FRANKLIN'S LECTURES

ALGORITHM ANALYSIS & DESIGN

Module 1 | Part 1

CST306

SYLLABUS

- Characteristics of Algorithms, Criteria for Analysing Algorithms, Time and Space Complexity.
- Best, Worst and Average Case Complexities.
- Asymptotic Notations - Big-Oh (O), Big- Omega (Ω), Big-Theta (Θ), Little-oh (o) and Little- Omega (ω) and their properties.
- Classifying functions by their asymptotic growth rate.
- Time and Space Complexity Calculation of simple algorithms.
- Analysis of Recursive Algorithms: Recurrence Equations.

- Solving Recurrence Equations – Iteration Method, RecursionTree Method, Substitution method and Master's Theorem (Proof not required).

PRACTICAL APPLICATIONS

- Designing software that is fast, scalable, and reliable
- Optimizing systems to reduce time, memory, and cost
- Enabling efficient problem-solving in real technologies

WHAT IS AN ALGORITHM ?

- An algorithm is a well-defined, step-by-step procedure or set of rules designed to solve a specific problem or accomplish a particular task.

- Simply, an algorithm is like a recipe or a sequence of instructions that guides you through completing a task, ensuring that each step is clear and leads to the desired outcome.

- More formally, an algorithm is a sequence of unambiguous, precise instructions that can be executed by a computer to solve a problem or perform a computation.

Find the sum of 2 numbers:

⇒ + i/p → 2 num.
+ process → add them.
+ o/p → their sum.

* clear → unambiguous → step-by-step

- These instructions must be clear enough to be implemented as a program, meaning they can be translated into code that a computer can run.

- Algorithms help turn problems into a list of steps that programmers use to write computer code.

- This code is what computers read and follow to do tasks

- When the program runs, it takes information (input), works on it using the algorithm, and then gives an answer (output).

Alg → Code → Sln.

- This process—input, process, output—shows how computers use algorithms to solve problems every day.

IPO

- i/p → what data we receive?
- Process → what steps we apply?
- o/p → what answers we produce?

i/p : {10, 5, 3}

Process : compare & arrange

o/p : {3, 5, 10}

CHARACTERISTICS OF AN ALGORITHM

- Every algorithm must satisfy the following criteria..
  - **Input**: Algorithm must accept zero or more input values from external sources.
  - **Output**: Algorithm must produce a result as output.
  - **Definiteness**: Each instruction must be clear and unambiguous, allowing only one interpretation.
  - **Finiteness**: Algorithm must complete and produce a result within a finite number of steps for all cases.
  - **Effectiveness**: Each instruction must be basic, feasible, and executable.

Example:

- **Problem Description**: Determine the maximum number from a provided list.
- **Input**: A collection of positive integers, with the list containing at least one element.
- **Output**: The greatest number found in the input list. Let the input list as 'L' and the output—the largest number—as 'max'.

Step 1: Initialize a variable named 'max' with the value 0.

Step 2: Examine the first element in list 'L', call it 'x'. If 'x' exceeds the current 'max', update 'max' to 'x'.

Step 3: Continue this comparison for every number in 'L', updating 'max' whenever a larger number is found.

Step 4: After processing all elements, present the value stored in 'max' as the final result.

I/P = {3, 17, 6, 9} => L

O/P = max = 17

Procen :- max = 0 | max = 3 | max = 17

x = 3 ⇒ x > max

x = 17 ⇒ x > max

x = 6 ⇒ x < max

x = 9 ⇒ x < max

ANALYSIS OF AN ALGORITHM

- Imagine you want to travel from city "A" to city "B". There are many ways to make this journey.
- You could choose to travel by flight, bus, train, or even by bicycle.
- Your choice depends on factors like availability, cost, speed, and convenience.
- Similarly, in computer science, there are often multiple algorithms available to solve the same problem.

- Each algorithm may differ in how fast it runs, how much memory it uses, or how complex it is to implement.

- When faced with several algorithms, it's important to select the one that best fits the specific needs of the problem and the resources available.

- Performance analysis is the process of evaluating and comparing these algorithms based on criteria such as **time complexity** and **space complexity**.

SPACE COMPLEXITY

- When we design an algorithm to solve a problem, it requires a certain amount of computer memory to run properly. This memory is needed for several important reasons:

  - **Storing the program's instructions**: The algorithm's code must be kept in memory so the computer knows what steps to follow.
  - **Keeping constant values**: These are fixed values that do not change during the execution but are necessary for calculations or decisions.
  - **Saving variable values**: Variables hold data that can change as the algorithm runs, such as counters or temporary results.

- **Managing function calls and control flow:** Memory is also used to keep track of function calls, jumps, and other control structures that guide the algorithm's execution.

- The term "space complexity" refers to the total amount of memory an algorithm needs from start to finish. It helps us understand how efficiently an algorithm uses memory, which is important when working with limited resources or large data sets.

"how much memory an algorithm needs"

- **Space needed by each algorithm is the sum of following components:**
  - **Fixed space requirement**

    It include instruction space(space for storing code), space for simple variables, constants etc.
  - **Variable space requirement**

    Space for variables whose size vary with change in i/p + space for referenced variables + recursion stack space

- **C is a constant representing the fixed space requirement and $S_P(I)$ is the variable space requirement of a program P working on an instance I**
  - Total space requirements S(P) of any program can be expressed as:

$$S(P) = C + S_P(I)$$

count

Fixed space.

TIME COMPLEXITY

- Time complexity measures the computer time an algorithm needs to execute.
- It represents the total time required for an algorithm to complete its task.
- Time complexity quantifies the execution duration of an algorithm.
- Time complexity measures how an algorithm's running time depends on input size.
- Running time can be influenced by hardware factors like processor type (single vs. multi), bit architecture (32-bit vs. 64-bit), and read/write speeds.

- Algorithm operations include arithmetic, logical, return value, and assignment operations.
- When calculating time complexity, only input data size is considered; hardware differences are ignored.
- The focus is on how the algorithm behaves with varying input sizes, regardless of machine specifics.
- So basically, **Time complexity of an algorithm measures how much time an algorithm requires to run the program completely.**
- The total time T(P) taken by a program P is the sum of its compile time and its run time.
  - T= T(compilation) + T(execution)
  - T ≈ T(execution)

- Time complexity count the no. of instruction steps performed by the algorithm.
- If the time complexity is lower, it means the algorithm is faster.

METHODS TO CALCULATE TIME COMPLEXITY

- **Frequency Count (step count):** the no. of operations/steps

  - To find Time complexity of a given program- summing the frequency counts of all statements of that program.
  - Every simple statements in an algorithm takes one unit of time.

- **Rules:**
  - Comments & Declarations has frequency count = 0
  - Return & assignment statement has frequency count = 1
  - Ignore low order exponent when higher order exponents are present.
  - For looping statements- no. of times the loop is repeating

$2n^3 + 3n + 4$

BEST, WORST, AVERAGE CASE COMPLEXITIES

- **Best Case**
  - The scenario where the algorithm performs the minimum number of operations.
  - Example (Linear Search):
    - If the key is found at the first position, only 1 comparison is required.
    - Best-case complexity: Ω(1).

- **Worst Case**
  - The scenario where the algorithm performs the maximum number of operations.
  - Example):
    - Key not present in an unsorted array → must check all n elements.
    - Worst-case complexity: O(n).
    - Worst-case analysis is most important in:
      - Real-time systems
      - Safety-critical systems
      - Performance guarantees

- **Average Case**
  - Represents the expected number of operations for a random input.
  - Example (Linear Search): Assuming key occurs uniformly:

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

    - Average case typically reflects real-world performance.

ASYMPTOTIC NOTATIONS

- Asymptotic analysis describes how the running time of an algorithm grows with the size of the input (n).
- Instead of exact time, we compare growth rates, which is more
- reliable and hardware-independent.
- We use mathematical notations to express these bounds.

BIG-OH NOTATION — O(F(N))

- Big-Oh represents an upper bound on the running time.

If

$$T(n) \leq C \cdot f(n)$$

for sufficiently large n,

then

$$T(n) = O(f(n)).$$

$3n+5 \Rightarrow O(n)$

- Meaning:
  - The algorithm will not grow faster than f(n).
  - Example:

    If

    $T(n)=3n+10$,

    then

    $T(n)=O(n)$.

BIG-OMEGA NOTATION — Ω(F(N))

- Big-Omega represents a lower bound.

  If

  $$T(n) \geq C \cdot f(n)$$

  for large n,

  then

  $$T(n) = \Omega(f(n)).$$

- Meaning:
  - The algorithm takes at least f(n) time.
  - Example:

    $$3n+10 = \Omega(n)$$

Linear Search - Ω(1)

BIG-THETA NOTATION — Θ(F(N))

- Theta gives a tight bound — both upper and lower.

  If

  $$T(n)=O(f(n)) \text{ and } T(n)=\Omega(f(n)),$$

  then

  $$T(n)=\Theta(f(n)).$$

- Example:

  $$3n+10=\Theta(n)$$

⇒ "if O & Ω meets ⇒ Θ"

BIG-THETA NOTATION — Θ(F(N))

- Theta gives a tight bound — both upper and lower.

  If

  $$T(n)=O(f(n)) \text{ and } T(n)=\Omega(f(n)),$$

  then

  $$T(n)=\Theta(f(n)).$$

- Example:

  $$3n+10=\Theta(n)$$

LITTLE-OH AND LITTLE-OMEGA NOTATIONS

- These notations express strict upper and lower bounds.

  Little-oh Notation — o(f(n))

  Little-oh represents a strictly smaller upper bound.

$$T(n) = o(f(n)) \text{ if } \lim_{n \to \infty} \frac{T(n)}{f(n)} = 0$$

  Meaning:

  T(n) grows strictly slower than f(n).

Example:

n=o(nlogn)

because $$\frac{n}{n \log n} = \frac{1}{\log n} \to 0$$

2 runners

f(n) – Slow runner

g(n) – fast runner

LITTLE-OMEGA NOTATION — ω (F(N))

Little-omega represents a strictly larger lower bound.

$$T(n) = \omega(f(n)) \text{ if } \lim_{n \to \infty} \frac{I(n)}{f(n)} = \infty$$

Meaning:

T(n) grows strictly faster than f(n).

f(n): very fast runner
g(n): slow runner.

Example:

nlogn=ω(n)

PROPERTIES OF ASYMPTOTIC NOTATIONS

- Reflexivity
- Symmetry
- Transpose Symmetry
- Transitivity

Reflexivity

~~PROPERTIES OF ASYMPTOTIC NOTATIONS~~

Statements

- f(n)=O(f(n)) → f(n) does not grow & lower than its of
- f(n)=Ω(f(n)) → " " " " " " "
- f(n)=Θ(f(n)) → " " " " " " "

x min y kms

x = x-1

SYMMETRY

- $f(n)=\Theta(g(n))$ iff $g(n)=\Theta(f(n))$

if f & g grow at the same speed, it works both ways.

=> Same growth goes in different directions.

TRANSPOSE SYMMETRY

Statements

- f(n)=O(g(n))iff g(n)=Ω(f(n))
- f(n)=o(g(n))iff g(n)=ω(f(n))

O & Ω are opposites

* if 'f' is smaller, g is bigger
* if 'f' is bigger, g is smaller.

TRANSITIVITY

Statements

- f(n) = O(g(n)) and g(n) = O(h(n)) f(n) = O(h(n))

If A is smaller then B, & B is smaller then C then A is smaller then C.

$n = O(n^2)$

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

$n = O(n^3)$

Complexity Functions

- Constant Time — O(1)
- Logarithmic Time — O(log n)
- Linear Time — O(n)
- Quadratic Time — $O(n^2)$
- Polynomial Time — $O(n^k)$
- Exponential Time — $O(2^n)$
- Factorial Time — O(n!)

Order of Growth (Smallest → Largest)

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$$

FRANKLIN'S LECTURES

THANK YOU