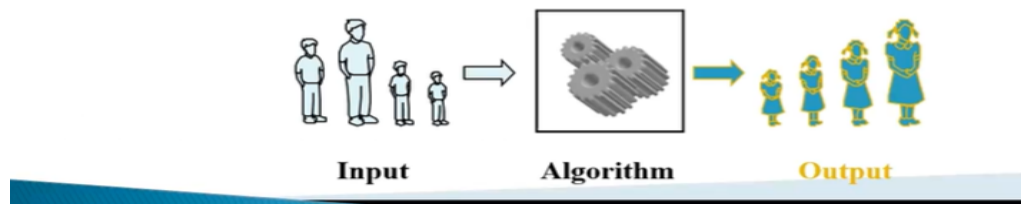


Module 3 – Algorithm Analysis

- Time Complexity and Space Complexity
- Review and use of Big-O Notation
- Recurrence Relations and Simple Solutions
- Use of Divide-and-conquer in designing operation on data Structure

Analysis of Algorithms

An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.



Good Algorithms?

- ▶ Run in less time
- ▶ Consume less memory


But computational resources (time complexity) is usually more important

Analysis of Algorithms

- ▶ Efficiency measure
 - how long the program runs **time complexity**
 - how much memory it uses **space complexity**
- ▶ Why analyze at all?
 - Decide what algorithm to implement before actually doing it
 - Given code, get a sense for where bottlenecks must be, without actually measuring it

Time Complexity:

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

- ▶ The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.
 - ▶ It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.
- 

Example 1: Addition of two scalar variables.

Algorithm ADD SCALAR(A, B)

//Description: Perform arithmetic addition of two numbers

//Input: Two scalar variables A and B

//Output: variable C, which holds the addition of A and B

$C \leftarrow A + B$

return C

The of two scalar numbers requires one addition operation. the time complexity of this algorithm is constant, so $T(n) = O(1)$.

- ▶ The efficiency of an algorithm is a measure of the amount of resources consumed in solving a problem of size n .
 - The resource we are most interested in is time
 - We can use the same techniques to analyze the consumption of other resources, such as memory space.
- ▶ It would seem that the most obvious way to measure the efficiency of an algorithm is to run it and measure how much processor time is needed

RUNNING TIME OF AN ALGORITHM

- ▶ Depends upon

- Input Size
- Nature of Input

- ▶ Generally time grows with size of input, so running time of an algorithm is usually measured as function of input size.

- ▶ Running time is measured in terms of number of steps/primitive operations performed

- ▶ Independent from machine, OS

Finding running time of an Algorithm / Analyzing an Algorithm

- ▶ Running time is measured by number of steps/primitive operations performed
- ▶ Steps means elementary operation like
 - $, +, *, <, =, A[i]$ etc
- ▶ We will measure number of steps taken in term of size of input

Space Complexity:

- ▶ Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.
- ▶ The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input.

COMPARING FUNCTIONS

- Big Oh Notation: Upper bound
 - Omega Notation: Lower bound
 - Theta Notation: Tighter bound
- Big Oh*

Big Oh Notation [1]

If $f(N)$ and $g(N)$ are two complexity functions, we say

$$f(N) = O(g(N))$$

(read " $f(N)$ is order $g(N)$ ", or " $f(N)$ is big-O of $g(N)$ ")

if there are constants c and N_0 such that for $N > N_0$,

$$f(N) \leq c * g(N)$$

for all sufficiently large N .

Big O notation

- ▶ Big O notation is used in Computer Science to describe the performance or complexity of an algorithm.
 - ▶ Big O specifically describes the worst-case scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.
 - ▶ **O(1)**
 - O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.
 - ▶ **O(N)**
 - O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.
 - Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.
 - ▶ **O(N²)**
 - O(N²) represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
 - This is common with algorithms that involve nested iterations over the data set.
- Deeper nested iterations will result in O(N³), O(N⁴) etc.

Properties of Big O Notation

- ▶ Certain essential properties of Big O Notation are discussed below:

- **Constant Multiplication:**

If $f(n) = c.g(n)$, then $O(f(n)) = O(g(n))$ where c is a nonzero constant.

- **Summation Function:**

If $f(n) = f_1(n) + f_2(n) + \dots + f_m(n)$ and $f_i(n) \leq f_{i+1}(n) \forall i=1, 2, \dots, m$,
then $O(f(n)) = O(\max(f_1(n), f_2(n), \dots, f_m(n)))$.

Recurrence Relations

- ▶ A **recurrence relation** for the sequence $\{a_n\}$ is an equation that expresses a_n in terms of one or more of the previous terms of the sequence, namely, a_0, a_1, \dots, a_{n-1} , for all integers n with $n \geq n_0$, where n_0 is a nonnegative integer.

- ▶ A sequence is called a **solution** of a recurrence relation if its terms satisfy the recurrence relation.

Recurrence Solution Approaches:

- ▶ The Substitution Approach
- ▶ The Iteration Approach

The Substitution Approach

- ▶ The substitution approach entails predicting the answer's structure, then using mathematical induction to identify the constants and demonstrate that the answer is correct. When the inductive hypothesis is applied to lower numbers, the estimated answer is substituted for the function, thus the name. This approach is effective, but it can only be used in situations when guessing the form of the response is simple.
- ▶ The substitution technique may be used to provide upper and lower boundaries on recurrences.

Let's look at an example of determining a recurrence upper bound.

$$1. T(n) = 2T([n/2]) + n$$

$T(n) = O(n \lg n)$ is our best guess for the answer.

This approach is used to demonstrate that $T(n) \leq cn \lg n$ for the proper selection of the constant $c > 0$.

- ▶ $T(n) = 2T(\lfloor n/2 \rfloor) + n$ ----- 2 $T(n) = O(n \lg n)$
- ▶ $T(n) \leq cn \lg n$ ----- 1
- ▶ Put n as $n/2$ $T(n) \leq cn \lg n$
- ▶ $T(\lfloor n/2 \rfloor) \leq c(\lfloor n/2 \rfloor) \lg (\lfloor n/2 \rfloor)$ ----- 3
- ▶ Substitute $T(\lfloor n/2 \rfloor)$ in 2
- ▶ $T(n) \leq (c \lfloor n \rfloor \lg (\lfloor n/2 \rfloor)) + n$
- ▶ $T(n) \leq cn \lg (n/2) + n$
- ▶ $T(n) \leq cn \lg n$
- ▶ $\leq cn \lg n = O(n \lg n)$

Let's look at an example of determining a recurrence upper bound.

$$1. T(n) = 2T(\lfloor n/2 \rfloor) + n$$

$T(n) = O(n \lg n)$ is our best guess for the answer. 2

This approach is used to demonstrate that $T(n) \leq cn \lg n$ for the proper selection of the constant $c > 0$.

Iterative Approach

Q. $T(n) = T(n-1) + 1$ and $T(1) = O(1)$.

$$\begin{aligned}T(n) &= T(n-1) + 1 \\&= (T(n-2) + 1) + 1 = (T(n-3) + 1) + 1 + 1 \\&= T(n-4) + 4 = T(n-5) + 1 + 4 \\&= T(n-5) + 5 = T(n-k) + k\end{aligned}$$


Where $k = n-1$ $T(n-(n-1)) + n-1 = T(1) + n-1$

$$T(n-k) = T(1) = O(1)$$

$$T(n) = O(1) + (n-1) = 1 + n - 1 = n = O(n).$$

Divide and conquer approach

- ▶ Divide and conquer approach breaks down a problem into multiple sub-problems recursively until it cannot be divided further.
- ▶ These sub-problems are solved first and the solutions are merged together to form the final solution.

- ▶ The common procedure for the divide and conquer design technique is as follows –
 - **Divide** – We divide the original problem into multiple sub-problems until they cannot be divided further.
 - **Conquer** – Then these subproblems are solved separately with the help of recursion
 - **Combine** – Once solved, all the subproblems are merged/combined together to form the final solution of the original problem.
 - ▶ There are several ways to give input to the divide and conquer algorithm design pattern. Two major data structures used are – **arrays** and **linked lists**.
- 

- ▶ This technique is commonly used in designing operations on data structures.
- ▶ For example, consider the problem of searching for an element in a sorted array. One way to solve this problem is to use a linear search algorithm that checks each element in the array one by one. However, this approach has a time complexity of $O(n)$, which can be inefficient for large arrays.
- ▶ Instead, we can use the divide-and-conquer approach to design a more efficient search algorithm.

We can divide the array into two halves and recursively search the left or right half based on whether the element we are looking for is smaller or larger than the middle element of the array. This reduces the search space by half at each step and results in a time complexity of $O(\log n)$.

- ▶ Another example is the merge sort algorithm, which uses the divide-and-conquer approach to sort an array of elements.

The algorithm divides the array into two halves, recursively sorts each half, and then merges the two sorted halves to obtain the final sorted array. This algorithm has a time complexity of $O(n \log n)$ and is widely used in practice.

- ▶ In general, the divide-and-conquer approach can be used to design efficient operations on data structures such as trees, graphs, and arrays. The key idea is to break down the problem into smaller subproblems and solve each subproblem independently, before combining the results to obtain the final solution.