Algorithms And Data Structures

Kapilraj Trivedi

Training outline

Module 1

Introduction

Module 2

DATA STRUCTURES

Module 3

COMPLEXITY ANALYSIS

Module 4

SORTING ALGORITHMS

Module 5

NON-LINEAR DATA STRUCTURES

Module 6

ABSTRACT DATA TYPES

Module 7

BINARY SEARCH

Module 8

DIJKSTRA'S ALGORITHM

Module 1: Introduction to Algorithms

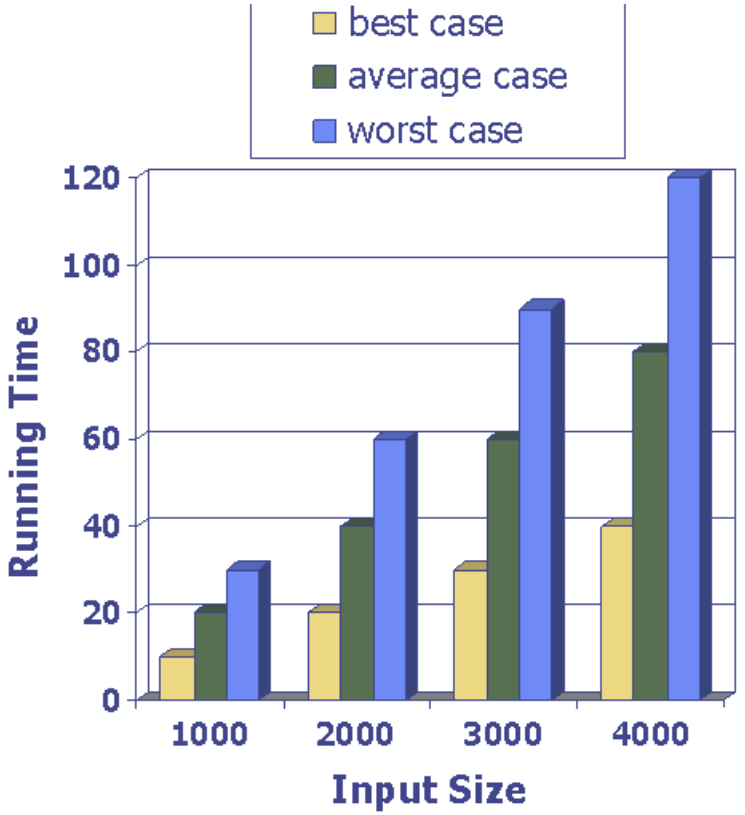
- Algorithms are precise sequences of steps or instructions designed to solve specific problems or perform tasks.
- They are fundamental to computer science and are used in various applications, from simple calculations to complex data processing.
- **Importance of Algorithms**:
  - Algorithms form the backbone of computer programs and systems.
  - Understanding algorithms is crucial for developing efficient software solutions, optimizing processes, and solving problems effectively.Prototyping Model
- **Real-life Examples**: Algorithms are prevalent in everyday life. From following a recipe to navigate through traffic, we often rely on algorithms without even realizing it.

Time Complexity

- Time complexity measures the amount of time an algorithm takes to execute relative to the size of its input. It indicates how the algorithm's runtime grows as the input size increases.
- *Example*: A sorting algorithm may have a time complexity of O(n log n), meaning its runtime increases logarithmically with the size of the input list.

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst-case running time.
- Easier to analyze
- Crucial to applications such as games, finance and robotics

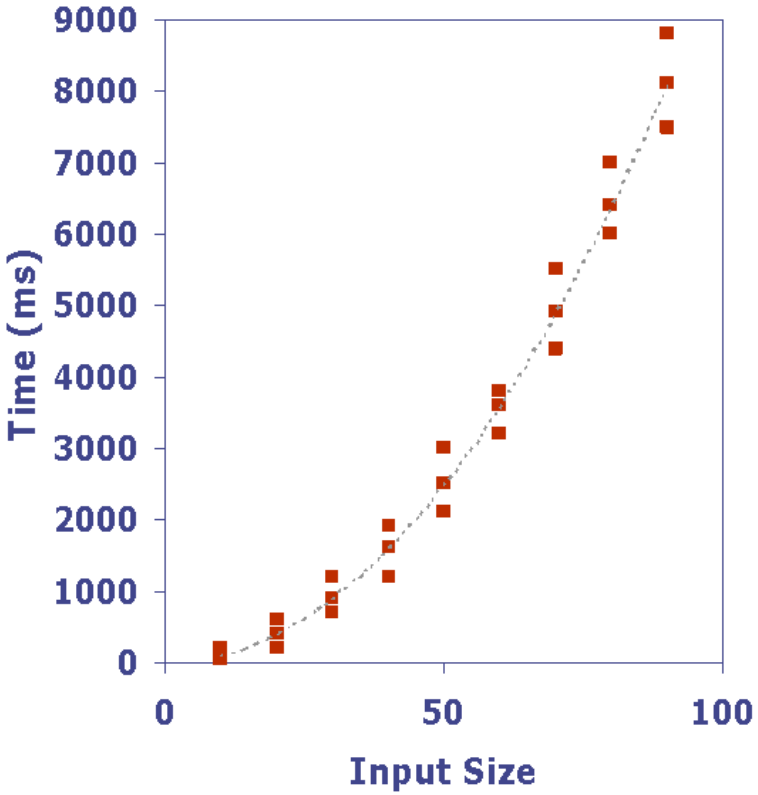


Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results
- In python - System.currentTimeMillis() can be done as follows:

```
import time

# Get current time in milliseconds
current_milliseconds = int(time.time() * 1000)
```



Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm.
- More structured than English prose.
- Less detailed than a program.
- Preferred notation for describing algorithms.
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax(A, n)
Input array 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
```

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model
- **Examples:**
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- Let us look at 2 different examples and try to calculate the time complexity

Estimating Running Time

- Algorithm arrayMax executes 8n - 2 primitive operations in the worst case. Define:
  a = Time taken by the fastest primitive operation
  b = Time taken by the slowest primitive operation
- Let T(n) be worst-case time of arrayMax. Then a (8n - 2) <= T(n) <= b(8n - 2)
- Hence, the running time T(n) is bounded by two linear functions

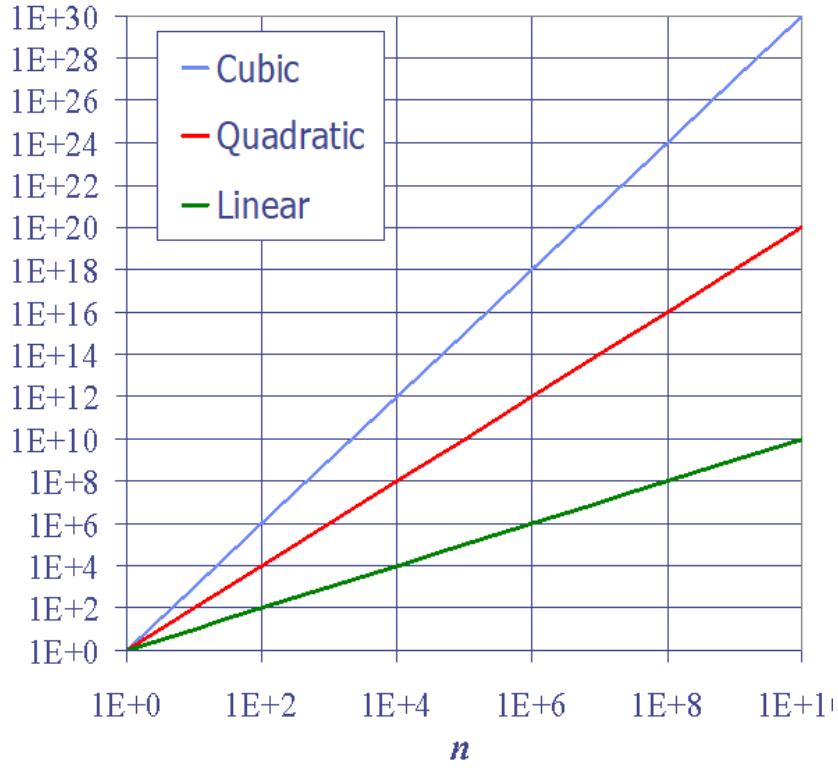
Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects T(n) by a constant factor,
  - but Does not alter the growth rate of T(n)
- The linear growth rate of the running time T(n) is an intrinsic property of algorithm arrayMax

Seven Important Functions

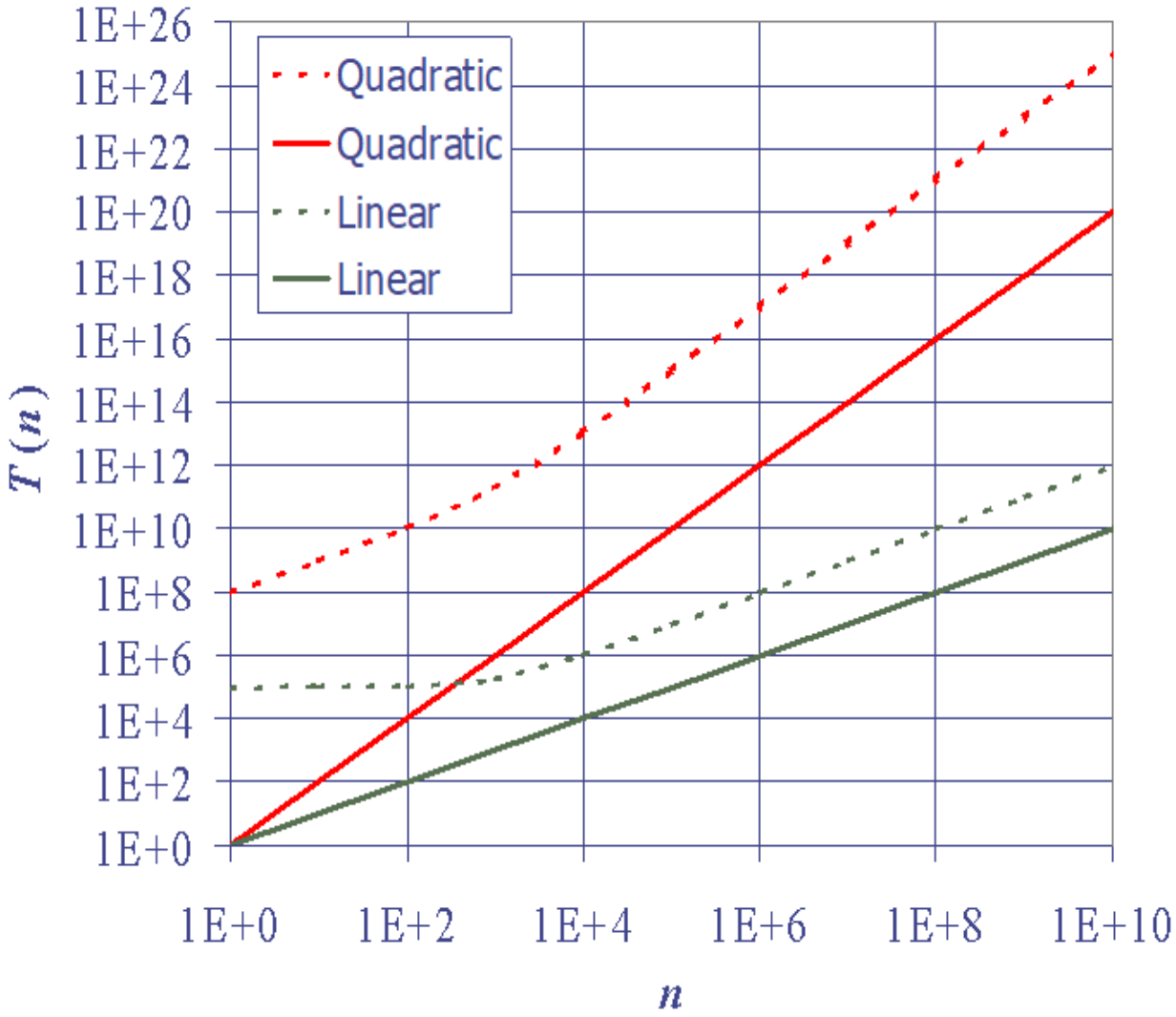
- Seven functions that often appear in algorithm analysis:
  - Constant = 1
  - Logarithmic = log n
  - Linear = n
  - N-Log-N = n log n
  - Quadratic = n^2
  - Cubic = n^3
  - Exponential = 2^n

- In a log-log chart, the slope of the line corresponds to the growth rate of the function



Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- **Examples:**
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function

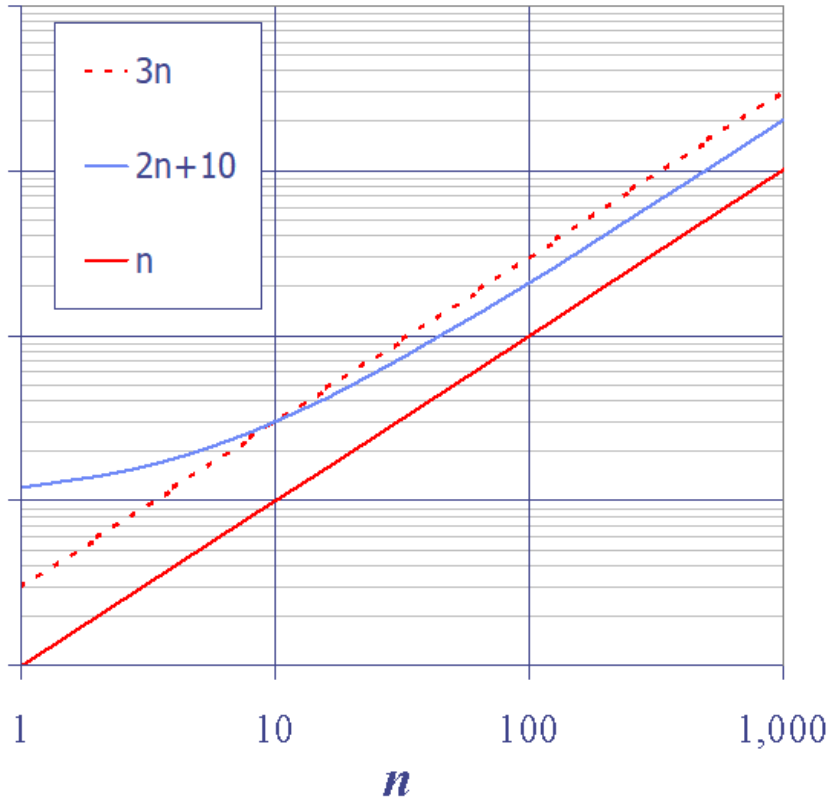


Big-Oh Notation

- Given functions **f(n)** and **g(n)**, we say that **f(n)** is **O(g(n))** if there are positive constants c and n_0 such that

  f(n) <= cg(n) for n <= n_0

- Example: 2n + 10 is O(n)
  - 2n + 10 <= cn
  - (c-2) n >= 10
  - n >=10/(c - 2)
  - Pick c = 3 and n_0 = 10



Big-Oh Examples

- **$7n-2$**
  - $7n-2$ is $O(n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$
  - this is true for $c = 7$ and $n_0 = 1$
- **$3n^3 + 20n^2 + 5$**
  - $3n^3 + 20n^2 + 5$ is $O(n^3)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
  - this is true for $c = 4$ and $n_0 = 21$
- **$3 \log n + 5$**
  - $3 \log n + 5$ is $O(\log n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
  - this is true for $c = 8$ and $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement “f(n) is O(g(n))” means that the growth rate of f(n) is no more than the growth rate of g(n).
- We can use the big-Oh notation to rank functions according to their growth rate.

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

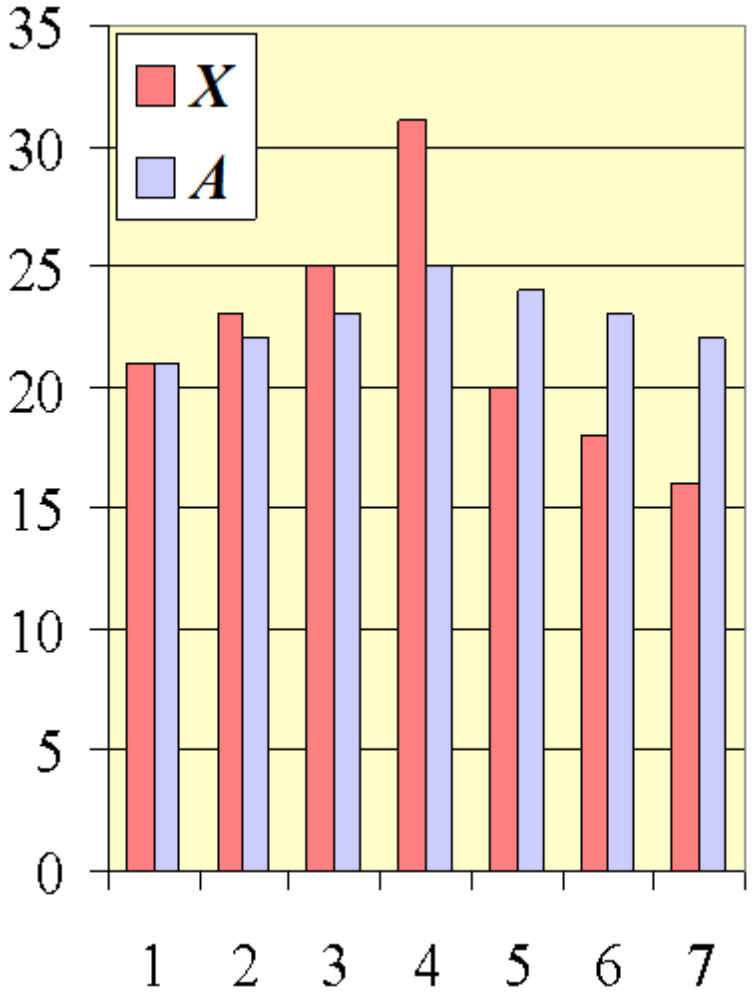
Big-Oh Rules

- If is f(n) a polynomial of degree d, then f(n) is O(nd), i.e.,
  - Drop lower-order terms
  - Drop constant factors
- Use the smallest possible class of functions
  - Say “2n is O(n)” instead of “2n is O(n2)”
- Use the simplest expression of the class
  - Say “3n + 5 is O(n)” instead of “3n + 5 is O(3n)”

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first (i + 1) elements of X:
- A[i] = (X[0] + X[1] + … + X[i])/(i+1)

- Computing the array A of prefix averages of another array X has applications to financial analysis



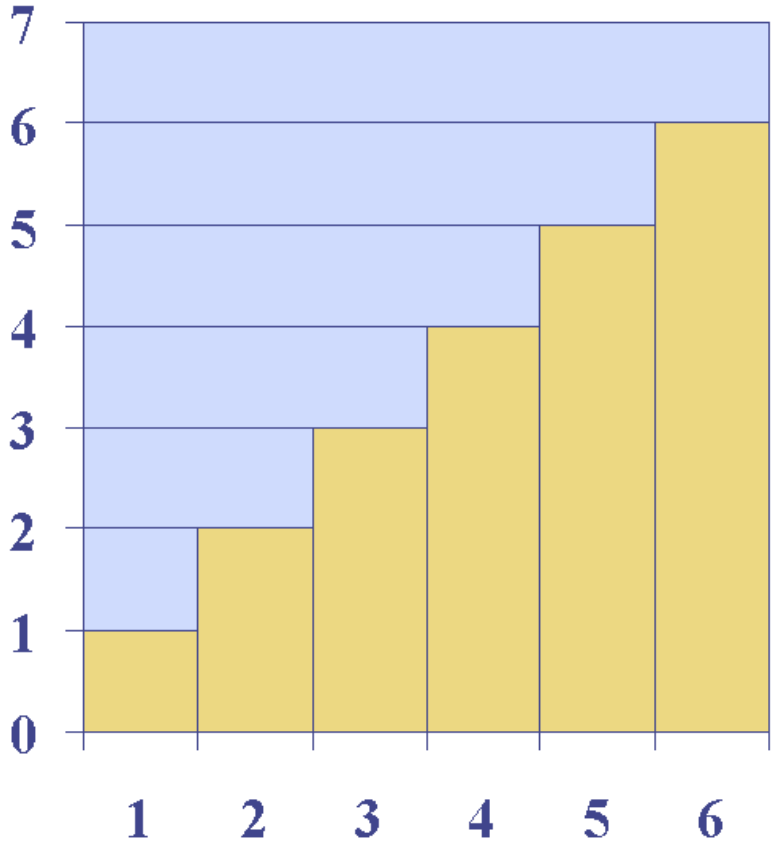
Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

```
Algorithm prefixAverages1(X, n)                          #operations
    Input array X of n integers
    Output array A of prefix averages of X
    A ← new array of n integers                          n
    for i ← 0 to n − 1 do                                n
        s ← X[0]                                         n
        for j ← 1 to i do                                1 + 2 + …+ (n − 1)
            s ← s + X[j]                                 1 + 2 + …+ (n − 1)
        A[i] ← s / (i + 1)                               n
    return A                                             1
```

Arithmetic Progression

- The running time of prefixAverages1 is O(1 + 2 + …+ n)
- The sum of the first n integers is n(n + 1) / 2
  - There is a simple visual proof of this fact
- Thus, algorithm prefixAverages1 runs in $O(n^2)$ time



Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm arrayMax executes at most 8n - 2 primitive operations
  - We say that algorithm arrayMax "runs in O(n) time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum
- Algorithm prefixAverages2 runs in O(n) time

```
Algorithm prefixAverages2(X, n)                          #operations
    Input array X of n integers
    Output array A of prefix averages of X
    A ← new array of n integers                          n
    s ← 0                                                1
    for i ← 0 to n − 1 do                                n
        s ← s + X[i]                                     n
        A[i] ← s / (i + 1)                               n
    return A                                             1
```

Math you need to Review

- Summations
- Logarithms and Exponents
- Proof techniques
- Basic probability

- **properties of logarithms:**
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b xa = a\log_b x$
  $\log_b a = \log_x a/\log_x b$
- **properties of exponentials:**
  $a^{(b+c)} = a^b a^c$
  $a^{bc} = (a^b)^c$
  $a^b / a^c = a^{(b-c)}$
  $b = a^{\log_a b}$
  $b^c = a^{c*\log_a b}$

Relatives of Big-Oh

- **big-Omega** Ω
  - **Definition**: Omega notation represents the lower bound of an algorithm's time or space complexity. It describes the best-case scenario in terms of performance.
  - f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant n_0 >= 1 such that f(n) >= c•g(n) for n >= n_0
  - **Calculation:**
    - **Identify Dominant Term**: Determine the term in the algorithm's time or space complexity expression that grows the slowest with input size.
    - **Ignore Higher Order Terms**: Disregard any constant factors and higher-order terms.
    - **Drop Constant Factors**: Keep only the term with the lowest exponent and drop constant factors.

- **big-Theta** Θ
  - **Definition**: Theta notation represents both the upper and lower bounds of an algorithm's time or space complexity. It describes the tightest bound on the algorithm's performance.
  - f(n) is Θ(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant n0 □ 1 such that c'•g(n) □ f(n) □ c''•g(n) for n □ n0

- **big-Theta** Θ
  - **Definition**: Theta notation represents both the upper and lower bounds of an algorithm's time or space complexity. It describes the tightest bound on the algorithm's performance.
  - **Calculation**:
    - **Identify Dominant Term**: Determine the term in the algorithm's time or space complexity expression that grows at the same rate for both best and worst-case scenarios.
    - **Drop Constant Factors**: Keep only the term with the highest or lowest exponent and drop constant factors

  • f(n) is Θ(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant n_0 >= 1 such that c'•g(n) <= f(n) >= c''•g(n) for n >= n_0

Intuition for Asymptotic Notation

- Big-Oh
  - f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)
- Big-Omega
  - f(n) is □(g(n)) if f(n) is asymptotically greater than or equal to g(n)
- Big-Theta
  - f(n) is □(g(n)) if f(n) is asymptotically equal to g(n)

Example Uses of the Relatives of Big-Oh

- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \bullet g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \bullet g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

Space Complexity

•Space complexity refers to the amount of memory space required by an algorithm to solve a problem, typically in terms of auxiliary space or extra space used by the algorithm in addition to the input data.
•It helps analyze how much memory an algorithm consumes relative to the size of the input.
•Space complexity is essential for evaluating the efficiency of algorithms, especially when dealing with large datasets or limited memory resources.

Types of Space Complexity

- **Auxiliary Space Complexity**: This refers to the space required by an algorithm, excluding input space. It includes variables, data structures, and any additional memory used during the execution of the algorithm but not directly related to the input size.
- **Total Space Complexity**: This encompasses both the input space and auxiliary space. It represents the total memory consumption by the algorithm, considering both input data and any additional memory used

How to Calculate Space Complexity

- **Identify Variables and Data Structures**: Identify all variables and data structures used by the algorithm, including arrays, lists, sets, maps, etc.
- **Analyze Memory Consumption**: Determine the memory consumption of each variable and data structure based on their data types and sizes. For example, integers typically consume a fixed amount of memory, while dynamic data structures like lists may vary in size based on the input.
- **Ignore Constants**: Ignore constants and focus on dominant terms. Constants represent fixed overhead and do not change with input size, so they are often omitted from space complexity analysis.
- **Summarize Space Consumption**: Summarize the space consumption of all variables and data structures to derive the overall space complexity of the algorithm.

- Let us look at some examples and understand how to calculate space complexity

Thank you