Importance of analysis

- Assume:
  - Algorithm X takes time $2n^2$ (written by best programmer) running on machine with 1000 MIPS
  - Algorithm Y takes time $50n \lg n$ (written by worst programmer) running on machine with 10 MIPS



18

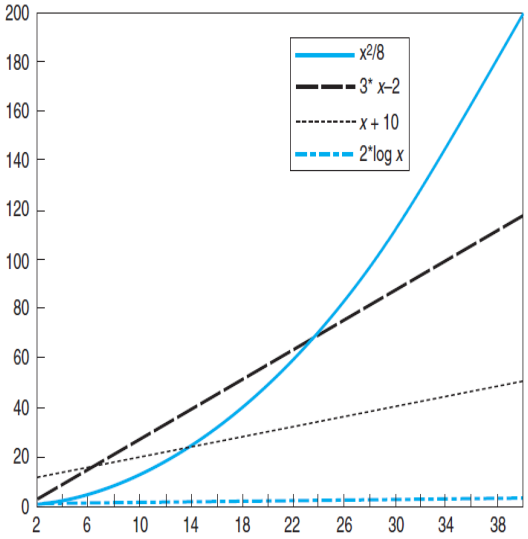
Order of growth

- The **rate of increase in operations** for an algorithm to solve a problem as **the size of the problem increases**.
- This is referred to as the rate of growth of the algorithm.

19

Order of growth

- The function based on x^2 *increases slowly at* first, but as the problem size gets larger, it begins to grow at a rapid rate.
- The functions that are based on x both grow at a steady rate for the entire length of the graph.
- The function based on log x seems to not grow at all, but this is because it is actually growing at a very slow rate.



20

Values of some important functions as $n \to \infty$

	lg *n*	*n*	*n* lg *n*	n^2	n^3	2^n
1	0.0	1.0	0.0	1.0	1.0	2.0
2	1.0	2.0	2.0	4.0	8.0	4.0
5	2.3	5.0	11.6	25.0	125.0	32.0
10	3.3	10.0	33.2	100.0	1000.0	1024.0
15	3.9	15.0	58.6	225.0	3375.0	32768.0
20	4.3	20.0	86.4	400.0	8000.0	1048576.0
30	4.9	30.0	147.2	900.0	27000.0	1073741824.0
40	5.3	40.0	212.9	1600.0	64000.0	1099511627776.0
50	5.6	50.0	282.2	2500.0	125000.0	1125899906842620.0
60	5.9	60.0	354.4	3600.0	216000.0	1152921504606850000.0
70	6.1	70.0	429.0	4900.0	343000.0	1180591620717410000000.0
80	6.3	80.0	505.8	6400.0	512000.0	1208925819614630000000000.0
90	6.5	90.0	584.3	8100.0	729000.0	1237940039285380000000000000.0
100	6.6	100.0	664.4	10000.0	1000000.0	1267650600228230000000000000000.0

21

Order of growth

- if we determine that an algorithm's complexity is a combination of two of these classes, we will frequently ignore all but the fastest growing of these terms.

- **Example** : if the complexity is $x^3 - 30x$

  we tend to **ignore $30x$ term**

22

Kinds of analysis

For some algorithms, efficiency depends on form of input:

- **Worst case**: $C_{worst}(n)$ – ***maximum time*** of algorithm over inputs of size n

- **Best case:** $C_{best}(n)$ – ***minimum time*** of algorithm over inputs of size n

- **Average case:** $C_{avg}(n)$ – ***"average"*** over inputs of size n

23

Classification of Growth

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes.

- **O(g(n)):** class of functions f(n) that grow **no faster** than g(n)
  - All functions with smaller or the same order of growth as $g(n)$

$$n \in O(n^2), \quad 100n+5 \in O(n^2), \quad 0.5n(n-1) \in O(n^2), \; n^3 \notin O(n^2)$$

- **Ω(g(n)):** class of functions f(n) that grow **at least as fast** as g(n)
  - All functions that are larger or have the same order of growth as $g(n)$

$$n^3 \in \Omega(n^2), \quad 0.5n(n-1) \in \Omega(n^2), \quad 100n+5 \notin \Omega(n^2),$$

- **Θ(g(n)):** class of functions f(n) that grow **at same rate** as g(n)
  - Set of functions that have the same order of growth as $g(n)$

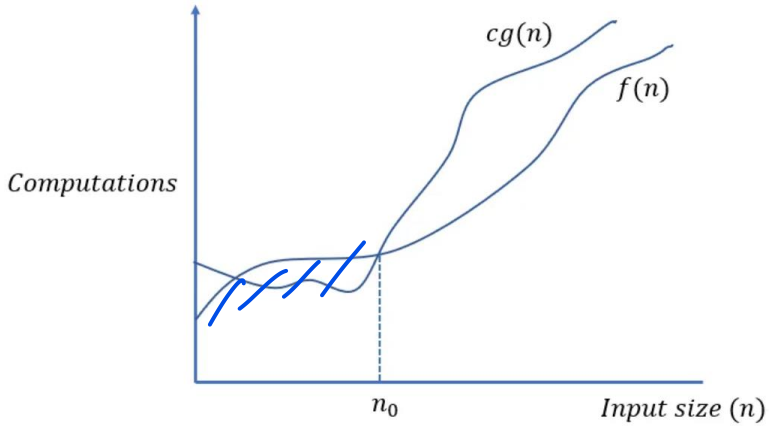
$$an^2 + bn \in \Theta(n^2)$$

24

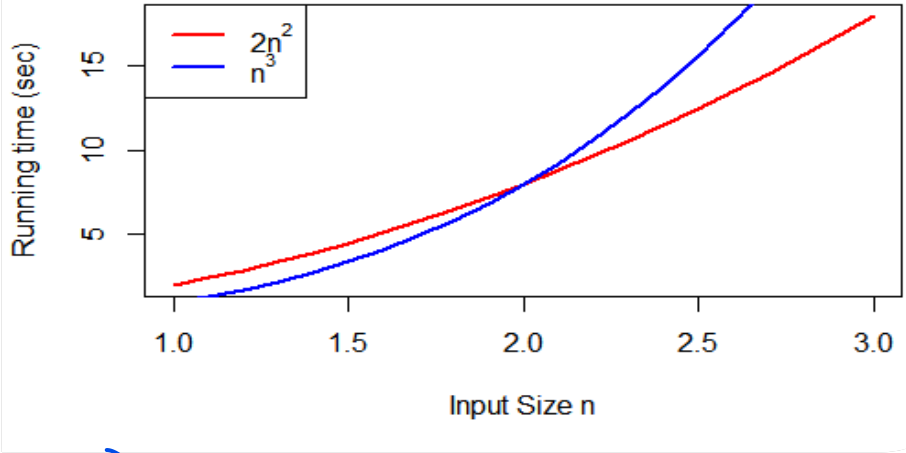
Big-oh

- O(g(n)): class of functions f(n) that grow no faster than g(n)
- if there exist some positive constant c and some nonnegative n_0 such that

$$0 \leq f(n) \leq cg(n) \quad for\ n \geq n_0$$



25



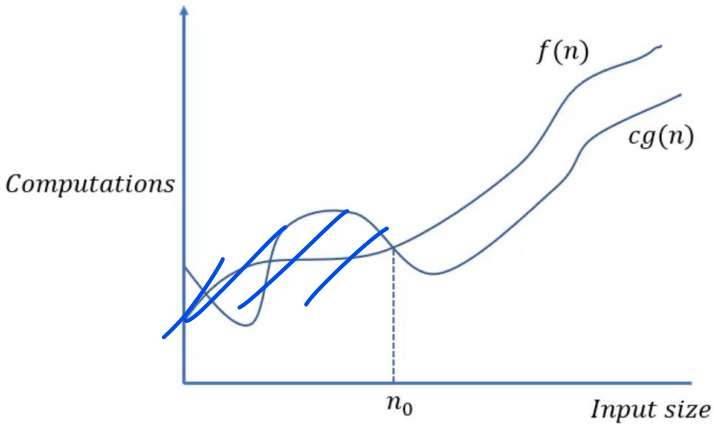
EXAMPLE: $2n^2 = O(n^3)$

26

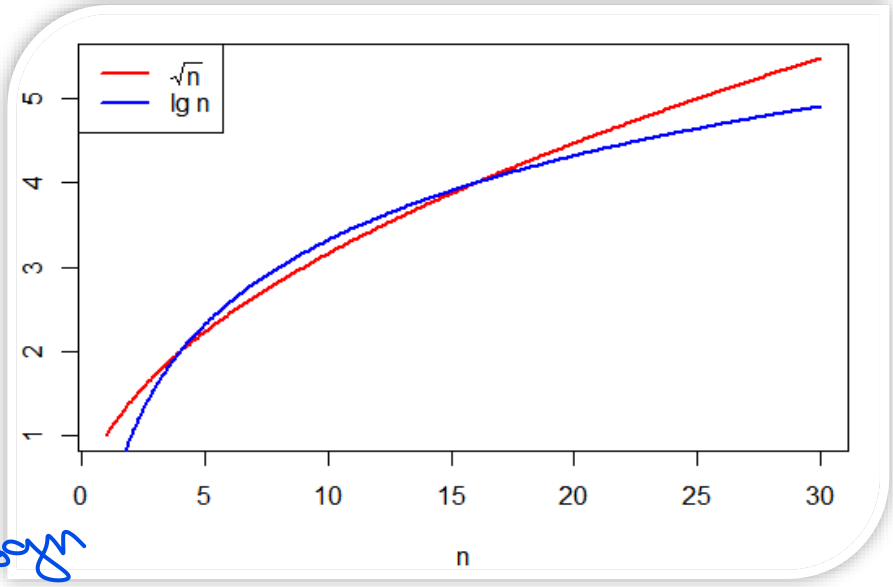
Big-omega

Ω(g(n)): class of functions f(n) that grow <u>at least as fast</u> as g(n)

$$0 \le cg(n) \le f(n) \quad for\ n \ge n_0$$



27



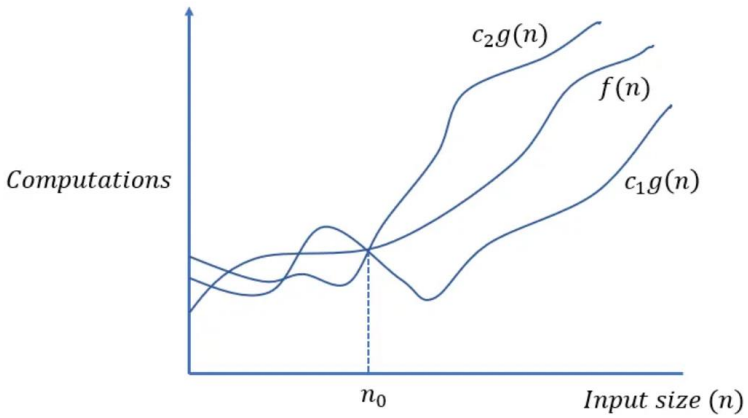
EXAMPLE: $\sqrt{n} = \Omega(\lg n)$

28

Big-theta

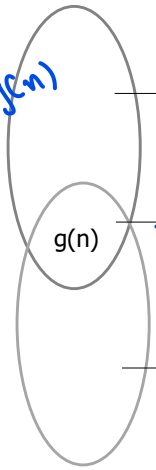
Θ(g(n)): class of functions f(n) that grow at same rate as g(n)

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \quad for\ all\ n \geq n_0$$



29

Summary



>=

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 than g(n)

30

Basic asymptotic efficiency classes

1	**constant**
log *n*	**logarithmic**
n	**linear**
n* log *n	***n*-log-*n***
n^2	**quadratic**
n^3	**cubic**
2^n	**exponential**
***n*!**	**factorial**

31

Analysis of Non recursive Algorithms

General Plan for Analysis

- Decide on parameter *n* indicating ***input size.***
- Identify algorithm's ***basic operation.***
- Determine ***worst***, ***average***, and ***best*** cases for input of size *n.*
- **Set up a sum for the number of times the basic operation is executed.**
- **Simplify the sum using standard formulas and rules.**

32

Useful summation formulas and rules

$\Sigma_{l \le i \le u} 1 = 1+1+ \cdots +1 = u - l + 1$

In particular, $\Sigma_{l \le i \le u} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n} i = 1+2+ \cdots +n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+ \cdots +n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \quad \Sigma c a_i = c \Sigma a_i \quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

33

33

Example 1: The Largest Element

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 $maxval$

$T(n) = 1 + \sum_{1}^{n-1} 1 = 1 + (n-1-1+1) = n$

T(n) = $\Sigma_{1 \le i \le n-1} (1) = n-1 =$ O(n) comparisons

34

Example 2
Element uniqueness problem

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 ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

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

$$C_{worst} = (n-1)+(n-2)+\ldots 1 = \frac{n(n-1)}{2} \quad = O(n^2)$$

35

Example 3
Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

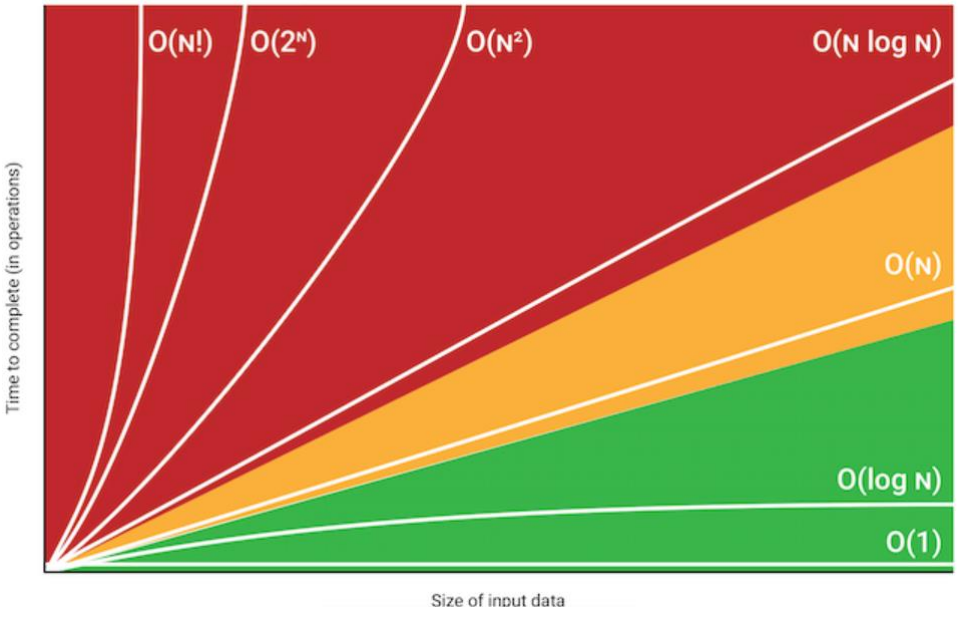
```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

T(n) = $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$ = $O(n^3)$ multiplications

Ignored addition for simplicity

36

Big-oh (summary)



37

مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology

THANK YOU!

DR YASSER ELAWADY
YELAWADY@ZEWAILCITY.EDU.EG