SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act,1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

Course

CSE318 - Algorithm Design Strategies & Analysis Laboratory

Topic

Algorithm Analysis

Shri B. Srinivasan

Assistant Professor-III, School of Computing

SASTRA Deemed To Be University

Topics

- ✓ Syllabus
- ✓ Algorithm – Introduction
- ✓ Algorithms Design Model
- ✓ Asymptotic Notations
- ✓ Analysis of algorithms
  - ✓ Examples for complexity analysis

Syllabus

Unit - I

Introduction:

- ✓ Algorithm Specification
- ✓ Performance Analysis - Space Complexity, Time Complexity
- ✓ Asymptotic Notations - Time and Space Trade-Offs
- ✓ Analysis of Recursive Algorithms through Recurrence Relations
- ✓ Solving Recurrence Relations
  - ➢ Substitution Method
  - ➢ Recursion Tree Method

Fundamental Algorithmic Strategies:

- ✓ *Brute-Force Method* – Heuristics Method
  - ➢ Travelling Salesman Problem
- ✓ *Greedy Method* - General Method
  - ➢ Knapsack Problem
  - ➢ Job Sequencing with Deadlines

Unit – II

Advanced Strategies:

- ✓ *Dynamic Programming Method*
  - ➢ Optimal Binary Search Trees
  - ➢ String Editing
  - ➢ 0/1 Knapsack
  - ➢ Travelling Salesman problem
- ✓ *Branch and Bound Method*
  - ➢ 0/1 Knapsack Problem
  - ➢ Travelling Salesman Problem
- ✓ *Backtracking Method*
  - ➢ 8-Queens Problem
  - ➢ Sum of Subsets
  - ➢ Hamiltonian Cycles
  - ➢ Knapsack Problem

Unit – III

Graph and Tree Algorithms:

- ✓ *Traversal algorithms*
  - Breadth First Search
  - Depth First Search
  - Topological Sort
- ✓ *Minimum Spanning Tree*
  - Kruskal's Algorithm
  - Prim's Algorithm
- ✓ *Shortest path algorithms*
  - Bellman-Ford Algorithm
  - Dijkstra's Algorithm
  - Floyd-Warshall Algorithm
- ✓ *Flow Networks Problem*
  - Ford-Fulkerson Algorithm

Unit – IV

Tractable and Intractable Problems:

- ✓ Nondeterministic Algorithms
- ✓ The classes NP-Hard and NP-Complete
- ✓ Cook's Theorem
- ✓ Clique Decision Problem
- ✓ Node Cover Decision Problem -
- ✓ Travelling Salesperson Decision Problem.

Advanced Topics:

- ✓ *Approximation algorithms*
  - ➢ Scheduling Independent Tasks
  - ➢ Bin Packing
  - ➢ Interval Partitioning
- ✓ Randomized algorithms
- ✓ Class of problems beyond NP - P SPACE
- ✓ Introduction to Quantum Algorithms

Text Book

1. Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, Fundamental of Computer Algorithms, Computer Science Press, Second Edition, 2008.

2. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms, Prentice Hall of India, Third Edition, 2009. (Paperback-2011)

3. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. The Design and Analysis of Computer Algorithms, Pearson Education, 2003.

Introducing Algorithm

What is an Algorithm?

- ✓ What is an Algorithm?
- ✓ Representation: Procedure, Flowchart and Pseudocode
- ✓ Why do we need many algorithms for solving same problem?
- ✓ How to choose best algorithm for a problem?
- ✓ How to quantify the Efficiency of an algorithm?
- ✓ What is "Analysis of Algorithm"?
- ✓ Time complexity and Space Complexity

Algorithm: Procedure vs Pseudocode

Procedure findMax

Input: Three distinct numbers, number1, number2 and number 3.

Output: The biggest number among the given 3 numbers

Steps:

1. Compare number1 with number2 and number3. If it is greater than both number2 and number3, return number1 as biggest.
2. Compare number2 with number1 and number3. If it is greater than both number1 and number3, return number2 as biggest.
3. Compare number3 with number1 and number2. If it is greater than both number2 and number3, return number1 as biggest.

Algorithm: Procedure vs Pseudocode

```
Algorithm findMax(a,b,c)
    Input: Three numbers: a, b and c
    Output: The biggest number
    if a>b and a>c then
            return a
    endif
    if b>a and b>c then
            return b
    endif
    if c>a and c>b then
            return c
    endif
end findMax
```

Pseudocode

```
int findMax(int a, int b, int c)
{
        if(a>b && a>c)
        {
                return a;
        }
        if(b>a && b>c)
        {
                return b;
        }
        if(c>a && c>b)
        {
                return c;
        }
}
```

Code in C Language

How to quantify Efficiency of an Algorithm?

Three different Algorithms for the same problem: *To find the maximum among 3 distinct numbers*

Which is the best?

```
int findMax(int a, int b, int c)
{
        if(a>b && a>c)
        {
                return a;
        }
        if(b>a && b>c)
        {
                return b;
        }
        if(c>a && c>b)
        {
                return c;
        }
}
```

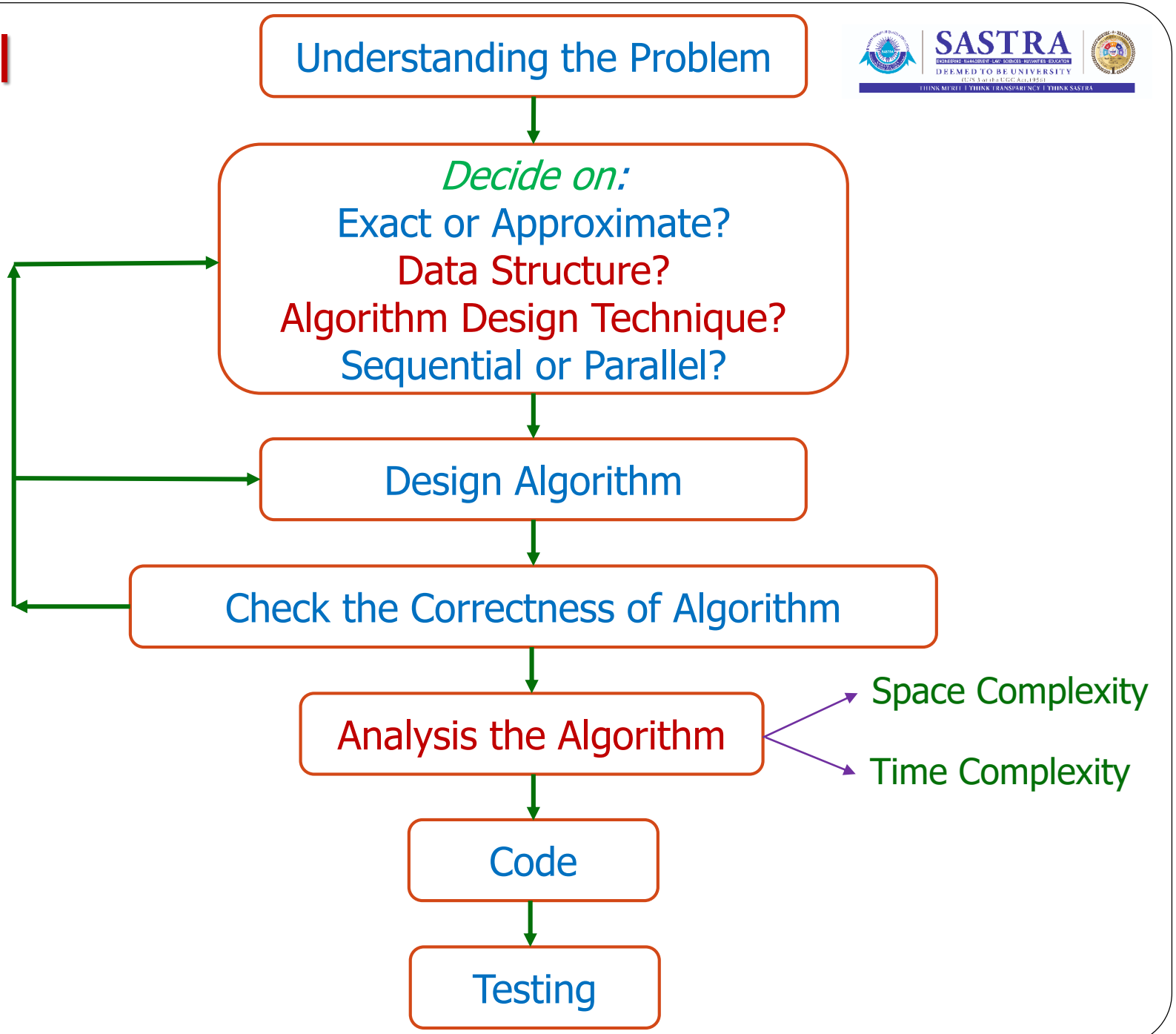
```
int findMax(int a, int b, int c)
{
        if(a>b && a>c)
        {
                return a;
        }
        else
        {
                if(b>c)
                {
                        return b;
                }
                else
                {
                        return c;
                }
        }
}
```

```
int findMax(int a, int b, int c)
{
        if(a>b)
        {
                if(a>c)
                {
                        return a;
                }
                else
                {
                        return c
                }
        }
        else
        {
                if(b>c)
                {
                        return b;
                }
                else
                {
                        return c;
                }
        }
}
```

Algorithm Design Model

Algorithm Design Model

1. What kind of Data Structure to be Choosed?
   - *Stack*
   - *Queue*
   - *List*
   - *Tree*
   - *Graph*
2. What kind of Algorithm Design Technique to be used to solve the problem?
   - *Bruteforce*
   - *Divide & Conquer*
   - *Dynamic Programming*
   - *Greedy*
   - *Backtracking*
   - *Branch and Bound, etc.*
3. What is the Efficiency of algorithm? – Need to Analysis of Algorithm:
   - *Time Complexity*
   - *Space Complexity*



Asymptotic Notations – For Representing Time Complexity

Asymptotic Notations

- ✓ Mathematical Model – To represent the Time Complexity of an algorithm
- ✓ Time Complexity – Also known as Rate of Growth of an algorithm
- ✓ Asymptotic Notation – A relation between two function f(n) an g(n)
- ✓ It is a formal notation for discussing and analysing the "Classes of Functions"
- ✓ Example:

Let two functions, $f(n) = 4n^2+3n+5$ and $g(n) = n^2$

Then, $4n^2+3n+5 \in O(n^2)$

$4n^2+3n+5 \in \Omega(n^2)$

$4n^2+3n+5 \in \theta(n^2)$

Rate of Growth		
Input Size	Time	
n	Algorithm 1	Algorithm 2
1	1	1
10	20	10
100	5000	100
1000	500000	1000
Best?		✓

Asymptotic Notations

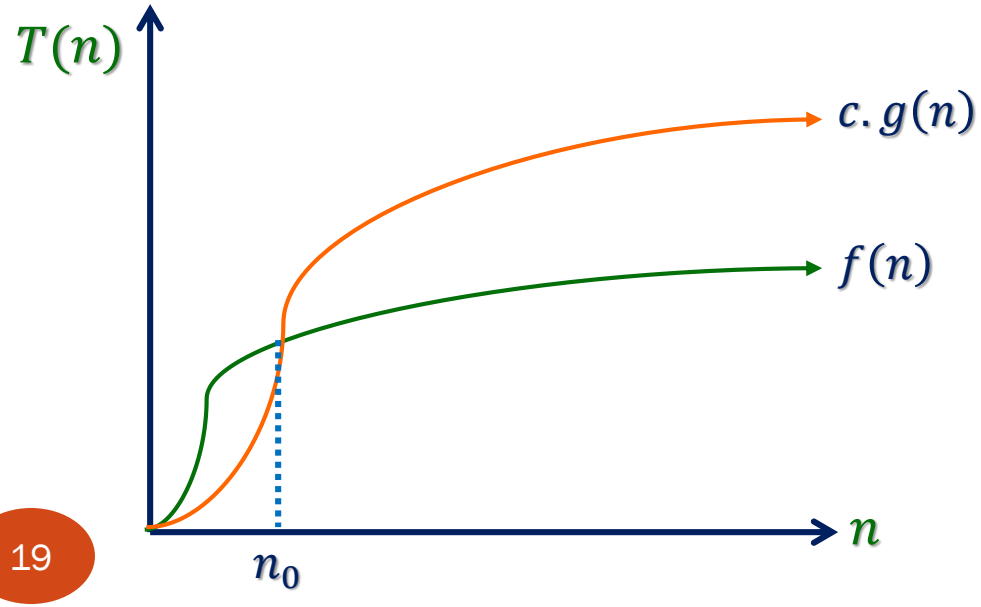
- ✓ Tight Bound
  - Big-Oh [O] – Define Upper Bound
  - Big-Omega [Ω] – Define Lower Bound
  - Theta [θ] – Define Both Upper and Lower Bound
- ✓ Loose Bound
  - Small-Oh or Little-Oh [o]
  - Small-Omega or Little-Omega [ω]

Note: Usually Big-Oh [O] and Theta [θ] notations are used for representing time complexity of an algorithm

Big-Oh [O]

A function f(n) is said to be in Big-Oh, O(g(n)), denoted as f(n) ∈ O(g(n)) or simply, f(n) = O(g(n)), if f(n) is bounded ABOVE by some +ve constant multiples of g(n) for all large 'n', if there exist some +ve constants c, and some non-negative integer n_0

$$0 \leq f(n) \leq c.g(n), \forall n \geq n_0$$



Example

Let two functions, $f(n) = 4n^2+3n+5$ and $g(n) = n^2$, Then, prove that $4n^2+3n+5 \in O(n^2)$

Solution:

- Let $f(n) = 4n^2+3n+5$.
- Rise the terms '3n' and '5' to n^2
- Need to multiply by n and n^2 respectively
- $f(n) = 4n^2+3n+5 \leq 4n^2+3n^2+5n^2$
- $f(n) \leq 4n^2+3n^2+5n^2 \rightarrow f(n) \leq 12n^2$
- $f(n) \leq 12.\ g(n)$, where c = 12.
- Need to prove this is true for large n values.
- Need to verify for n=1,2,3, and so on.,

n=1 → $4(1)^2+3(1)+5 \leq 12.\ (1)^2$ → $12 \leq 12$ ✓

n=2 → $4(2)^2+3(2)+5 \leq 12.\ (2)^2$ → $27 \leq 48$ ✓

n=3 → $4(3)^2+3(3)+5 \leq 12.\ (3)^2$ → $50 \leq 108$ ✓

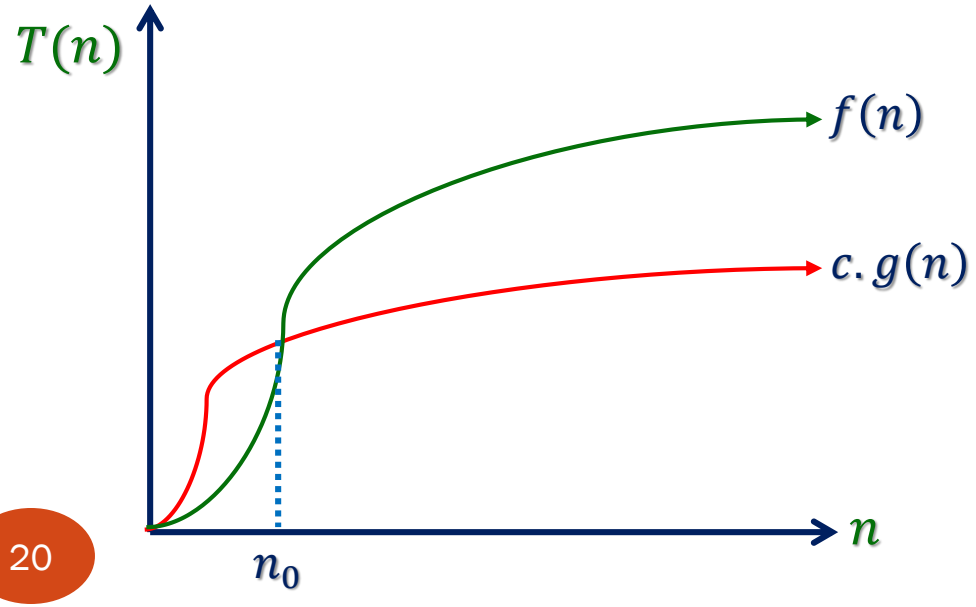
So, $f(n) \leq c.g(n)$ is True for c=12 and n_0=1

Hence proved, $4n^2+3n+5 \in O(n^2)$

Big-Omega [Ω]

A function f(n) is said to be in Big-Omega, Ω(g(n)), denoted as f(n) ∈ Ω(g(n)) or simply, f(n) = Ω(g(n)), if f(n) is bounded BELOW by some +ve constant multiples of g(n) for all large 'n', if there exist some +ve constants c, and some non-negative integer n_0

$$0 \geq f(n) \geq c.g(n), \forall n \geq n_0$$



Example

Let two functions, $f(n) = 4n^2+3n+5$ and $g(n) = n^2$, Then, prove that $4n^2+3n+5 \in \Omega(n^2)$

Solution:

- Let $f(n) = 4n^2+3n+5$.
- Reduce the terms '3n' and '5' to 0
- Need to remove these terms, so we can write,
- $f(n) = 4n^2+3n+5 \geq 4n^2$
- $f(n) \geq 4n^2 \rightarrow f(n) \geq 4.\ g(n)$, where c = 4.
- Need to prove this is true for large n values.
- Need to verify for n=1,2,3, and so on.,

n=1 → $4(1)^2+3(1)+5 \geq 4.\ (1)^2$ → $12 \geq 4$ ✓

n=2 → $4(2)^2+3(2)+5 \geq 4.\ (2)^2$ → $27 \geq 16$ ✓

n=3 → $4(3)^2+3(3)+5 \geq 4.\ (3)^2$ → $50 \geq 36$ ✓

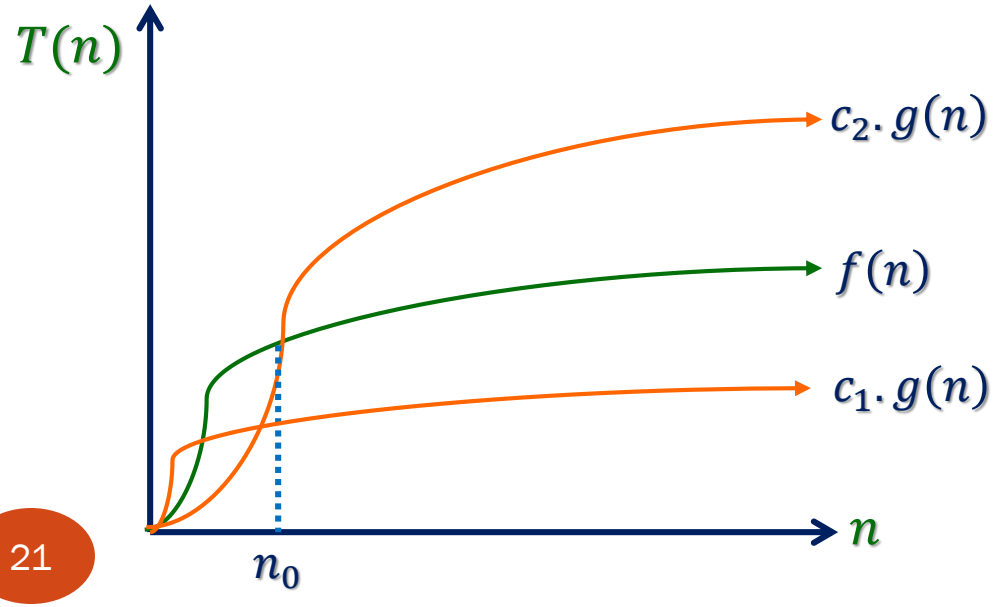
So, $f(n) \geq c.g(n)$ is True for c=4 and n_0=1

Hence proved, $4n^2+3n+5 \in \Omega(n^2)$

Theta [θ]

A function f(n) is said to be in Theta, θ(g(n)), denoted as f(n) ∈ θ(g(n)) or simply, f(n) = θ(g(n)), if f(n) is bounded BELOW and ABOVE by some +ve constant multiples of g(n) for all large 'n', if there exist some +ve constants c_1 and c_2, and some non-negative integer n_0

$$c_1.g(n) \leq f(n) \leq c_2.g(n), \forall n \geq n_0$$



Example

Let two functions, f(n) = $4n^2+3n+5$ and g(n) = n^2 .
Then, prove that $4n^2+3n+5$ ∈ θ (n^2)

Solution:

- If f(n) ∈ O(g(n)) and f(n) ∈ Ω(g(n)), then f(n) ∈ θ(g(n))
- We already proved f(n) ∈ O(g(n)) and f(n) ∈ Ω(g(n))
- So, f(n) ∈ θ(g(n))

So, $c_1.g(n) \leq f(n) \geq c_2.g(n)$ is True for

$c_1=4$, $c_2=12$ and $n_0=1$

Hence proved, $4n^2+3n+5$ ∈ Ω(n^2)

Small-Oh or Little-Oh [o]

A function f(n) is said to be in Small-Oh, o(g(n)), denoted as f(n) ∈ o(g(n)) or simply, f(n) = o(g(n)), if

$$0 \leq f(n) < c.g(n), \forall n \geq n_0$$

Small-Omega or Little-Omega [ω]

A function f(n) is said to be in Small-Omega, ω(g(n)), denoted as f(n) ∈ ω(g(n)) or simply, f(n) = ω(g(n)), if

$$0 \geq f(n) > c.g(n), \forall n \geq n_0$$

Rate of Growth – Comparing Algorithms

Rate of Growth

- ✓ Constant – $O(1)$ *or* $O(c)$
- ✓ Linear – $O(n)$
- ✓ Quadratic – $O(n^2)$
- ✓ Cubic – $O(n^3)$
- ✓ Polynomial – $O(n^d),\ d > 1$
- ✓ Logarithmic – $O(\log n)$
- ✓ Poly-logarithmic - $O(n^d \log n),\ d \geq 1$
- ✓ Fractional Power – $O(n^c),\ 0 < c < 1$
- ✓ Constant Power – $O(c^n),\ \ c > 1, 2^n, 3^n, 4^n, \ldots$
- ✓ Factorial – $O(n!)$

How to Compare Algorithms?

- ✓ If the time complexity of algorithm is given, we can rank the algorithm based on their efficiency in terms of time complexity.
- ✓ Substitute some large n value. Based on result we can rank.

Example:

Algorithm	Time Complexity	Input Size - n			Rank
		8	256	1000	
Algorithm 1	O(n)	8	256	1000	4
Algorithm 2	O(log n)	3	8	9.96578428	2
Algorithm 3	$O(n^2)$	64	65536	1000000	6
Algorithm 4	O(n log n)	24	2048	9965.78428	5
Algorithm 5	O(n!)	40320	Infinity	Infinity	7
Algorithm 6	O(1)	1	1	1	1
Algorithm 7	$O(\sqrt{n})$	2.82843	16	31.6227766	3

Analyzing Algorithms - Examples

How to Analyze Algorithm?

- ✓ Time Complexity – Rate of Growth of an algorithm
- ✓ It depends on "*How many operations are going to be executed*?"
- ✓ So, we need to count the number of operations to be performed.
- ✓ This count is usually depends on the input size 'n' for many algorithms.
- ✓ By counting the number of operations, it will give an polynomial expression in terms of input size n.
- ✓ Then, this polynomial function will be expressed as Big-Oh or Theta classes of functions.
- ✓ For example, if we get $4n^2+3n+5$ as the count, we can say that, this function is belongs to $O(n^2)$

Example:

Sum of N numbers

```
Algorithm Sum(A[1..n])
        S ← 0                          ----------> 1
        For i←1 to n do                ----------> (n-1+1)+1 = n+1
                S ← S + A[i]           ----------> n
        End For
        return S                       ----------> 1
End Sum
```

$$T(n) = 1 + n + 1 + n + 1 = 2n + 3 \in O(n)$$

$$T(n) = O(n)$$

Note: To find the number of iterations in a 'for' loop, we may use the formula, upper-lower+1

Some Observations...

- ✓ To find time complexity, we need to *count the number of operations*
- ✓ The number of operations mainly depends on the ACTIVE operations.
- ✓ Active operation is an operation which is executed more often in an algorithm
- ✓ So, most probably, the active operation is operation present in the iteration.
- ✓ So, we may count the number of iterations to be executed, if the algorithm containing iterative statements.
- ✓ Mostly, the number of iterations are depends on input size.
- ✓ So, the time complexity is represented in terms of input size.

Problem 1

```
for(i=1; i<=n; i++)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \sum_{i=1}^{n} 1 = n - 1 + 1 = n \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 2

```
for(i=n; i>=1; i--)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \sum_{i=1}^{n} 1 = n - 1 + 1 = n \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 3

```
for(i=1; i<=n; i=i+2)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \frac{\sum_{i=1}^{n} 1}{2} = \frac{n-1+1}{2} = \frac{n}{2} \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 4

```
for(i=1; i<=n; i=i*2)
{
        O(1) Statements…
}
```

Solution

$$T(n) = [i \leq n] = [2^{k-1} \leq n] = [k - 1 \leq \log_2 n] \Rightarrow k = \log_2 n + 1$$

Iteration	i
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
4	$8 = 2^3$
.	
.	
k	2^{k-1}

$$T(n) = O(\log_2 n)$$

Problem 5

```
for(i=n; i>=1; i=i/2)
{
        O(1) Statements…
}
```

Solution

$$T(n) = [i \geq 1] = \left[\frac{n}{2^{k-1}} \geq n\right] = [n \geq 2^{k-1}] = [\log_2 n \geq k-1]$$

$$T(n) = [k-1 \leq \log_2 n] \Rightarrow k = \log_2 n + 1$$

Iteration	i
1	$n = n / 2^0$
2	$n/2 = n / 2^1$
3	$n/4 = n / 2^2$
4	$n/8 = n / 2^3$
.	
.	
k	$n / 2^{k-1}$

$$T(n) = O(\log_2 n)$$

Problem 6

```
for(i=1, P=1; P<=n; i++)
{
        P = P + i;
}
```

Solution

$$T(n) = [P \leq n] = [k^2 \leq n] = [k \leq \sqrt{n}] \Rightarrow k \in O(\sqrt{n})$$

Iteration	P=P+i
1	1 = 1
2	3 = 1 + 2
3	6 = 1 + 2 + 3
4	10 = 1 + 2 + 3 + 4
.	
.	
k	$1+2+3+\ldots+k = k(k+1)/2 = O(k^2)$

$$T(n) = O(\sqrt{n})$$

Problem 7

```
for(i=1; i*i<=n; i++)
{
        O(1) Statements…
}
```

Solution

$$T(n) = [i^2 \leq n] = [k^2 \leq n] = [k \leq \sqrt{n}] \Rightarrow k \in O(\sqrt{n})$$

Iteration	i*i
1	1
2	$2*2 = 2^2$
3	$3*3 = 3^2$
4	$4*4 = 4^2$
.	
.	
k	$k*k = k^2$

$$\boldsymbol{T(n) = O(\sqrt{n})}$$

Problem 8

```
for(i=1; i<=n; i++)
{
        O(1) Statements…
}
for(j=1; j<=n; j++)
{
        O(1) Statements…
}
```

Solution

$$T(n) = \sum_{i=1}^{n} 1 + \sum_{j=1}^{n} 1$$

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

$$T(n) = 2n \in O(n)$$

$$\boldsymbol{T(n) = O(n)}$$

Problem 9

```
for(i=1; i<=n; i++)
{
        for(j=1; j<=n; j++)
        {
                O(1) Statements…
        }
}
```

Solution

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n - 1 + 1$$

$$T(n) = \sum_{i=1}^{n} n = n \,.\sum_{i=1}^{n} 1$$

$$T(n) = n\,.(n - 1 + 1) = n * n$$

$$T(n) = n * n \in O(n^2)$$

$$\boldsymbol{T(n) = O(n^2)}$$

Problem 10

```
for(i=1; i<=n; i++)
{
        for(j=1; j<=i; j++)
        {
                O(1) Statements…
        }
}
```

Solution

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{i} 1 = \sum_{i=1}^{n} i - 1 + 1$$

$$T(n) = \sum_{i=1}^{n} i = 1 + 2 + \cdots n$$

$$T(n) = n \,.\, \frac{(n+1)}{2}$$

$$T(n) = \frac{1}{2}(n^2 + n) \in O(n^2)$$

$$\boldsymbol{T(n) = O(n^2)}$$

Problem 11

```
for(i=1; i<=n; i++)
{
        for(j=1; j<=n; j++)
        {
                for(k=1;k<n;k++)
                {
                        O(1) Statements…
                }
        }
}
```

Solution

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 1 = \sum_{i=1}^{n}\sum_{j=1}^{n} n - 1 + 1$$

$$T(n) = \sum_{i=1}^{n}\sum_{j=1}^{n} n = n \,.\sum_{i=1}^{n}\sum_{j=1}^{n} 1$$

$$T(n) = n \,.\sum_{i=1}^{n} n = n.n.\sum_{i=1}^{n} 1 \,.= \; n^3$$

$$T(n) = n * n * n \in O(n^3)$$

$$\boldsymbol{T(n) = O(n^3)}$$

How to analyze algorithm?

If the number of Iterations depends on
Not only the input size, but also the input values

Example: Linear Search

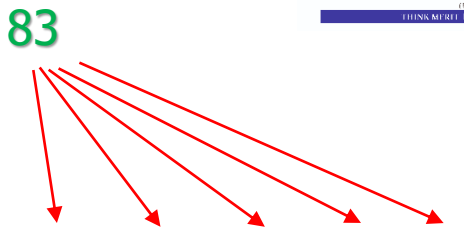
Example:

Linear Search

```
Algorithm Search(A[0..n-1], n,  Key)
        For i←0 to n-1 do
                If Key = A[i] Then
                        Return True
                End IF
        End For
        Return False
End Search
```

- ✓ In this example, the exact number of iterations is not known in advance.
- ✓ It depends on, "where the key element is present in the array?"

Example:



Consider an array A = [29, 45, 12, 57, 83]

Case 1: [If Key is 29]

In this case, since the key is present in the first location, the algorithm need only one iteration to give the result.

Case 2: [If Key is 83 OR 25]

If key is present in last location OR the key is not present in the array.

In this case, the algorithm needs 'n' iterations to obtain the result.

Worst, Best and Average Case Complexity

When the number of iterations depends on value of the input, we need to analyse the algorithm in three different cases.

Best Case Complexity: [Case 1 in above example]

It is the time complexity of the algorithm for the BEST CASE INPUT.

Worst Case Complexity: [Case 2 in above example]

It is the time complexity of the algorithm for the WORST CASE INPUT.

Average Case Complexity:

This can be obtained by considering probability of input elements.

For Linear Search Algorithm:

$$T_{Best}(n) = O(1)$$

$$T_{Worst}(n) = O(n)$$

$$T_{Average}(n) = O(n)$$

Insertion Sort – Algorithm, Example & Analysis

Insertion Sort – Algorithm

```
Algorithm InsertionSort(a[0..n-1], n)
        for i=1 to n-1 do
                Key = a[i]
                j = i-1
                while j>=0 and Key<a[j] do
                        a[j+1] = a[j]
                        j = j-1
                end while
                a[j+1] = Key
        end for
end InsertionSort
```

Insertion Sort – Example (Tracing Algorithm)

Iteration-1 (i=1)

Key = a[i]

Key
23

		j=0	j=-1	a[j+1] = Key
0	56	56	Stop	23
1	23	56		56
2	76	76		76
3	34	34		34
4	65	65		65
5	87	87		87
6	20	20		20
7	45	45		45

Iteration-2 (i=2)

Key

Key
76

76<56, false, stop the iteration

		j=1	j=0	j=-1	a[j+1] = Key
0	23	23		Stop	23
1	56	56			56
2	76	76			76
3	34	34			34
4	65	65			65
5	87	87			87
6	20	20			20
7	45	45			45

Stop

Insertion Sort – Example (Tracing Algorithm)

Key

34

Iteration-3 (i=3)

34<76 34<56 34<23

	j=2	j=1	j=0	j=-1
0	23	23	23	23	Stop
1	56	56	56	56	
2	76	76	56	56	
3	34	76	76	76	
4	65	65	65	65	
5	87	87	87	87	
6	20	20	20	20	
7	45	45	45	45	

Stop

a[j+1] = Key

23
34
56
76
65
87
20
45

Insertion Sort – Example (Tracing Algorithm)

Key

65

Iteration-4 (i=4)

65<76 65<56

		j=3	j=2	j=1	j=0	j=-1	a[j+1] = Key
0	23	23	23			Stop	23
1	34	34	34				34
2	56	56	56				56
3	76	76	76				65
4	65	76	76				76
5	87	87	87				87
6	20	20	20				20
7	45	45	45				45

Stop

Insertion Sort – Example (Tracing Algorithm)

Key

87

Iteration-5 (i=5)

87<76

		j=4	j=3	j=2	j=1	j=0	j=-1
0	23	23					Stop
1	34	34					
2	56	56					
3	65	65					
4	76	76					
5	87	87					
6	20	20					
7	45	45					
		Stop					

a[j+1] = Key

23	0
34	1
56	2
65	3
76	4
87	5
20	6
45	7

Insertion Sort – Example (Tracing Algorithm)

Iteration-6 (i=6)

Key: 20

20<87 20<76 20<65 20<56 20<34 20<23

j=5	j=4	j=3	j=2	j=1	j=0	j=-1
23	23	23	23	23	23	23	Stop
34	34	34	34	34	34	23	
56	56	56	56	56	34	34	
65	65	65	65	56	56	56	
76	76	76	65	65	65	65	
87	87	76	76	76	76	76	
20	87	87	87	87	87	87	
45	45	45	45	45	45	45	

a[j+1] = Key

a[j+1] = Key
20
23
34
56
65
76
87
45

Insertion Sort – Example (Tracing Algorithm)

Key = a[i]

Iteration-7 (i=7) | 45

45<87 45<76 45<65 45<56 45<34

	j=6	j=5	j=4	j=3	j=2	j=1	j=0	j=-1	a[j+1] = Key
0	20	20	20	20	20	20			Stop	20
1	23	23	23	23	23	23				23
2	34	34	34	34	34	34				34
3	56	56	56	56	56	56				45
4	65	65	65	65	56	56				56
5	76	76	76	65	65	65				65
6	87	87	76	76	76	76				76
7	45	87	87	87	87	87				87
					Stop				

Insertion Sort – Analysis of Algorithm

Insertion Sort - Analysis:

Alg. InsertionSort(A[0..n-1], n)

```
(1) ← for i ← 1 to n-1 do
(2) ←     key ← A[i]
(3) ←     j ← i-1
(4) ←     while j ≠ -1 and A[j] > key do
(5) ←         A[j+1] ← A[j]
(6) ←         j ← j-1
          end while.
(7) ←     A[j+1] ← key
    end for
end Insertion Sort
```

Total 7 Statements.

(1) ⇒ [(n-1) + 1 + 1] + 1 ⇒ n-1+1 = n

(2) ⇒ (n-1) + 1 + 1 ⇒ n-1

(3) ⇒ " ⇒ n-1

(7) ⇒ " ⇒ n-1

For Step (4):

- No. of Comparisons is not known.
- It depends on 2 Conditions [i ≠ 1 and A[i] > key]
- i.e., The no. of Compaisons is depending on the VALUE of input array. i.e., Not only depending on size of input.

Insertion Sort – Analysis of Algorithm

So, Let the no. of comparisons Required in Iteration No. 1 as t_1

i.e.,

When,

$i = 1$, No. of comparison $= t_1$

$i = 2$, " $= t_2$

$i = 3$, " $= t_3$

⋮

$i = n-1$, " $= t_{n-1}$

$$\text{Step } ④ \doteq \sum_{i=1}^{n-1} t_i$$

Step ⑤ (Body of the loop) & ⑥

Top ②

When $i=1$, if ④ took t_1 comparisons

⑤ & ⑥ will take $t_1 - 1$ times.

i.e., In general,

$$⑤ \& ⑥ = \sum_{i=1}^{n-1} (t_i - 1)$$

Now. need to find total no. of operations required for step ① to step ⑦ execution.

$$T(n) = n + 3(n-1) + \sum_{i=1}^{n-1} t_i + 2 \cdot \sum_{i=1}^{n-1} (t_i - 1)$$

Insertion Sort – Analysis of Algorithm

$$T(n) = 2 \cdot \sum_{i=1}^{n-1} (ti-1) + \sum_{i=1}^{n-1} ti + 4n - 3 \quad \rightarrow (1)$$

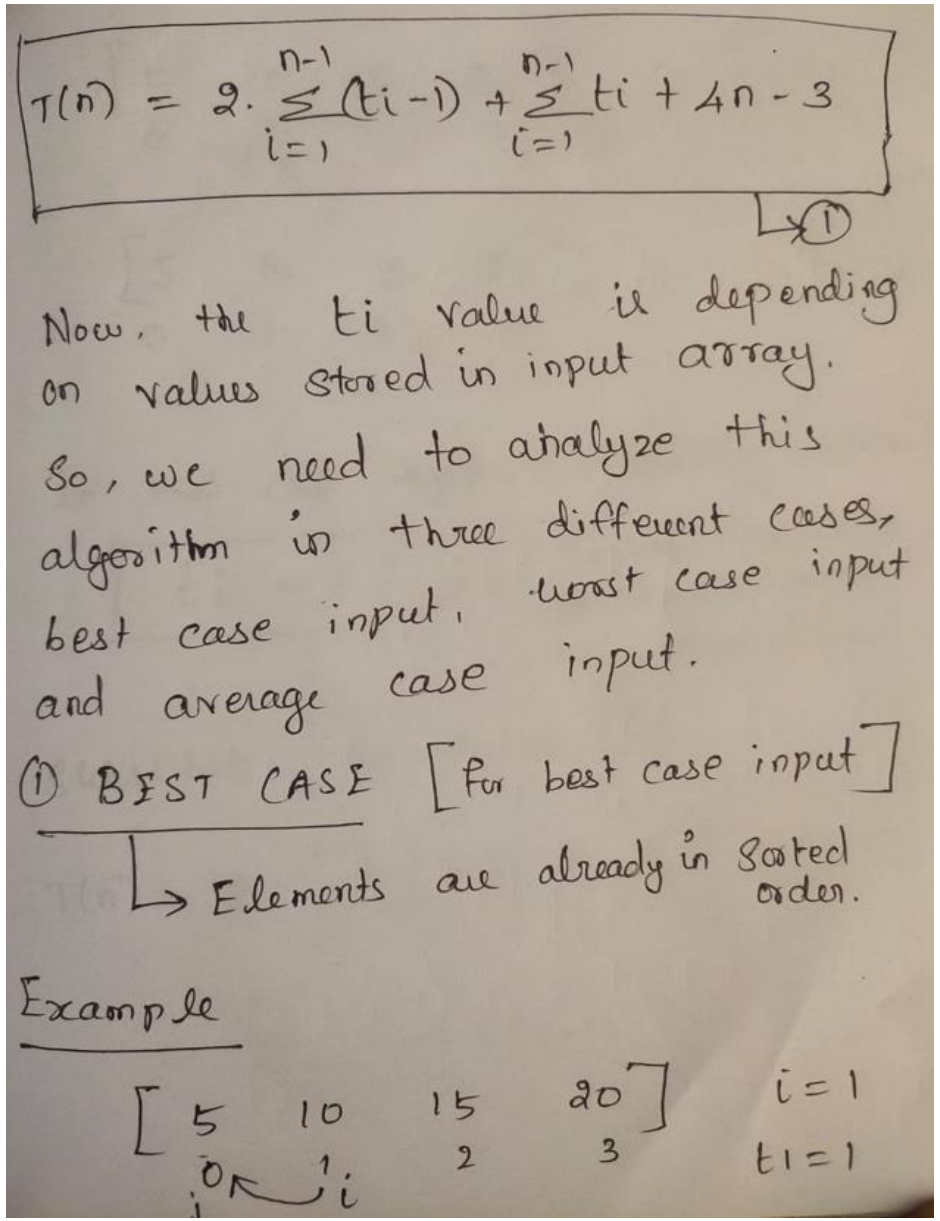
Now, the ti value is depending on values stored in input array.

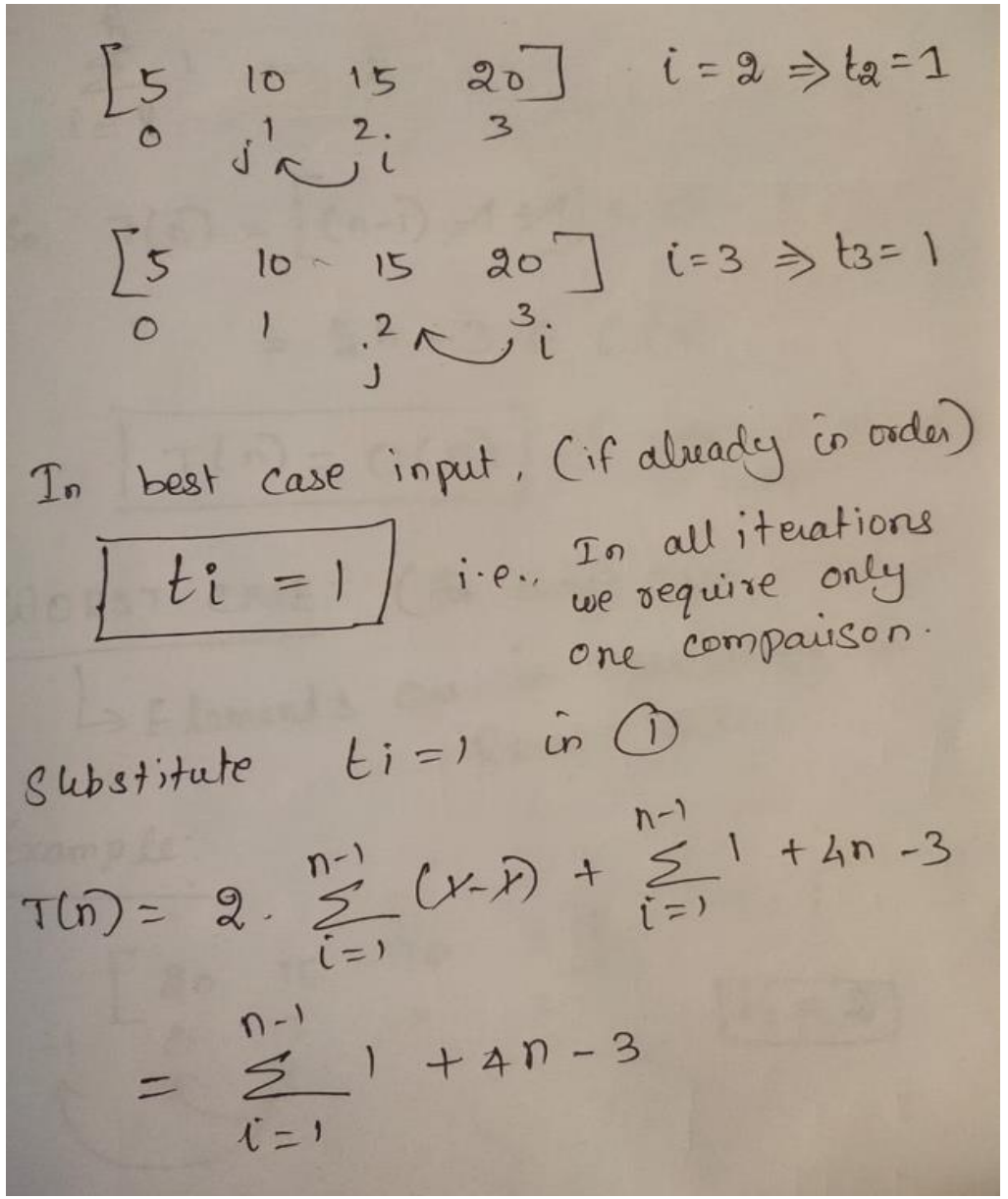
So, we need to analyze this algorithm in three different cases, best case input, worst case input and average case input.

① BEST CASE [For best case input]

↳ Elements are already in sorted order.

Example





In best case input, (if already in order)

$ti = 1$ i.e., In all iterations we require only one compaison.

Substitute $ti = 1$ in (1)

$$T(n) = 2 \cdot \sum_{i=1}^{n-1} (1-1) + \sum_{i=1}^{n-1} 1 + 4n - 3$$

$$= \sum_{i=1}^{n-1} 1 + 4n - 3$$

Insertion Sort – Analysis of Algorithm

$\sum_{i=l}^{h} 1 = h - l + 1$

So, $T(n) = [(n-1) \times 1 + 1] + 4n - 3$

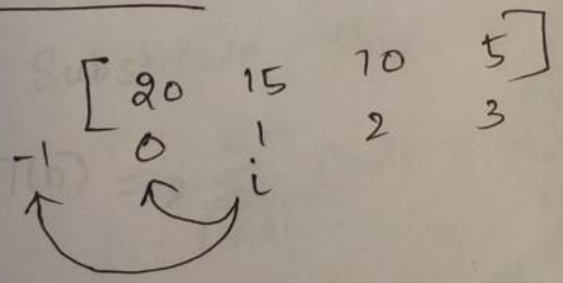
$= 5n - 3 \in O(n)$

$\boxed{T(n) = O(n)}$ if input is already in order.

② WORST CASE: (For worst case input)

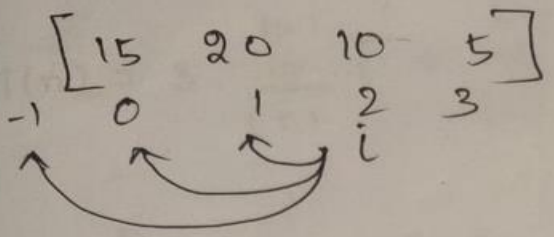
↳ Elements are in Completely in Reverse order.

Example:



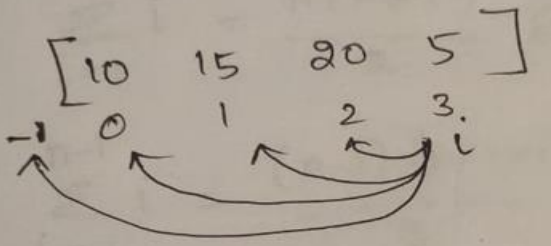
$i = 1$

$\boxed{t_1 = 2}$



$i = 2$

$\boxed{t_2 = 3}$



$i = 3$

$\boxed{t_3 = 4}$

[5 10 15 20]

In general, At i^{th} iteration,

$\boxed{t_i = i + 1}$

Substitute this in (1)

$T(n) = 2\sum_{i=1}^{n-1}(i+1) - 1 + \sum_{i=1}^{n-1} i + 1 + 4n - 3$

Insertion Sort – Analysis of Algorithm