# ALGORITHMS
# CSE 2201

**Course Teacher**

Anupam Kumar Bairagi, PhD
Associate Professor
Computer Science and Engineering Discipline
Khulna University, Khulna-9208, Bangladesh

## You can follow ~~me~~ my works in the following links:

- http://discipline.ku.ac.bd/cse/faculty/anupam
- https://www.researchgate.net/profile/Anupam_Bairagi2
- https://scholar.google.co.kr/citations?user=68f3-kQAAAAJ&hl=en

- **Disliking in the Class Room**:
  - Religious Discussion
  - Political Discussion
  - Use of Mobile and other Electronic Devices
  - Any kind of Indiscipline

## Course Overview

- Schedule (Every week)
  - Tuesday, 10.50 am ~ 12.30 pm (1.40 hours)
  - Wednesday, 11.40 am ~ 12.30 pm (50 minutes)
- Textbook and References
  - Ellis Horowitz, and Sartaj Sahni, *Fundamentals of Computer Algorithms*, Galgotia Publications (P) Ltd.
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, Prentice-Hall of India Private Limited.
  - Handouts that I will provide
- Grading Distribution
  - Attendance – **10%**
  - Class Test / Quizz – **30%**
  - Final Exam – **60%**

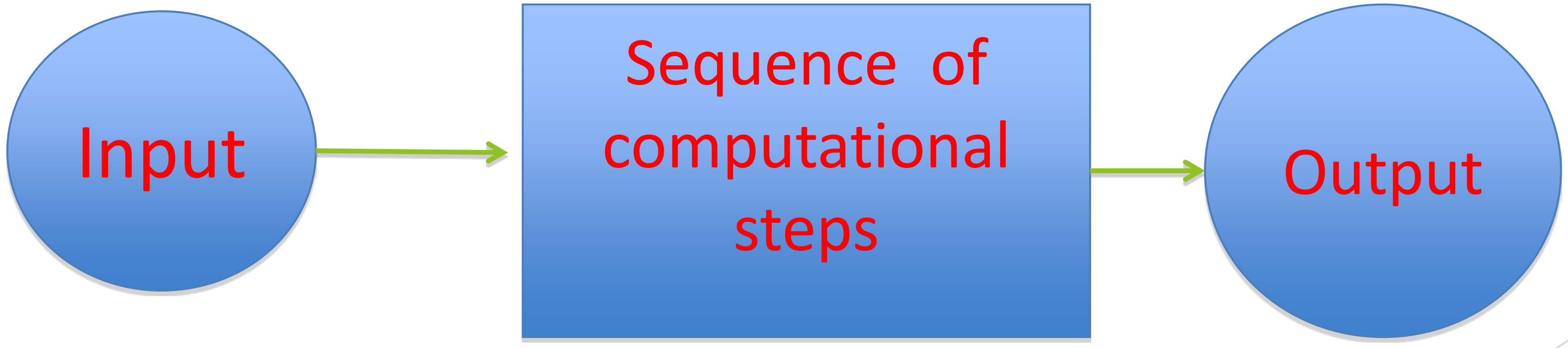
## What is this Course About?

- ❑ Analyze the asymptotic performance of algorithms.
- ❑ Apply important algorithmic design paradigms and methods including :
  - Divide and Conquer
  - Dynamic Programming
  - Greedy Algorithms
  - Backtracking
  - And many more

## Introduction

## Algorithm

- An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.



## Example of Algorithm: Sorting Problem

- **Input**: A sequence of n numbers: $a_1, a_2, \ldots, a_n$
- **Output**: A permutation (reordering) $a_1, a_2, \ldots, a_n$ of the input sequence such that $a_1 \le a_2 \le \ldots \le a_n$
- **Example**: *Input: sequence 31, 41, 59, 26, 41, 58*

  *Output: sequence 26, 31, 41, 41, 58, 59*

## Correct Algorithms

- An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a correct algorithm solves the given computational problem.
- An incorrect algorithm might not halt at all on some input instances, or it might halt with an answer other than the desired one.
- Incorrect algorithms can sometimes be useful, if their error rate can be controlled. (An example of this when we study algorithms for finding large prime numbers.)

## Hard problems

- There are some problems for which no efficient solution is known, which are known as NP-complete:
  - it is unknown whether or not efficient algorithms exist for NP-complete problems.
  - the set of NP-complete problems has the remarkable property that if an efficient algorithm exists for any one of them, then efficient algorithms exist for all of them.
  - a small change to the problem statement can cause a big change to the efficiency of the best known algorithm.

## Choosing Algorithms

- **Example**: Fibonacci sequence is defined as follows:

  F(0) = 0, F(1) = 1, and

  F(n) = F(n-1) + F(n-2) for n > 1.

- Write an algorithm to computer F(n).

- There many algorithms, but what is the most efficient ?

## Algorithms 1 and 2 for Fibonacci

- **Algorithm 1 for Fibonacci**

```
function fib1(n){
   if n < 2 then return n;
   else return fib1(n-1) + fib1(n-2);
   }
```

- **Algorithm 2 for Fibonacci**

```
function fib2(n){
   i= 1;  j = 0;
   for k = 1 to n do { j = i+j; i = j- i;}
   return j;
   }
```

## Algorithms 3 for Fibonacci

- **Algorithm 1 for Fibonacci**

```
function  fib3(n){
   i = 1; j = 0; k = 0; h = 1;
   while n>0 do {
         if (n odd) then {  t = jh;
                     j = ih + jk +t;
                     i = ik +t;}
         t = h^2;
         h = 2kh+t;
         k = k^2+t;
         n = n div 2;}
   return j;
}
```

## Example of Running Times for Fibonacci

| n | 10 | 20 | 30 | 50 | 100 | 10000 | 1000000 | 100000000 |
|---|---|---|---|---|---|---|---|---|
| fib1 | 8 ms | 1 s | 2 min | 21 days | | | | |
| fib2 | 1/6 ms | 1/3 ms | ½ ms | ¾ ms | 3/2 ms | 150 ms | 15 s | 25 min |
| fib3 | 1/3 ms | 2/5 ms | ½ ms | ½ ms | ½ ms | 1 ms | 3/2 ms | 2 ms |

## Insertion Sort

- Efficient algorithm for sorting a small number of elements

- We start with an empty left hand and the cards face down on the table.
- We then remove one card at a time from the table and insert it into the correct position in the left hand.
- To find the correct position for a card, we compare it with each of the cards already in the hand, from right to left.

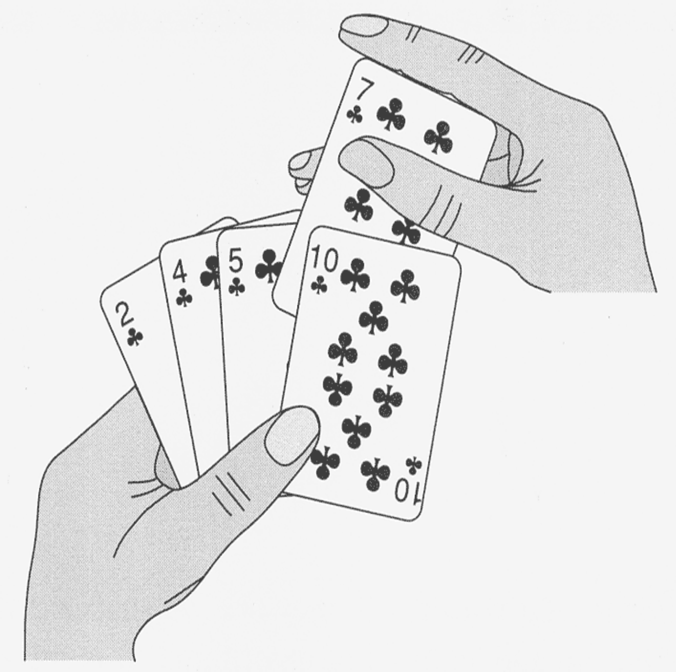


**Figure 2.1** Sorting a hand of cards using insertion sort.

# Insertion Sort (Cont..)

```
INSERTION-SORT(A)
    for j = 2 to length[A] do
        key = A[j]
        //Insert A[j] into the sorted sequence A[1.. j - 1].
        i = j-1
        while i>0 and A[i]>key
            A[i + 1] = A[i]
            i = i-1
        end while
        A[i+1] = key
    end for
```

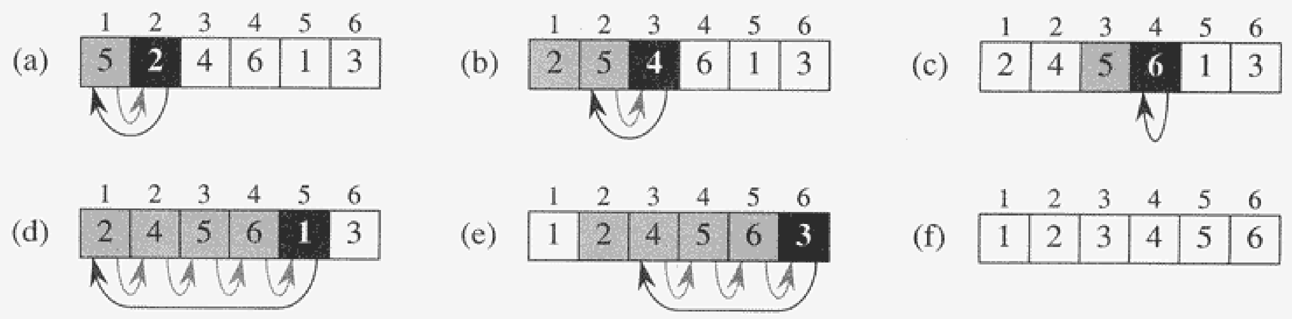
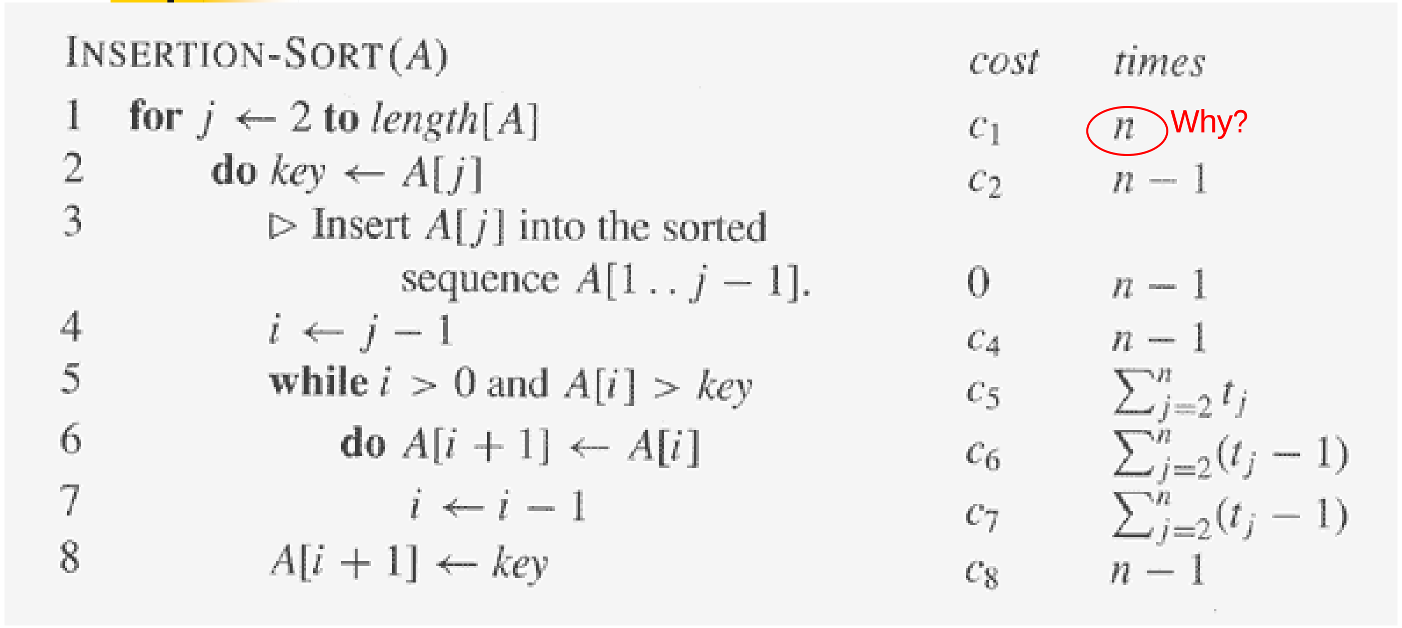

(a) 1 2 3 4 5 6 | 5 2 4 6 1 3
(b) 1 2 3 4 5 6 | 2 5 4 6 1 3
(c) 1 2 3 4 5 6 | 2 4 5 6 1 3
(d) 1 2 3 4 5 6 | 2 4 5 6 1 3
(e) 1 2 3 4 5 6 | 1 2 4 5 6 3
(f) 1 2 3 4 5 6 | 1 2 3 4 5 6

**Figure 2.2** The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

# Analyzing Algorithms

- Analyzing an algorithm: for an input size,
  - measure memory (space)
  - measure computational time (running time).
- Input size: depends on the problem:
  - Sorting: number of items in the input; array size,… O(n)
  - Big integer (multiplying, …): number of bits to represent the input in binary notation O(log n)
  - Two number: input of a graph can be O(n, m), number of vertices and number of edges.
- Running time:
  - A constant amount of time is required to execute each line
  - Each execution of the ith line takes time $c_i$ , where $c_i$ is a constant.

# Analysis of Insertion Sort



| | INSERTION-SORT$(A)$ | *cost* | *times* |
|---|---|---|---|
| 1 | **for** $j \leftarrow 2$ **to** *length*$[A]$ | $c_1$ | $n$ Why? |
| 2 | **do** *key* $\leftarrow A[j]$ | $c_2$ | $n-1$ |
| 3 | $\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$. | $0$ | $n-1$ |
| 4 | $i \leftarrow j-1$ | $c_4$ | $n-1$ |
| 5 | **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 | **do** $A[i+1] \leftarrow A[i]$ | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 7 | $i \leftarrow i-1$ | $c_7$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 8 | $A[i+1] \leftarrow key$ | $c_8$ | $n-1$ |

- ❑ What are the *best* and *worst*-case running?
- ❑ How about *average*-case?

$t_j$ is the number of times the while loop test in line 5 is executed for that value of j.

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

Where are $c_6$ & $c_7$ !!

## Analysis of Insertion Sort (Cont..)

- **Best case**: the array is already sorted

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

$$T(n) = an + b$$

Linear function of n

## Analysis of Insertion Sort (Cont..)

- **Worst case**: the array is already in reverse sorted order

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$T(n) = \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{C_5}{2} - \frac{C_6}{2} - \frac{C_7}{2} + c_8\right)\mathrm{n} - (c_2 + c_4 + c_5 + c_8)$$

$$T(n) = an^2 + bn + c$$

Quadratic function of n

## Growth Function

- Asymptotic notation
  - The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency.
  - For input sizes large enough, we make only the order of growth of the running time relevant, so we study the asymptotic efficiency of algorithms.

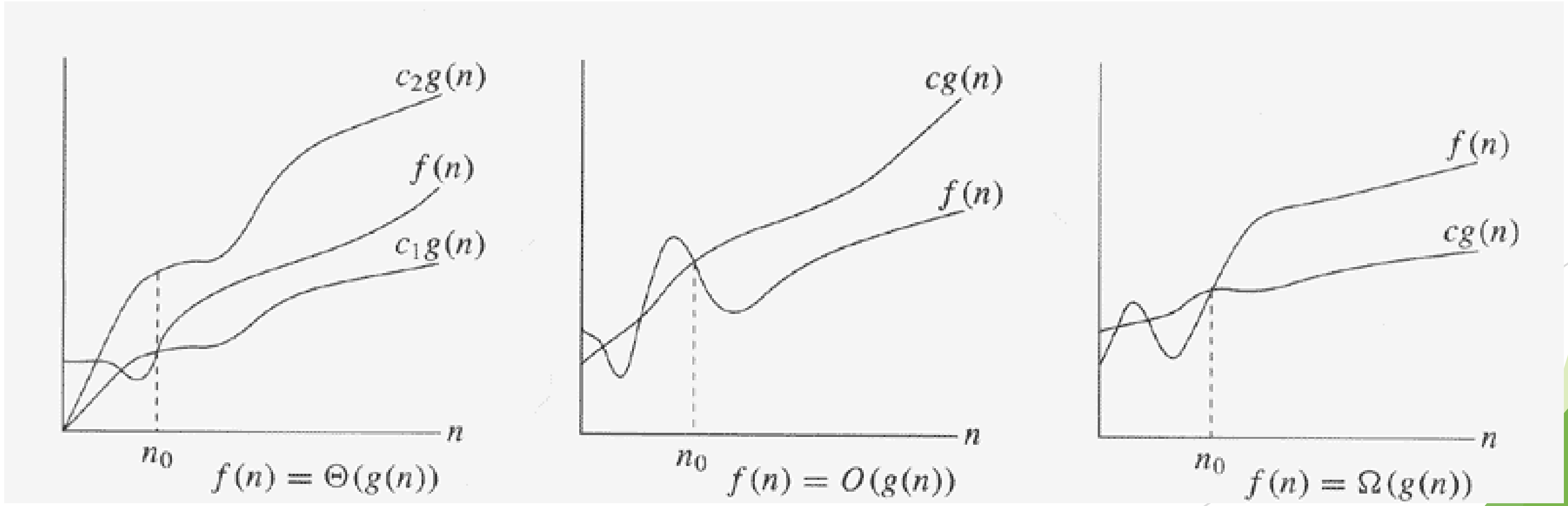
# Asymptotic notation

- **Θ notation**
  - $\Theta(g(n)) = \{f(n)$ : *there exist positive constants* $c_1$, $c_2$, *and* $n_0$ *such that* $0 \le c_1\, g(n) \le f(n) \le c_2\, g(n)$ *for all* $n \ge n_0\}$.
  - *We usually write* $f(n) = \Theta(g(n))$ *to express the same notation*
  - *We say that g(n) is an asymptotically tight bound for f(n)*
- **O notation**
  - $O(g(n)) = \{f(n)$: *there exist positive constants c and* $n_0$ *such that*

    $0 \le f(n) \le cg(n)$ *for all* $n \ge n_0\}$
  - *We write* $f(n) = O(g(n))$ to indicate that a function f(n) is a member of the set O(g(n))
  - *We use* O-notation when we have an asymptotic upper bound

## Asymptotic notation (Cont..)

- **Ω notation**
  - Ω(*g(n)*) = {*f(n)*: *there exist positive constants c and* $n_0$ *such that* $0 \le cg(n) \le f(n)$ *for all* $n \ge n_0$}.
  - *We write f(n)*= Ω (g(n)) to indicate that a function f(n) is a member of the set Ω (g(n))
  - Ω-notation provides an asymptotic lower bound



Thanks for your Attention

Q&A