# University of Barishal
# Department of Computer Science and Engineering

**CSE-2201 , Design and Analysis of Algorithms**
**2nd Year 2nd Semester**
**Session: 2018-19**

# Introduction to Algorithms By CLRS
**Summary of Chapter 1,2,3**

**Topics:**
The Role of Algorithms in Computing
Getting Started
Growth of Function

**Prepared by:**
Jubayer Abdullah Joy
Roll: 18CSE040
Session: 2018-19

**Submitted to:**
Md Erfan, Lecturer, Dept. of CSE, BU

# 1. The Role of Algorithms in Computing

This Chapter covers introduction to Algorithms, Insertion sort, Analysis of Algorithms, Designing algorithms, Asymptotic notation, Standard notations and common functions

## 1.1 Algorithms

An algorithm is a sequence of computational steps that transform the input into the desired output. To Solve a well-specified computational problem, Algorithms work like a tool. For example, sorting some numbers in ascending or descending order by using insertion sort algorithm.

Data structure is a way to organize and store data in order to facilitate access and modification. Different data structures has different strengths and limitations.

## 1.2 Algorithms as a technology

Computers are not infinitely fast and memory is not unlimited. So, implemented solutions must efficiently use those resources. Some solutions for the same problem may differ dramatically in terms of efficiency. To sort $10^7$ random number in a system capable of processing $10^7 \ instruction / second$:

Insertion Sort ( $O(n^2)$ ): $\frac{(10^7)^2 \ instruction}{10^7 \ instruction / second} = 10000000 \ seconds \ (2777.778 \ hours)$

Merge Sort ( $O(nlogn)$ ): $\frac{10^7 \ log \ (10^7) \ instruction}{10^7 \ instruction / second} = 23.2535 \ seconds$

So, to find a solution using the same system, Insertion Sort will take approx. $23.2535 \ days$ whereas Merge Sort will take only $23.2535 \ seconds$. This is a good example of the importance of using efficient Algorithms.

At the end of this chapter, we can conclude that, Algorithms and Data Structures are essential for solving problems efficiently. This is the factor which differentiate great programmers from average. So, deep knowledge about Algorithms is a must.

## 2. Getting Started

In the beginning of the $2^{nd}$ Chapter we examine $Insertion\ Sort$ Algorithm to solve sorting problems. We define $pseudocode$ to specify its steps, verify it sorts correctly, analyze running time of this algorithm. Then we develop $Merge\ Sort$ Algorithm using divide-and-conquer method. Afterwards, we analyze its running time.

### 2.1 Insertion Sort

Insertion Sort is a sorting algorithm that can sort a sequence of numbers. It results in $a_i \leq a_{i+1}$, $i = [0, 1, 2, \ldots, n-1]$

$Pseudocode$ :

$INSERTION - SORT(A)$
$for\ j = 2\ to\ A.length$
    $key = A[j]$
    $i = j - 1$
    $while\ i > 0\ and\ A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

To Verify its correctness we test the algorithm for $A = \{5, 3, 1, 4, 2, 6\}$. It Yields to sorted permutation of $A$ and $a_i \leq a_{i+1}$ is maintained. We also verify it using *loop Variant.* As Initialization is true, Maintenance is true and Termination results in a desired output, the algorithm is correct.

### 2.2 Analyzing Algorithms

For insertion sort:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

By simplifying this equation we can see it forms $an^2 + bn + c$
Worst-case is when sequence is reverse sorted and average-case is similar when sequence is random.

### 2.2 Designing Algorithm

The divide-and-conquer approach: To divide the problem into subproblems, Conquer them by solving them recursively and finally combine them into one solution for the original problem.

Merge Sort Algorithm is an example of divide-and-conquer.

To analyze this algorithm's running time, recurrence equation is taken into account.

$$T(n) = \begin{cases} \ominus(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

by analyzing recurrence equation, we can say that for Merge Sort, time complexity is $O(nlogn), \ominus(nlogn)\ and\ \Omega(nlogn).$

## 3. Growth of Functions

The order of growth of the running time of an algorithm gives simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of the alternative algorithms. This is the asymptotic efficiency of algorithms.

### 3.1 Asymptotic Notations

$\ominus - notation$ : This refers to the average case running time of an algorithm.
$O - notation$ : This refers to the worst case running time of an algorithm.
$\Omega - notation$ : This refers to the best case running time of an algorithm.

**Comparing functions:**
Many of the relational properties of real numbers apply to asymptotic comparisons as well. Transitivity, Reflexivity, Symmetry, Transpose symmetry, Trichotomy are supported for Asymptotic notations.

### 3.2 Standard notations and common functions

**Monotonicity** :

A function f is monotonically increasing if $m \leq n$ implies $f(m) \leq f(n)$. Similarly, it is monotonically decreasing if $m \leq n$ implies $f(m) \geq f(n)$ is strictly increasing if if $m \leq n$ implies $f(m) < f(n)$ and strictly decreasing if $m \leq n$ implies $f(m) > f(n)$.

**Floor and ceilings :**
The floor function denotes the biggest integer less than or equal to $x$ $by \lfloor x \rfloor$.
The ceiling function denotes the least integer greater than or equal to $x$ $by \lceil x \rceil$.
Both of the functions are monotonically increasing.

**Modular arithmetic :**
For any integer $a$ and any positive integer $n$, the value $a \bmod n$ is the remainder of the quotient $a / n$:
$a \bmod n = a - n\lfloor a / n \rfloor$, $0 \leq a \bmod n < n$.

**Polynomials:**
A function $f(n)$ is polynomially bounded if $f(n) = O(n^k)$ for some constant k.

**Exponentials:**
Any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

**Logarithms:**
Notations are as followed:
$lg\ n = log_2 n$
$ln\ n = log_e n$
$lg^k\ n = (lg\ n)^k$
$lg\ lg\ n = lg(lg\ n)$

**Factorials:**
The notation $n! = 1 * 2 * 3 \ldots n$, $n \geq 0$
$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$

**Fibonacci numbers:**
$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$ This recurrence defines fibonacci number. Its related to *golden ratio* $\Phi$ *and its conjugate* $\delta$ *silver ratio.*
$F_i = \frac{\Phi^i - \delta^i}{\sqrt{5}} \Rightarrow F_i = \lfloor \frac{\Phi^i}{\sqrt{5}} + \frac{1}{2} \rfloor$, thus the fibonacci number grows exponentially.