Designing and Analysis of Algorithms

Course Code: ECS 5101/CS514

- Dr Rahul Mishra
- IIT Patna

- **Lecture 4**

Growth of function

* Suppose M is an algorithm, and suppose n is the size of the input data.

* The complexity $\boxed{f(n) \text{ of } M}$ increases as $\boxed{n}$ increases.

* It is usually the rate of increase of $\boxed{f(n)}$ that we want to examine.

* This is usually done by comparing $f(n)$ with some standard function

$$\boxed{\log_2 n, \quad n, \quad n\log_2 n, \quad n^2, \quad n^3, \quad 2^n}$$

n \ $g(n)$	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	10^3
100	7	100	700	10^4	10^6	10^{30}
1000	10	10^3	10^4	10^6	10^9	10^{300}

"Rate of Growth of Standard Functions"

Asymptotic Analysis

- ***Dictionary meaning:*** *"Asymptotic function approaches a given value as an expression containing a variable tends to infinity."*

- We are concerned with how the **running time** of an algorithm increases with the size **of the input in the limit, as the size of the input increases without bounds.**

- An algorithm that is asymptotically more efficient will be the best choice for all but very small input.

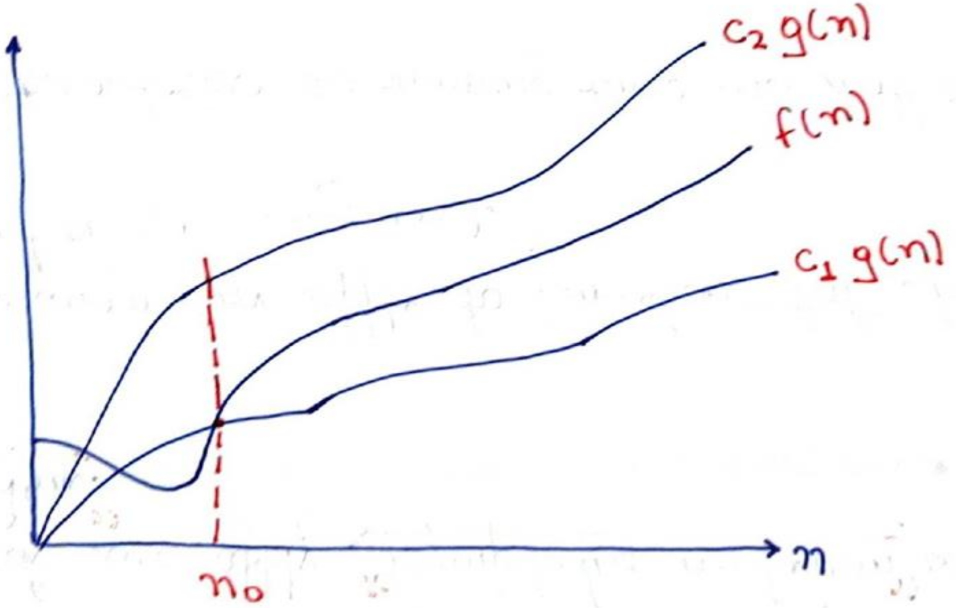
- The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domain is the set of natural numbers, **N** = {0,1,2,…}.

- They are used for defined worst-case running-time function **T(n),** which usually is defined only on integer input sizes.

Asymptotic Analysis: $\theta - notation$

Let us define what this notation mean. For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions.

$$\boxed{\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \geq n_0 \}.}$$



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

Asymptotic Analysis: $\theta - notation$

Θ - Notation

* A function $\boxed{f(n)}$ belongs to the set $\boxed{\Theta(g(n))}$ if there exist positive constants $\boxed{c_1}$ and $\boxed{c_2}$ such that it can be "sandwiched" between $\boxed{c_1 g(n) \text{ and } c_2 g(n)}$, for sufficiently large $\boxed{n}$.
    * → (read as f of n is theta of g of n)

* Since $\Theta(g(n))$ is a set, we can write "$f(n) \in \Theta(g(n))$" to indicate that $f(n)$ is a member of $\Theta(g(n))$.

* Instead, we will usually write "$f(n) = \Theta(g(n))$" to express the same notion.

* An intuitive picture of functions $\boxed{f(n) \text{ and } g(n)}$, where $f(n) = \Theta(g(n))$.

* For all values of n at and to the right of n_0, the value of $f(n)$ lies at or above $\boxed{c_1 g(n)}$ and at or below $\boxed{c_2 g(n)}$.

* $g(n)$ is an asymptotically tight bound for $f(n)$.

Asymptotic Analysis: $\theta - notation$

The definition of θ(g(n)) require that every member $f(n) \in \theta\left(g_{(n)}\right)$ by asymptotically non-negative, that is, that f**(n)** be non-negative whenever **n** is sufficiently large.

Example → $f(n) = 18n + 9$

since $f(n) \geq 18n$ and $f(n) \leq 27n$,

for $n \geq 1$

• $g(n) = 3n$

$$c_1 3n \leq \underbrace{18n + 9}_{f(n)} \leq c_2 3n$$

$c_1 = 6 \quad c_2 = 9$

Asymptotic Analysis: $\theta - notation$

* Let us briefly justify this intuition by using the formal definition to show that

$$\boxed{\tfrac{1}{2}n^2 - 3n = \Theta(n^2)}$$

* To do so, we must determine positive constants c_1, c_2, and n_0 such that

$$\boxed{c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2}$$

for all $n \geq n_0$. Dividing by n^2 yields.

$$\boxed{c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2}$$

* We can make the right-hand inequality hold for any value of $\boxed{n \geq 1}$ by choosing any constant $\boxed{c_2 \geq 1/2}$

* We can make the left-hand inequality hold for any value of $\boxed{n \geq 7}$ by choosing any constant $\boxed{c_1 \le 1/14}$.

Asymptotic Analysis: $\theta - notation$

* By choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $\boxed{\frac{1}{2} n^2 - 3n = \Theta(n^2).}$

* Certainly, other choices for the constants exist, but the important thing is that some choice exists.

* These constants depend on the function $\boxed{\frac{1}{2} n^2 - 3n}$; a different function belonging to $\boxed{\Theta(n^2)}$ would usually requires different constants.

* We can also use the formal definition to verify that $\boxed{6n^3 \neq \Theta(n^2)}$. Suppose for the purpose of contradiction that $\boxed{c_2}$ and $\boxed{n_0}$ exist such that

$$\boxed{6n^3 \leq c_2 n^2} \text{ for all } n \geq n_0.$$

* But then dividing by $\boxed{n^2}$ yield $\boxed{n \leq c_2/6}$, which cannot possibly hold for arbitrarily large n, since c_2 is constant.

Asymptotic Analysis: $\theta - notation$

* The lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n.

* When n is large, even a tiny fraction of the highest-order term suffices to dominate the lower-order terms.

* Thus, setting c_1 to a value that is slightly smaller than the coefficient of the highest-order term and setting c_2 to a value that is slightly ~~smaller than the coefficient~~ larger permits the inequalities in the definition of Θ-notation to be satisfied.

$$\boxed{f(n) = an^2 + bn + c}$$

where a, b, and c are constants and $a > 0$.

* The lower-order terms and ignoring the constant yields $f(n) = \Theta(n^2)$.

* To show the same thing, we take the constants $\boxed{c_1 = a/4,\ c_2 = 7a/4,\ \text{and } n_0 = 2}$.

$$\boxed{n_0 = 2 \cdot \max\left(|b|/a, \sqrt{|c|/a}\right)}$$

Asymptotic Analysis: $\theta - notation$

We can verify that $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ for all n>= n0.

* In general, for any polynomial

$\boxed{p(n) = \sum_{i=0}^{d} a_i n^i}$, where the a_i are constants and $a_d > 0$

* We have $p(n) = \Theta(n^d)$.

* Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$, or $\Theta(1)$.

* We shall often use the notation $\Theta(1)$ to mean either a constant or a constant function with respect to some variable.

Insertion sort

Let us consider a simple example of the Insertion sort algorithm. It is a simple comparison-based sorting algorithm. Insertion sort is an effective algorithm used for sorting a limited number of elements.

This method resembles how teachers typically sort our exam copies. It involves organizing a collection of exam copies based on their grades or other relevant criteria. The process resembles how people sort a hand of playing cards. Initially, You have a stack of exam copies face down on a table. You start with an empty pile. You pick up one copy at a time and insert it into its correct position among the copies you've already sorted. Next, to find the correct position, you compare the selected copy with the ones already in your sorted pile. This comparison is usually done from right to left, making sure that you place the copy in the right position based on the sorting criteria (*e.g.,* highest grade to lowest grade). You repeat this process until you've gone through all the exam copies. By the end of this process, you will have a sorted stack of exam copies. The ones you are holding are sorted, and they were originally the top copies from the initial unsorted stack on the table.

Different steps of Insertion sort are as follows:

1) Start with the **second element (Index 1)** in the unsorted array and assume it is the start of the sorted portion.
2) Compare the current element with the one before it. If they are in the wrong sorted order, **swap** them.
3) Move the current element back through the sorted portion of the array until it is in the **correct position**.
4) Move to the next element (Index 2) and repeat **steps 2 and 3** until the entire array is sorted.

Insertion sort

"pseudocode"

INSERTION-SORT (A, n) ⊳ $A[1 \,.\,.\, n]$

for $j \leftarrow 2$ **to** n

  do $key \leftarrow A[j]$

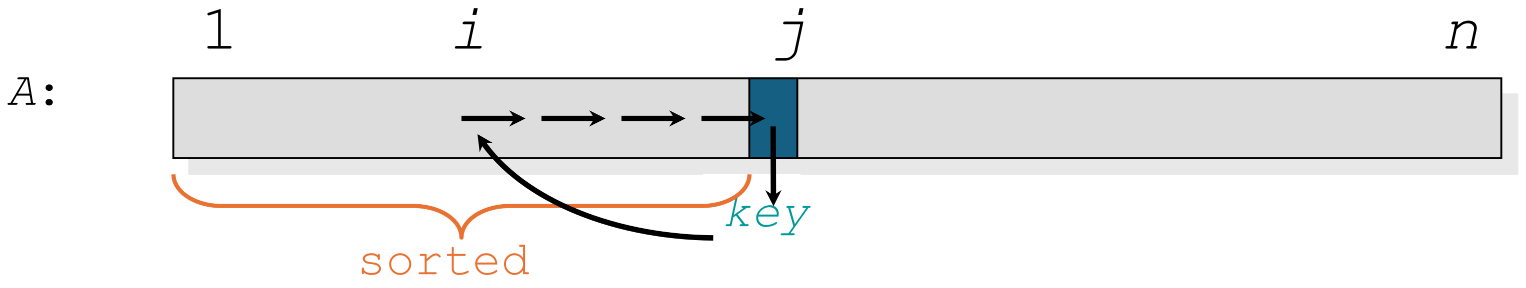
    $i \leftarrow j - 1$

    while $i > 0$ and $A[i] > key$

      do $A[i+1] \leftarrow A[i]$

        $i \leftarrow i - 1$

    $A[i+1] = key$



Example of insertion sort

8 2 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

Example of insertion sort

8 2 4 9 3 6

2 8 4 9 3 6

2 4 8 9 3 6

2 4 8 9 3 6

2 3 4 8 9 6

Example of insertion sort

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6

Example of insertion sort

8	2	4	9	3	6	
2	8	4	9	3	6	
2	4	8	9	3	6	
2	4	8	9	3	6	
2	3	4	8	9	6	
2	3	4	6	8	9	*done*

Insertion sort

Step 1:	8	2	4	9	3	6

Swap

Step 2:	2	8	4	9	3	6

Swap

Step 3:	2	4	8	9	3	6

No Swap

Step 4:	2	4	8	9	3	6

Swap

Step 5:	2	3	4	8	9	6

Swap

Output:	2	3	4	6	8	9

```
InsertionSort (array Arr):
    for i in range(2, len(Arr)):
        current_element = Arr[i]
        j = i - 1
        # Move elements of arr[0..i-1], that are greater than current_element,
        # one position ahead of their current position
        while j >= 0 and current_element < Arr[j]:
            Arr[j + 1] = Arr[j]
            j = j-1
        Arr[j + 1] = current_element
```