CS-218 DATA STRUCTURES AND ALGORITHMS

LECTURE 2

BY UROOJ AINUDDIN

IN THE LAST LECTURE…

- We learned about the algorithm.
- We learned about abstraction and its types.
- We explored the abstract data type in detail.
- We were introduced to the idea of a data structure.
- We explored the operations performed on a data structure.
- We were introduced to different kinds of data structures.

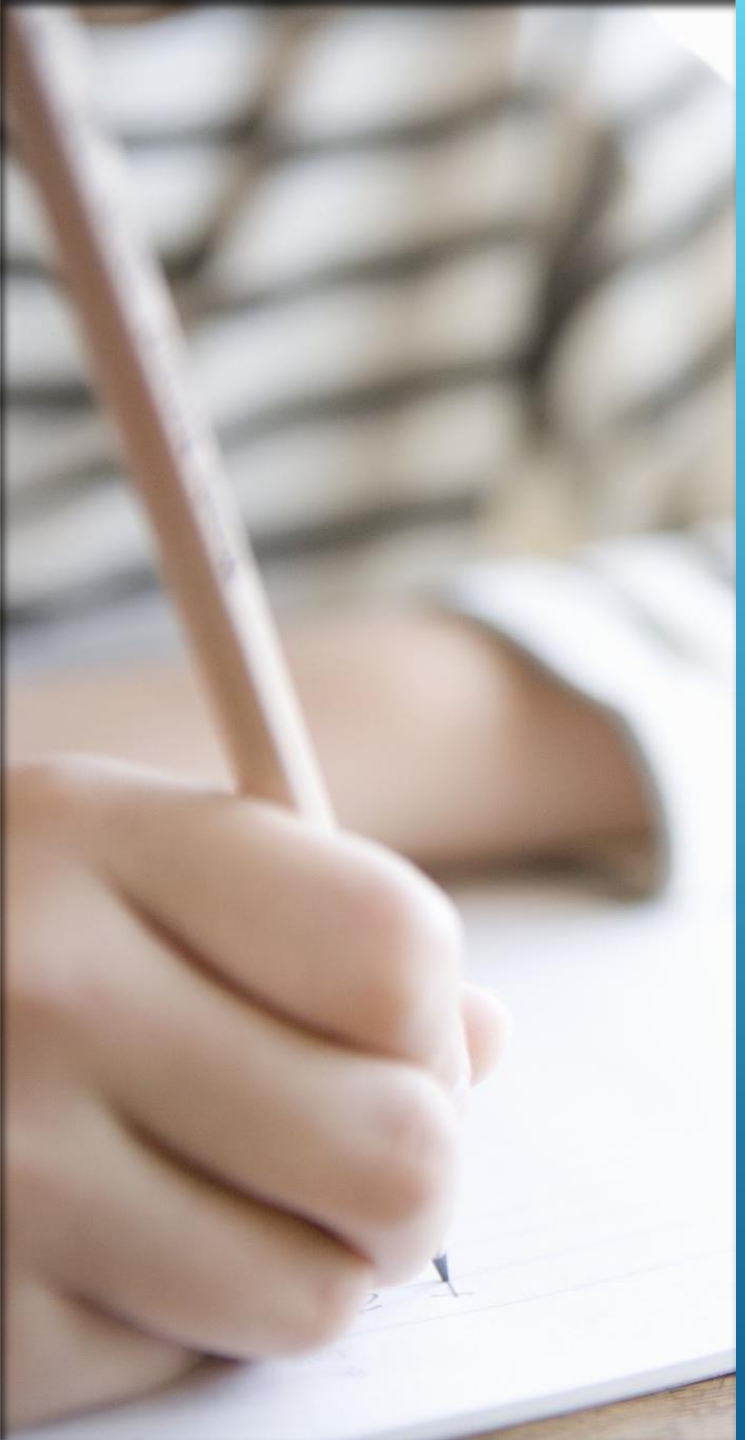
ANALYSIS OF ALGORITHMS

BOOK 1 CHAPTER 4

BOOK 2 CHAPTER 2

ALGORITHMS

- An **algorithm** is a sequence of clear and precise step-by-step instructions for solving a problem in a finite amount of time.
- An algorithm is a well-defined computational procedure that takes some values as input and produces some values as output in finite time.
- Algorithm + Data = Program



- Pseudocode is **plain language description** of the steps in an algorithm.
- Pseudocode often uses structural conventions of a normal programming language but is intended for human reading rather than machine reading. It typically omits details that are essential for machine understanding of the algorithm, such as variable declarations and language-specific code.
- The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation.
- The purpose of using pseudocode is that it is easier for people to understand than conventional programming language code.

PSEUDOCODE

WHY ANALYZE ALGORITHMS?

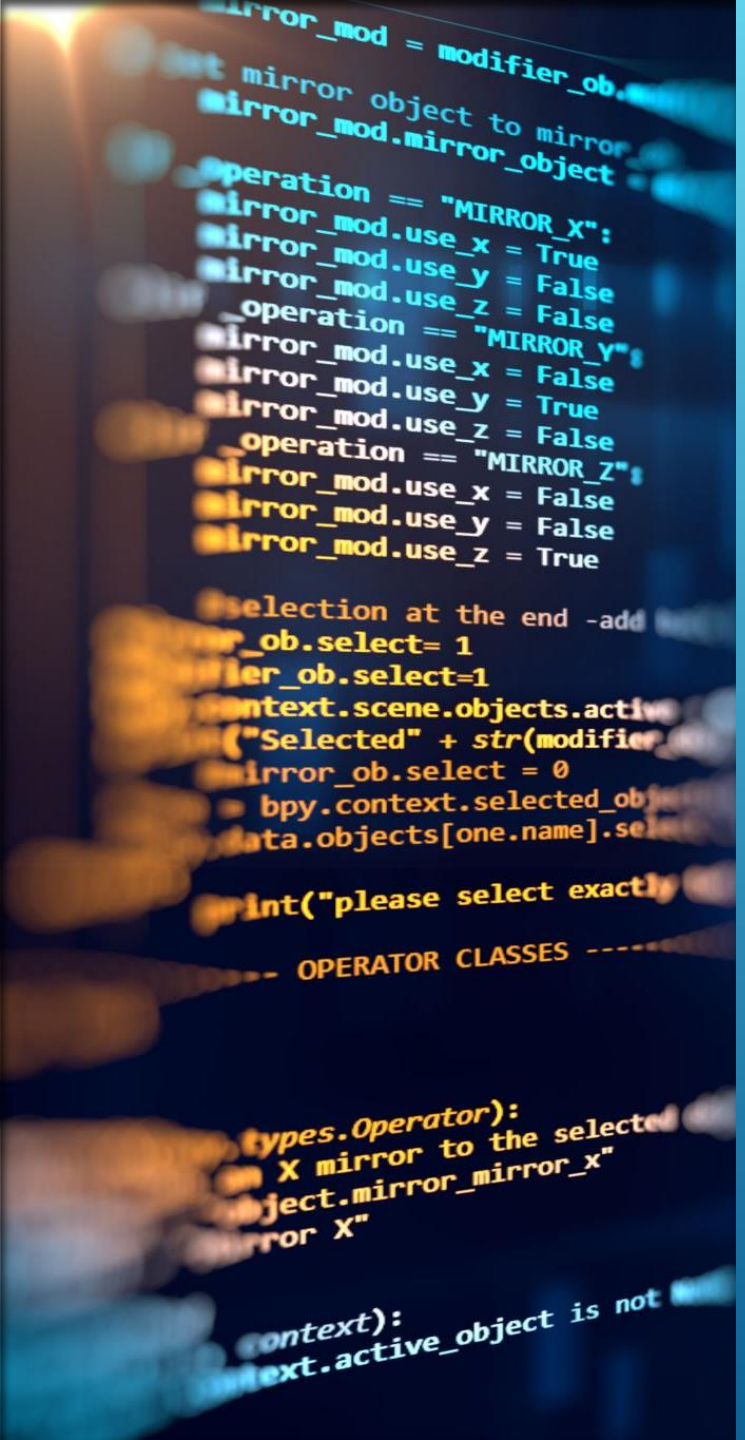
- Once an algorithm is given for a problem and decided (somehow) to be correct, an important step is to determine how much in the way of resources, such as time or space, the algorithm will require.
- An algorithm that solves a problem but requires a year is hardly of any use.
- Likewise, an algorithm that requires thousands of gigabytes of main memory is not useful on most machines.
- Analyzing an algorithm means predicting the resources it requires. These resources may be **memory**, **communication bandwidth**, **computer hardware**, and **computational time**.

HOW TO ANALYZE ALGORITHMS?

- Empirical analysis
- Theoretical analysis

EMPIRICAL ANALYSIS

- Empirical analysis is done by coding the algorithm in a high-level language and running it on a computer, over a suitable large range of inputs.
- The results of empirical analysis are highly dependent on:
  - The chosen high-level language
  - The machine on which the algorithm is run.
  - The inputs that are provided.



- You must code all algorithms, though you will finally keep only one of them, and discard the rest.
- A slow algorithm may perform better than a fast one, if it is written by a better programmer.
- Results on one set of inputs may be completely different from those for another set of inputs.
- The inputs chosen to be used may favor one algorithm more than the other.
- All algorithms under scrutiny must be run on the same software and hardware platforms.
- You may find that you do not have the budget to implement the better algorithm.

PROBLEMS WITH EMPIRICAL ANALYSIS

INPUT SIZE OR DATA SIZE

- The input size, or data size, of an algorithm is the number of data items provided to the algorithm by the calling entity.
- Input size is denoted by *n*.
- Consider the algorithm shown here.
- Two lists of length n are being added, so the input size here is *2n*.

a and *b* are lists of n elements each

```
algo(a, b):
Take a list c of n  elements
for i ← 0 to n – 1 do
        c[i] = a[i] + b[i]
return c
```

THEORETICAL ANALYSIS

- As empirical analysis is costly, and may not lead to a solution, we do not use it for algorithm analysis. Instead we use theoretical analysis.
- Theoretical analysis:
  - Uses pseudocode for the algorithm, instead of programming code.
  - Characterizes running time as a function of the input size, *n*.
  - Takes all possible inputs into account.
  - Evaluates the algorithm independent of the hardware and software.

PRIMITIVE OPERATIONS

- In theoretical analysis, we count the primitive operations that an algorithm performs for input size *n*.
- A **primitive** or **elementary operation** is defined as a high-level operation that is largely independent of the programming platform, input data and the algorithm used.
- Following is a list of primitive operations:
  - Performing an arithmetic operation (e.g. adding two numbers)
  - Comparing two numbers
  - Assigning a value to an identifier
  - Indexing into an array or following a pointer reference
  - Calling a method or function
  - Returning from a method or function

```
// Compute the sum of the list a containing n elements
Algorithm sum(a):
sum ← a[0]
for i = 1 to len(a) − 1 do
    sum ← sum + a[i]
return sum
```

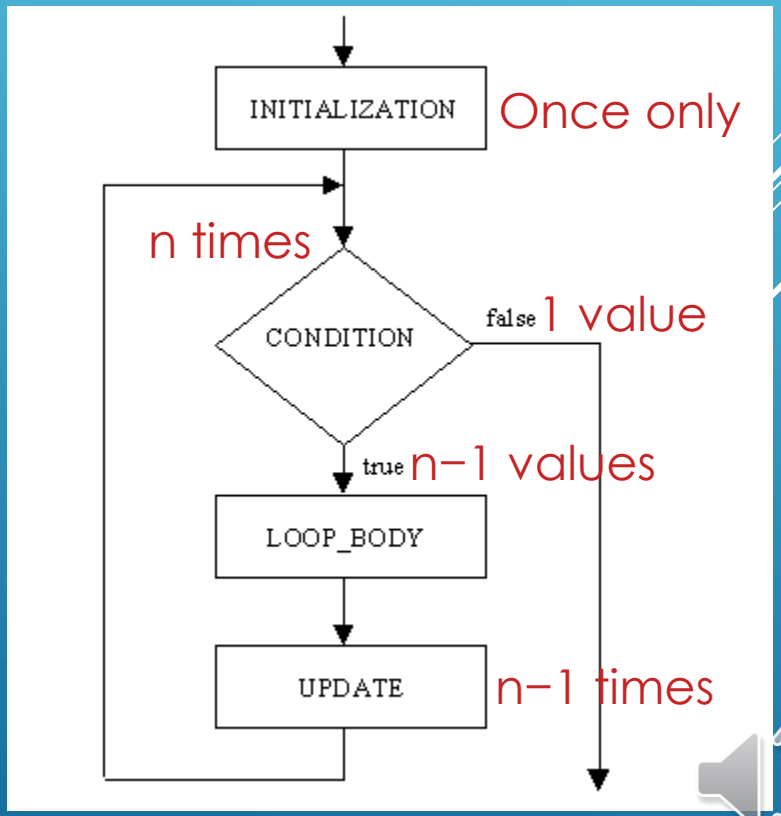
COUNT PRIMITIVE OPERATIONS FOR THE GIVEN ALGORITHM

// Compute the sum of the list *a* containing *n* elements

Algorithm sum(a):	No of operations
1. sum ← a[0]	2
2. for i = 1 to len(a) − 1 do	1 + n + n − 1 = 2n
3. sum ← sum + a[i]	3(n − 1) = 3n - 3
4. return sum	1
Total primitive operations	5n

$$T(n) = 5n$$

ANALYSIS



CASES FOR THEORETICAL ANALYSIS

- When we analyze an algorithm theoretically, we can choose to do so in three ways:
  - Best case analysis
  - Worst case analysis
  - Average case analysis

BEST CASE ANALYSIS

- Provide a set of inputs for which the algorithm takes the shortest possible time or executes the least number of primitive operations.
- The best-case running time of the algorithm is a **lower bound** on the running time of the algorithm.
- If t_i is the running time of the algorithm for a set of inputs i, and t_b is the best-case running time of the algorithm, then,

$$t_b \leq t_i$$

WORST CASE ANALYSIS

- Provide a set of inputs for which the algorithm takes the longest possible time or executes the greatest number of primitive operations.
- The worst-case running time of the algorithm is an **upper bound** on the running time of the algorithm.
- If t_i is the running time of the algorithm for a set of inputs i, and t_w is the worst-case running time of the algorithm, then,

$$t_i \leq t_w$$

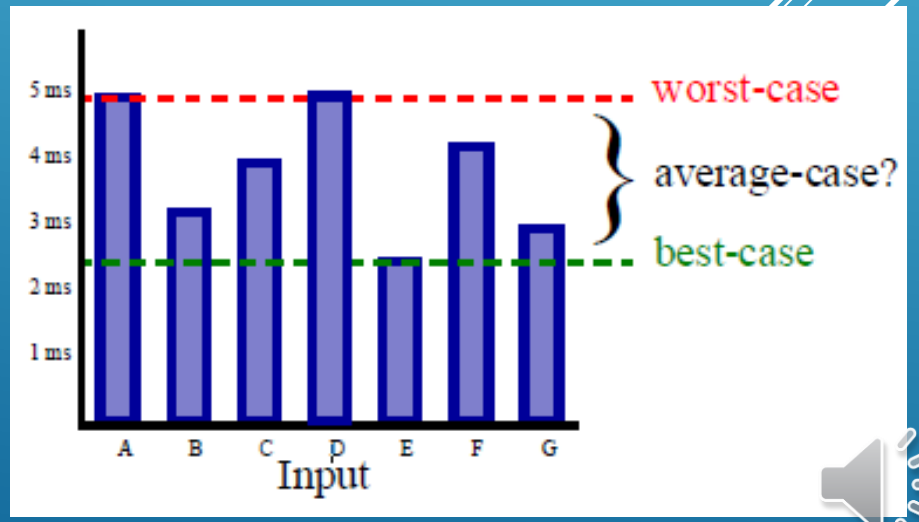
AVERAGE CASE ANALYSIS

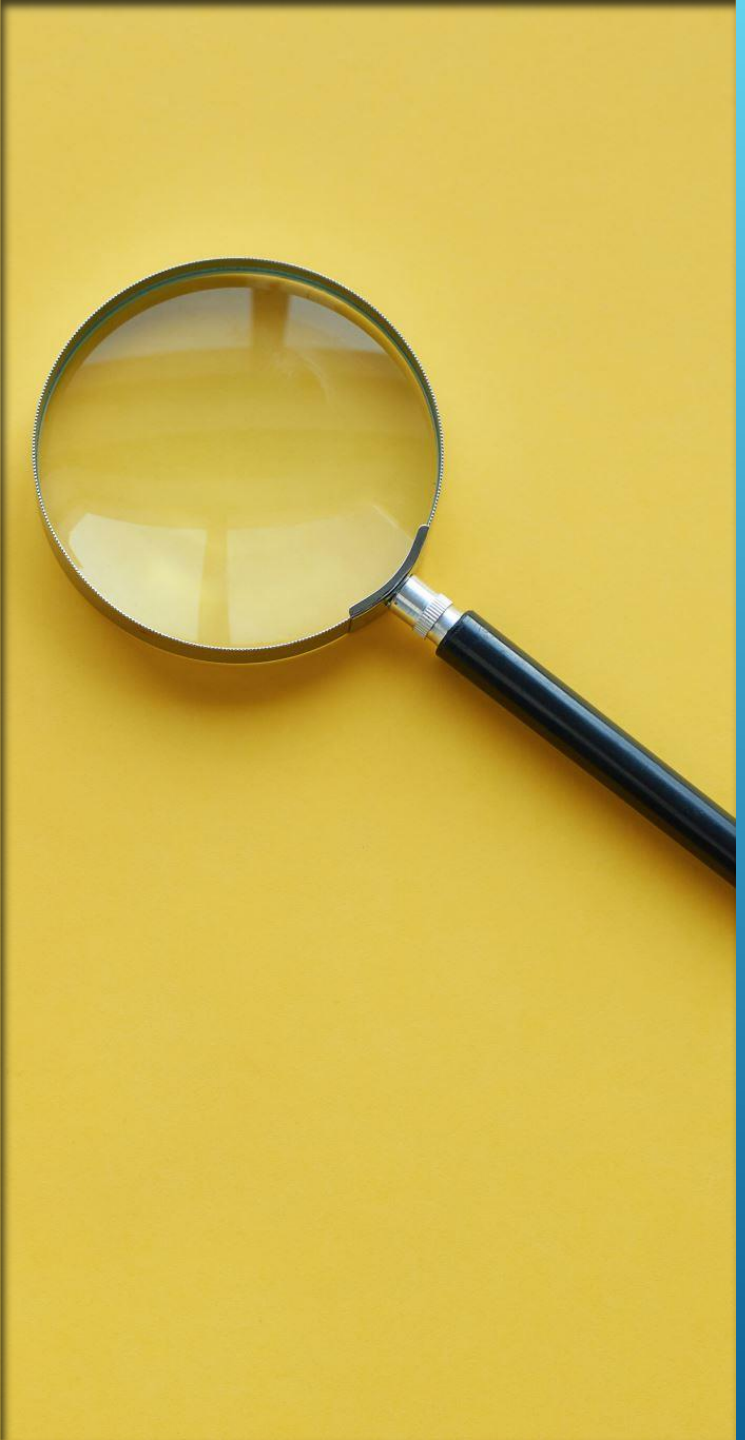
- Find the running time of the algorithm for all possible sets of inputs, find the probability of each input and calculate the average running time as shown:

$$T(n) = p_1T_1 + p_2T_2 + \cdots + p_xT_x$$

- The average running time is impossible to compute for algorithms with an infinite set of inputs.
- If t_{avg} is the average running time of the algorithm, t_b is the best-case running time of the algorithm and t_w is the worst-case running time of the algorithm, then,

$$t_b \leq t_{avg} \leq t_w$$





- If we know the worst-case running time of an algorithm, we can be sure that the algorithm will never exceed this duration for any input that is given to it.
- The worst-case analysis is most useful to a computer scientist.
- Average case analysis requires a lot of effort.
- We only conduct best case analysis when the best case occurs very frequently.

WHICH ANALYSIS IS THE MOST USEFUL?

```
Algorithm arrayMax(A, n)
//Input list A of n integers
//Output maximum element of A
currentMax ← A[0]
for i ← 1 to n − 1 do
    if A[i] > currentMax then
        currentMax ← A[i]
return currentMax
```

ANALYZE THE ALGORITHM FOR ITS BEST AND WORST CASES.

BEST CASE ANALYSIS

```
Algorithm arrayMax(A, n)
//Input list A of n integers
//Output maximum element of A
1. currentMax ← A[0]
2. for i ← 1 to n − 1 do
3.       if A[i] > currentMax then
4.             currentMax ← A[i]
5. return currentMax
```

No of operations
2
1 + n + n − 1 = 2n
2(n − 1) = 2n - 2
0
1

Total primitive operations 4n + 1

$$T(n) = 4n + 1$$

The best case happens when the **if condition is always false** i.e. when ~~A is sorted in descending order~~ ~~or~~ all elements of A are equal.

or the first element of A is the largest element.

WORST CASE ANALYSIS

Algorithm arrayMax(A, n)	
//Input list *A* of *n* integers	
//Output maximum element of *A*	No of operations
1. currentMax ← A[0]	2
2. for i ← 1 to n − 1 do	1 + n + n − 1 = 2n
3. if A[i] > currentMax then	2(n − 1) = 2n − 2
4. currentMax ← A[i]	2(n − 1) = 2n − 2
5. return currentMax	1
Total primitive operations	6n − 1

$$T(n) = 6n - 1$$

The worst case happens when the **if condition is always true** i.e. when A is sorted in ascending order.

```
// Compute the sum of the list a containing n elements
Algorithm sum(a):
sum ← a[0]
for i = 1 to len(a) − 1
    sum ← sum + a[i]
return sum
```

FIND OUT THE WORST CASE AND BEST CASE FOR THIS ALGORITHM.

TASK

Analyze the algorithm for its best and worst cases. Find equations for T(n) in both cases. Show all steps. You may do this on paper, or you may type your answer.

```
// Compute the real roots of a quadratic equation

Algorithm real_roots(a,b,c):

d ← b*b − 4*a*c

    if d >= 0

        r[0] ← (-b + sqrt(d)) / (2*a)

        r[1] ← (-b − sqrt(d)) / (2*a)

return r
```

Save your snapshot, pdf or doc as <my_roll_no>_lec2. (If your roll no. is 500, the file should be named 500_lec2.)

Submit your response in the assignment titled "Lecture 2 tasks" in Google Classroom.

SO WHAT DID WE LEARN TODAY?

- We differentiated between the algorithm and the pseudocode.
- We learned how to do empirical analysis of algorithms.
- We discussed the disadvantages of empirical analysis.
- We learned how to do theoretical analysis of algorithms.
- We were introduced to primitive operations.
- We discussed best, average and worst cases of algorithms.
- We analyzed algorithms for the first time!

THINGS TO DO

Read
- the book!

Submit
- your answer to the task in this lecture.

Note
- your questions and put them up in the relevant online session.

Email
- suggestions on content or quality of this lecture at uroojain@neduet.edu.pk