



CS-2001

DATA STRUCTURES

FALL2021

LECTURE#2 AND LECTURE#3 (COMBINED)

Dr. Hashim Yasin

Mr. Muhammad Husnain

**National University of Computer and
Emerging Sciences,
Faisalabad, Pakistan.**

Algorithm

- An algorithm is a definite procedure for solving a problem in a finite number of steps.
- Algorithm is a well-defined computational procedure that takes some value (s) as input and produces some value (s) as output.
- Algorithm is finite number of computational statements that transform input into the output

Algorithm

- Finite sequence of instructions.
- Each instruction having a clear meaning.
- Each instruction requires **finite amount of effort**.
- Each instruction requires **finite time to complete**.

Evaluation of Algorithm

- **Completeness:** is the strategy guaranteed to find a solution when there is one?
- **Optimality:** does the strategy find the highest-quality (least-cost) solution when there are several different solutions?
- **Time complexity:** how long does it take to find a solution?
- **Space complexity:** how much memory is needed to perform the search?

Analysis of an Algorithm

- What is the goal of analysis of algorithms?
 - ▣ To compare algorithms mainly **in terms of running time** but also in terms of **other factors** (e.g., memory requirements, programmer's effort etc.)

- What do we mean by **running time analysis**?
 - ▣ **Determine how running time increases as the **size** of the problem increases.**

Analysis of an Algorithm

- Ways of measuring efficiency:
 - ▣ Run the program and see how long it takes
 - ▣ Run the program and see how much memory it uses

- Lots of variables to control:
 - ▣ What is the input data?
 - ▣ What is the hardware platform?
 - ▣ What is the programming language/compiler?

Types of Analysis

❑ Worst case

- ❑ Provides an **upper bound** on running time
- ❑ An absolute **guarantee** that the **algorithm would not run longer**, no matter what the inputs are.

❑ Best case

- ❑ Provides a **lower bound** on running time
- ❑ Input is the one for which the **algorithm runs the fastest**

Types of Analysis

□ **Average case**

- Provides a **prediction** about the running time
- Assumes that the input is random.

$$\textit{Lower Bound} \leq \textit{Running Time} \leq \textit{Upper Bound}$$

Asymptotic Notation

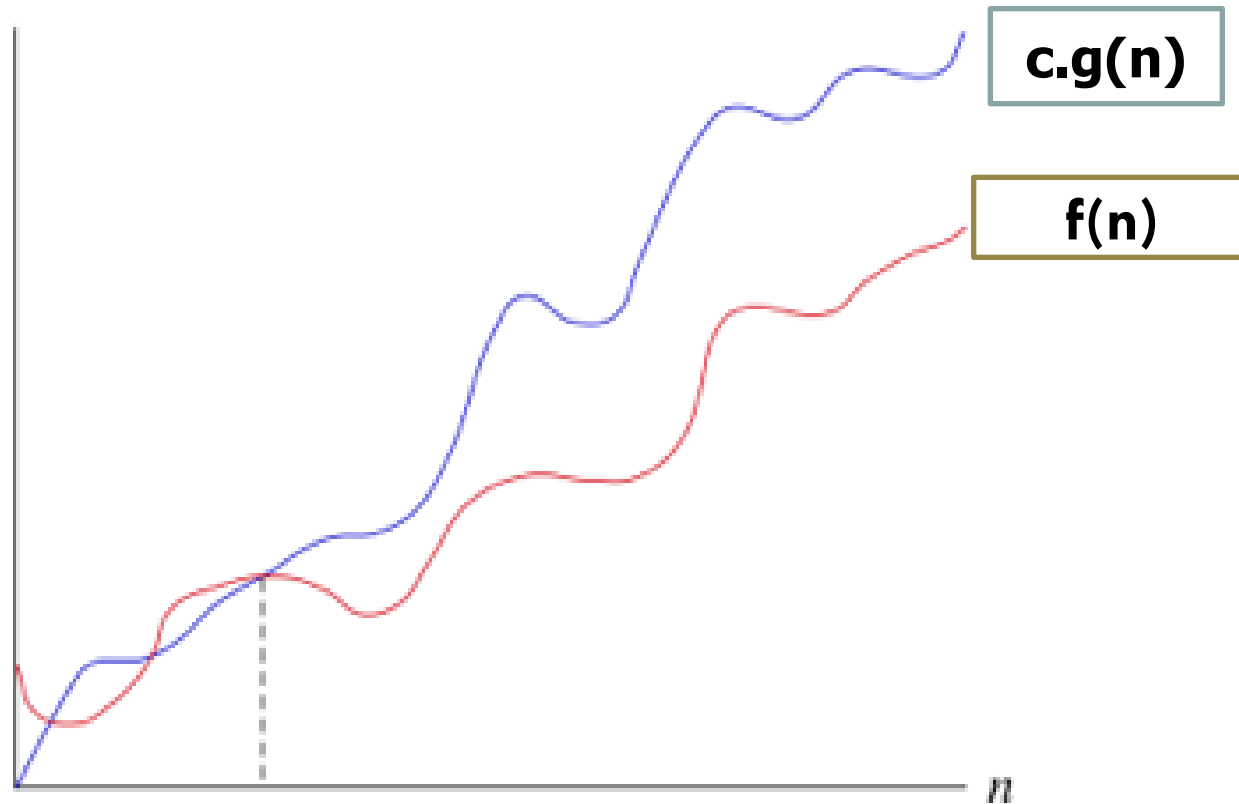
- Asymptotic notations – Asymptotic notations are the notations used to describe **the behavior of the time or space complexity**.
- Let us represent the time complexity and the space complexity using the common function $f(n)$.
 - O (*Big Oh notation*)
 - Ω (*Omega notation*)
 - θ (*Theta notation*)
 - o (*Little Oh notation*)

○ – Big Oh notation

- The big Oh notation provides an **upper bound** for the function $f(n)$.
- The function $f(n) = O(g(n))$ if and only if there exists positive constants c and n_0 such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0$$

○ – Big Oh notation



O – Big Oh notation

$$f(n) = 3n + 2$$

$$f(n) \leq cg(n) \text{ for all } n \geq n_0$$

Let us take $g(n) = n$

$$c = 4$$

$$n_0 = 2$$

Let us check the above condition

$$3n + 2 \leq 4n \quad \text{for all } n \geq 2$$

The condition is satisfied. Hence $f(n) = O(n)$.

O – Big Oh notation

$$f(n) = 10n^2 + 4n + 2$$

$$f(n) \leq cg(n) \text{ for all } n \geq n_0$$

Let us take $g(n) = n^2$

$$c = 11$$

$$n_0 = 6$$

Let us check the above condition

$$10n^2 + 4n + 2 \leq 11n \quad \text{for all } n \geq 6$$

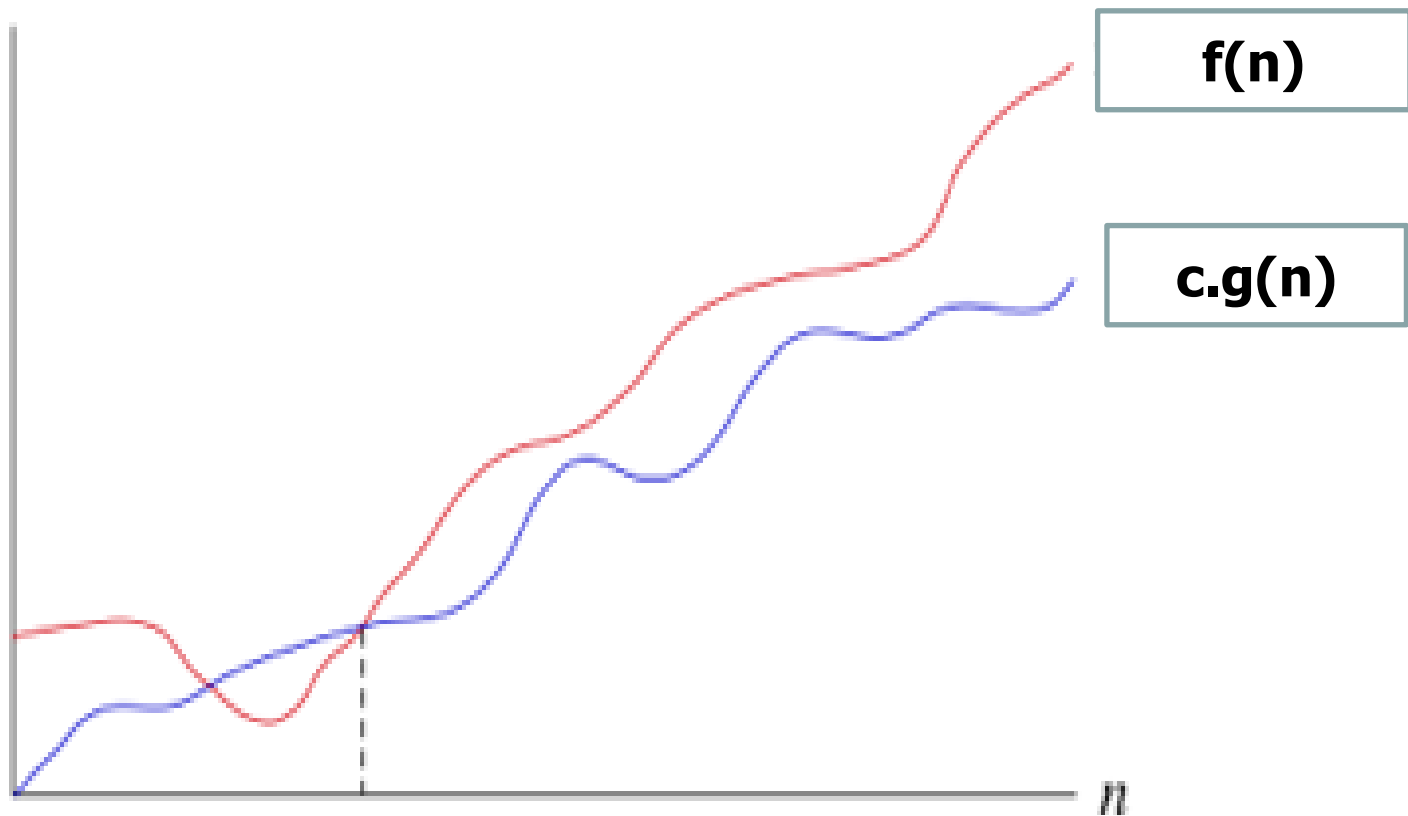
The condition is satisfied. Hence $f(n) = O(n^2)$.

Ω - Omega notation

- The omega notation (Ω) provides a **lower bound** for the function $f(n)$.
- The function $f(n) = \Omega(g(n))$ if and only if there exists positive constants c and n_0 such that

$$f(n) \geq cg(n) \text{ for all } n \geq n_0$$

Ω - Omega notation

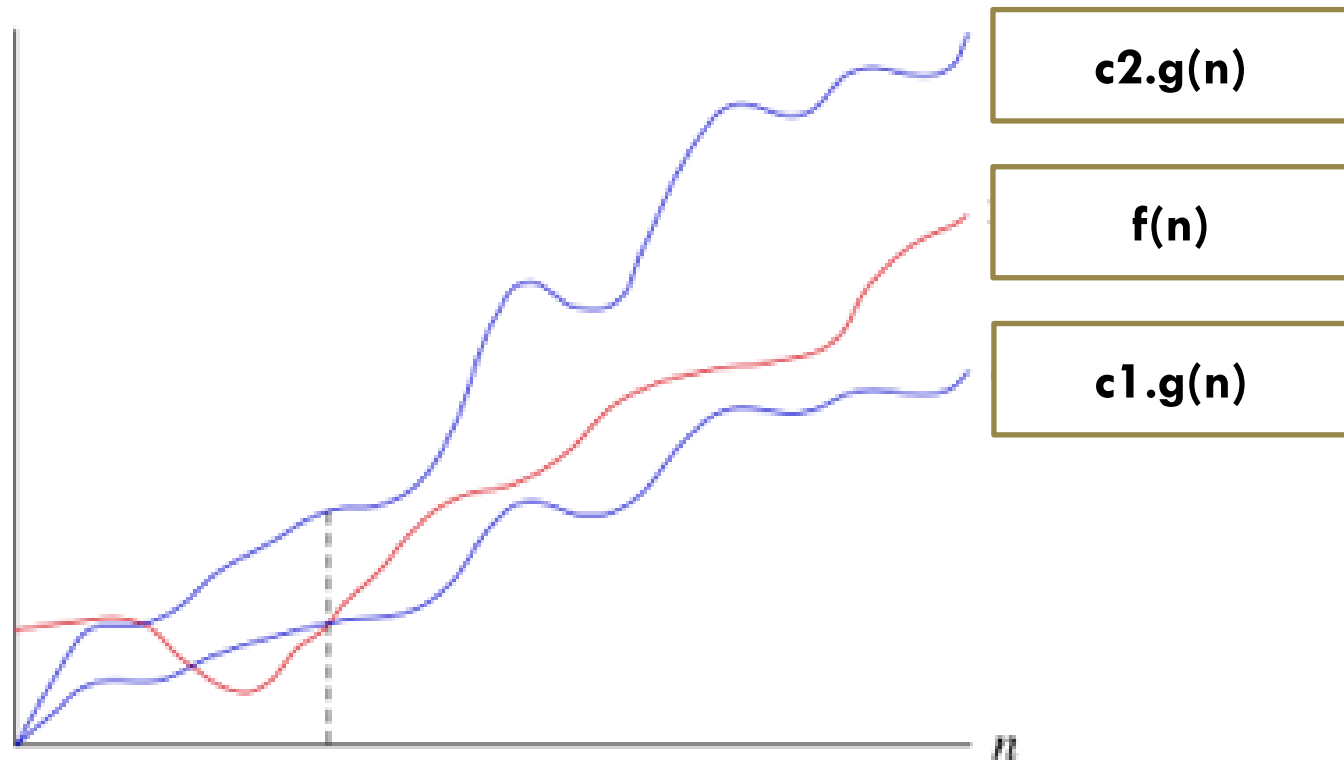


θ – Theta notation (Tight Bound)

- The theta notation (θ) is used when the function $f(n)$ can be bounded by both from above and below the same function $g(n)$.
- The function $f(n) = \theta(g(n))$ if and only if there exists positive constants c_1, c_2 and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0$$

Θ – Theta notation (Tight Bound)



o - Little Oh notation

- The little o notation (o) is,

$$\begin{aligned} f(n) = o(g(n)) \text{ if and only if} \\ f(n) = O(g(n)) \text{ and} \\ f(n) \neq \Omega(g(n)) \\ (OR) \end{aligned}$$

- *$(f(n) / g(n))$ becomes zero as 'n' approaches to infinity*
 - *i.e., $f(n)$ is strictly less than $g(n)$ in order of growth*

Asymptotic Notation

- O notation: asymptotic “less than”:
 - ▣ $f(n) = O(g(n))$ implies: $f(n) \leq g(n)$
- Ω notation: asymptotic “greater than”:
 - ▣ $f(n) = \Omega(g(n))$ implies: $f(n) \geq g(n)$
- θ notation: asymptotic “equality”:
 - ▣ $f(n) = \theta(g(n))$ implies: $f(n) = g(n)$

Analysis of a Simple Program

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A
int Sum(int A[], int N){
1.   int s=0;
2.   for (int i=0; i< N; i++)
3.       s = s + A[i];
4.   return s;
}
```

How should we analyse this?

Analysis of a Simple Program

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A
```

```
int Sum(int A[], int N){
```

```
    int s=0; ← ①
```

```
    for (int i=0; i< N; i++) ← ②, ③, ④
```

```
        s = s + A[i]; ← ⑤, ⑥, ⑦
```

```
    return s; ← ⑧
}
```

1,2,8: Once

3: N+1 times

4,5,6,7: Once per each iteration
of for loop, N iteration

Total: $5N + 4$

The *complexity function* of the
algorithm is : $f(N) = 5N + 4$

Analysis of a Simple Program

- Estimated running time for different values of N for $5N+4$:
 - ▣ $N = 10 \Rightarrow 54$ steps
 - ▣ $N = 100 \Rightarrow 504$ steps
 - ▣ $N = 1,000 \Rightarrow 5004$ steps
 - ▣ $N = 1,000,000 \Rightarrow 5,000,004$ steps
- As N grows, the number of steps grow in linear proportion to N for this function “Sum”

Example 1 Constant time

```
void printFElementOfArray(int arr[]) {  
    printf("First element of array = %d",arr[0]);  
}
```

- This function runs in $O(1)$ time (or "constant time") relative to its input.

Constant time statements

- Simplest case: $O(1)$ time statements
- Assignment statements of simple data types
 - ▣ `int x = y;`
- Arithmetic operations:
 - ▣ `x = 5 * y + 4 - z;`
- Array referencing:
 - ▣ `A[i] = 5;`
- Most conditional tests:
 - ▣ `if (x < 12) ...`

Example 2 Constant time

```
int sum = 0, j;  
    for (j = 0; j < 100; j++)  
        sum = sum + j;
```

- Loop executes 100 times
- 4 steps per iteration = $O(1)$
- Total time is $100 * O(1) = O(100 * 1) = O(100) = O(1)$

Example 3 Linear time

```
void printAllElementOfArray(int arr[], int size){  
    for (int i = 0; i < size; i++) {  
        printf("%d\n", arr[i]);  
    }  
}
```

- This function runs in $O(n)$ time (or “linear time”) relative to its input, where n is the number of items in the array.
 - ▣ If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

Example 4 Quadratic time

```
void printAllPossibleOrderedPairs(int arr[], int size) {  
    for (int i = 0; i < size; i++) {  
        for (int j = 0; j < size; j++) {  
            printf("%d = %d\n", arr[i], arr[j]);  
        }  
    }  
}
```

- This function runs in $O(n^2)$ time (or “quadratic time”) relative to its input, where n is the number of items in the outer array and n for inner array.
 - If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

Example 5 Exponential time

```
int fibonacci(int num) {  
    if (num <= 1)  
        return num;  
    return fibonacci(num - 2) + fibonacci(num - 1);  
}
```

- An example of an $O(2^n)$ function is the recursive calculation of Fibonacci numbers.
- The growth curve of an $O(2^n)$ function is exponential.

Example 6 “Drop the Constants”

```
void printAllItemsTwice(int arr[], int size)
{
    for (int i = 0; i < size; i++) {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < size; i++) {
        printf("%d\n", arr[i]);
    }
}
```

- This is $O(2n)$, which we just call $O(n)$..

Example 7 “Drop the Constants”

```
void printItem100Times(int arr[], int size) {  
    printf("First element of array = %d\n", arr[0]);  
    for (int i = 0; i < size/2; i++) {  
        printf("%d\n", arr[i]);  
    }  
    for (int i = 0; i < 100; i++) {  
        printf("Hi\n");  
    }  
}
```

- This is $O(1 + n/2 + 100)$, which we just call $O(n)$.

Example 8 “Drop the less Significant Items”

```
void printAllNumbersAndPairSums(int arr[], int size) {  
    for (int i = 0; i < size; i++) {  
        printf("%d\n", arr[i]);  
    }  
    for (int i = 0; i < size; i++) {  
        for (int j = 0; j < size; j++) {  
            printf("%d\n", arr[i] + arr[j]);  
        }  
    }  
}
```

- Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$.

Example 8 “Drop the less Significant Items”

- Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$.

Similarly,

- $O(n^3 + 160n^2 + 50000)$ is $O(n^3)$
- $O((n + 40) * (n + 4))$ is $O(n^2)$

Example 9

```
bool arrayWithElement(int arr[], int size, int element) {  
    for (int i = 0; i < size; i++) {  
        if (arr[i] == element)  
            return true;  
    }  
    return false;  
}
```

- Generally, we say this is $O(n)$ runtime and the "worst case" part would be implied. But to be more specific we could say this is worst case $O(n)$ and best case $O(1)$ runtime.

Example 10

```
for (int i = 1; i <= n; i *= c)
{
    // some O(1) expressions
}
```

```
for (int i = n; i > 0; i /= c)
{
    // some O(1) expressions
}
```

□ $O(\log n)$

EFFICIENCY OF THE ALGORITHM



Efficiency of the Algorithm

n	$\log_2 n$	$n \log_2 n$	n^2	2^n
1	0	0	1	2
2	1	2	2	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	4,294,967,296

Growth rates of various functions

Efficiency of the Algorithm


Time for $f(n)$ instructions on a computer that executes 1 billion instructions per second

n	$f(n) = n$	$f(n) = \log_2 n$	$f(n) = n \log_2 n$	$f(n) = n^2$	$f(n) = 2^n$
10	0.01 μ s	0.003 μ s	0.033 μ s	0.1 μ s	1 μ s
20	0.02 μ s	0.004 μ s	0.086 μ s	0.4 μ s	1ms
30	0.03 μ s	0.005 μ s	0.147 μ s	0.9 μ s	1s
40	0.04 μ s	0.005 μ s	0.213 μ s	1.6 μ s	18.3min
50	0.05 μ s	0.006 μ s	0.282 μ s	2.5 μ s	13 days
100	0.10 μ s	0.007 μ s	0.664 μ s	10 μ s	4×10^{13} years
1000	1.00 μ s	0.010 μ s	9.966 μ s	1ms	
10,000	10 μ s	0.013 μ s	130 μ s	100ms	
100,000	0.10ms	0.017 μ s	1.67ms	10s	
1,000,000	1 ms	0.020 μ s	19.93ms	16.7m	
10,000,000	0.01s	0.023 μ s	0.23s	1.16 days	
100,000,000	0.10s	0.027 μ s	2.66s	115.7 days	

Efficiency of the Algorithm

Function $g(n)$	Growth rate of $f(n)$
$g(n) = 1$	The growth rate is constant and so does not depend on n , the size of the problem.
$g(n) = \log_2 n$	The growth rate is a function of $\log_2 n$. Because a logarithm function grows slowly, the growth rate of the function f is also slow.
$g(n) = n$	The growth rate is linear. The growth rate of f is directly proportional to the size of the problem.
$g(n) = n \log_2 n$	The growth rate is faster than the linear algorithm.
$g(n) = n^2$	The growth rate of such functions increases rapidly with the size of the problem. The growth rate is quadrupled when the problem size is doubled.
$g(n) = 2^n$	The growth rate is exponential. The growth rate is squared when the problem size is doubled.

Efficiency of the Algorithm

Complexity	Growth Rate	
$O(k) = O(1)$	Constant Time	 Increasing Complexity
$O(\log_b N) = O(\log N)$	Logarithmic Time	
$O(N)$	Linear Time	
$O(N \log N)$	Log Linear	
$O(N^2)$	Quadratic Time	
$O(N^c)$	Polynomial Time	
....		
$O(k^N)$	Exponential Time	

Efficiency of the Algorithm

□ It can be shown that,

$$O(1) \leq O(\log_2 n) \leq O(n) \leq O(n \log_2 n) \leq O(n^2) \leq O(2^n).$$

n dominates $\log_2(n)$

$(n \times \log_2(n))$ dominates n

n^2 dominates $(n \times \log_2(n))$

n^m dominates n^k when $m > k$

a^n dominates n^m for any $a > 1$ and $m \geq 0$

Reading Materials

- Mark Allen Weiss – Chapter#2
- Nell Dale: Chapter # 3 (Section 3.4)
- D. S. Malik – Chapter#1