

Unit 2

Asymptotic notations and Mathematic Analysis of algorithms

Topics Covered

Asymptotic Notations and Basic Efficiency classes - Informal Introduction - O-notation – Ω -notation - Θ -notation - mathematical analysis of non-recursive algorithms - mathematical analysis of recursive algorithms.

Asymptotic Notations :

- Asymptotic notation is a notation, which is used to take meaningful statement about the efficiency of a program.
- The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency.
- To compare and rank such orders of growth, computer scientists use three notations, they are:

- O - Big oh notation
- Ω - Big omega notation
- Θ - Big theta notation

Let $t(n)$ and $g(n)$ can be any nonnegative functions defined on the set of natural numbers. The algorithm's running time $t(n)$ usually indicated by its basic operation count $C(n)$, and $g(n)$, some simple function to compare with the count.

Example 1:

$$n \in O(n^2), \quad 100n + 5 \in O(n^2), \quad \frac{1}{2}n(n-1) \in O(n^2).$$

$$n^3 \notin O(n^2), \quad 0.00001n^3 \notin O(n^2), \quad n^4 + n + 1 \notin O(n^2).$$

$$n^3 \in \Omega(n^2), \quad \frac{1}{2}n(n-1) \in \Omega(n^2), \quad \text{but } 100n + 5 \notin \Omega(n^2).$$

, where $g(n) = n^2$.

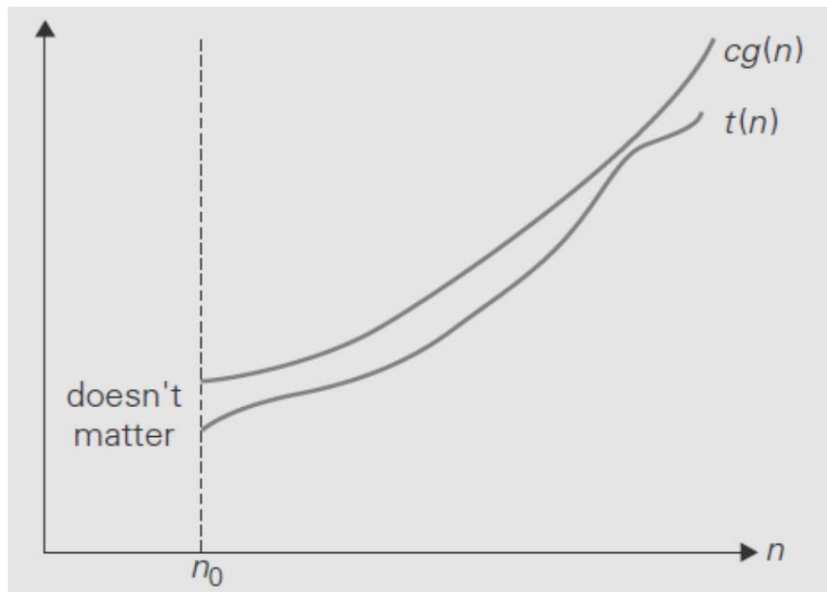
(i) O - Big oh notation :

A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \leq c \cdot g(n), \text{ for all } n \geq n_0$$

Where , $t(n)$ and $g(n)$ are non-negative functions defined on the set of natural numbers.

O = Asymptotic upper bound = Useful for worst case analysis = Loose bound



Example 2: Prove the assertions $100n + 5 \in O(n^2)$

$$\begin{aligned} \text{Proof: } 100n + 5 &\leq 100n + n \text{ (for all } n \geq 5) \\ &= 101n \\ &\leq 101n^2 \text{ (}\because n \leq n^2\text{)} \end{aligned}$$

Since, the definition gives us a lot of freedom in choosing specific values for constants c and n_0 . We have $c = 101$ and $n_0 = 5$

Example 3: Prove the assertions $100n + 5 \in O(n)$

$$\begin{aligned} \text{Proof: } 100n + 5 &\leq 100n + 5n \text{ (for all } n \geq 1) \\ &= 105n \\ \text{i.e., } 100n + 5 &\leq 105n \\ \text{i.e., } t(n) &\leq c.g(n) \end{aligned}$$

$\therefore 100n + 5 \in O(n^2)$ with $c=105$ and $n_0=1$

(ii) Ω - Big omega notation :

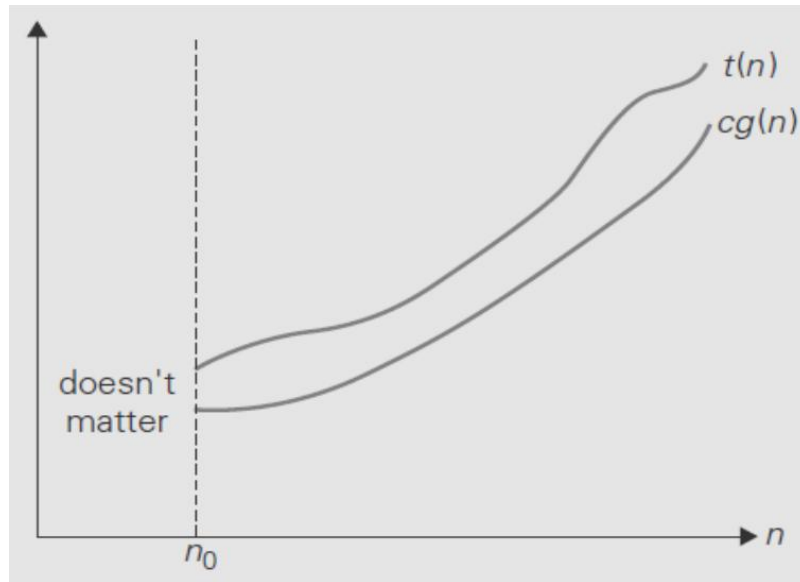
A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n ,

i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq c.g(n), \text{ for all } n \geq n_0$$

Where , $t(n)$ and $g(n)$ are non-negative functions defined on the set of natural numbers.

Ω = Asymptotic lower bound = Useful for best case analysis = Loose bound



Example 4: Prove the assertions $n^3 + 10n^2 + 4n + 2 \in \Omega(n^2)$.

Proof: $n^3 + 10n^2 + 4n + 2 \geq n^2$ (for all $n \geq 0$)

i.e., by definition $t(n) \geq cg(n)$, where $c=1$ and $n_0=0$

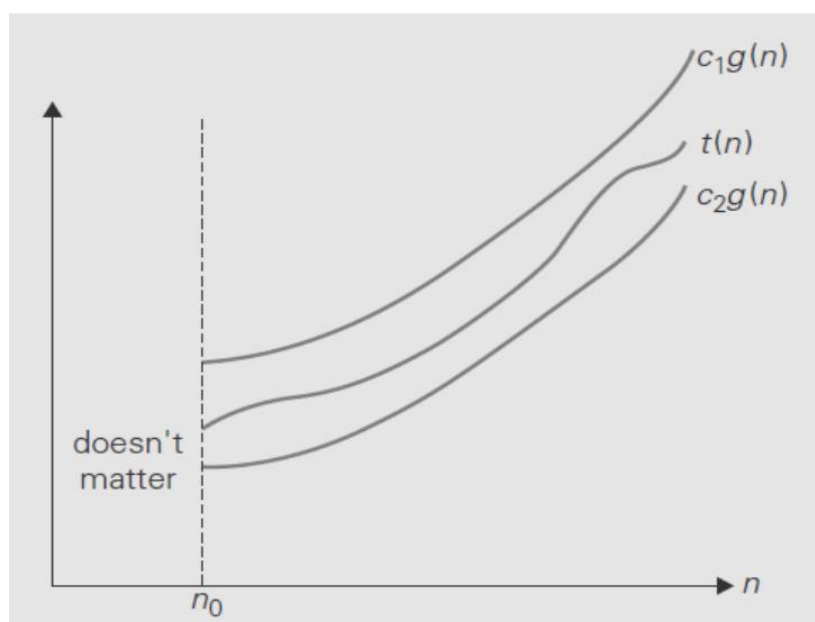
(ii) Θ - Big theta notation :

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n , i.e., if there exist some positive constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2g(n) \leq t(n) \leq c_1g(n) \text{ for all } n \geq n_0.$$

Where $t(n)$ and $g(n)$ are nonnegative functions defined on the set of natural numbers.

Θ = Asymptotic tight bound = Useful for average case analysis



MATHEMATICAL ANALYSIS FOR NON-RECURSIVE ALGORITHMS :

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation (in the innermost loop).
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. Using standard formulas and rules of sum manipulation either find a closed form formula for the count or at the least, establish its order of growth.

EXAMPLE 1: Consider the problem of finding the value of **the largest element in a list of n numbers**. Assume that the list is implemented as an array for simplicity.

ALGORITHM MaxElement($A[0..n - 1]$)

//Determines the value of the largest element in a given array.

//**Input:** An array $A[0..n - 1]$ of real numbers.

//**Output:** The value of the largest element in A .

maxval $\leftarrow A[0]$

for $i \leftarrow 1$ **to** $n - 1$ **do**

if $A[i] > \text{maxval}$

 maxval $\leftarrow A[i]$

return maxval

Algorithm analysis

- The measure of an input's size here is the number of elements in the array, i.e., n .
- There are two operations in the for loop's body:
 - The comparison $A[i] > \text{maxval}$ and
 - The assignment $\text{maxval} \leftarrow A[i]$.
- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n ; therefore, there is no need to distinguish among the worst, average, and best cases here.

- Let $C(n)$ denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and $n - 1$, inclusive.
- Therefore, the sum for $C(n)$ is calculated as follows:

$$c(n) = \sum_{i=1}^{n-1} 1$$

i.e., Sum up 1 in repeated $n-1$ times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Basic rules of sum manipulation

$$\sum_{i=l}^u c a_i = c \sum_{i=l}^u a_i, \quad (\text{R1})$$

$$\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i, \quad (\text{R2})$$

Summation formulas

$$\sum_{i=l}^u 1 = u - l + 1 \quad \text{where } l \leq u \text{ are some lower and upper integer limits, } (\text{S1})$$

$$\sum_{i=0}^n i = \sum_{i=1}^n i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \quad (\text{S2})$$

EXAMPLE 2: Consider the element uniqueness problem:

check whether all the Elements in a given array of n elements are distinct.

ALGORITHM UniqueElements($A[0..n - 1]$)

//Determines whether all the elements in a given array are distinct

//**Input:** An array $A[0..n - 1]$.

//**Output:** Returns “true” if all the elements in A are distinct and “false” otherwise

for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow i + 1$ **to** $n - 1$ **do**

if $A[i] = A[j]$

return false

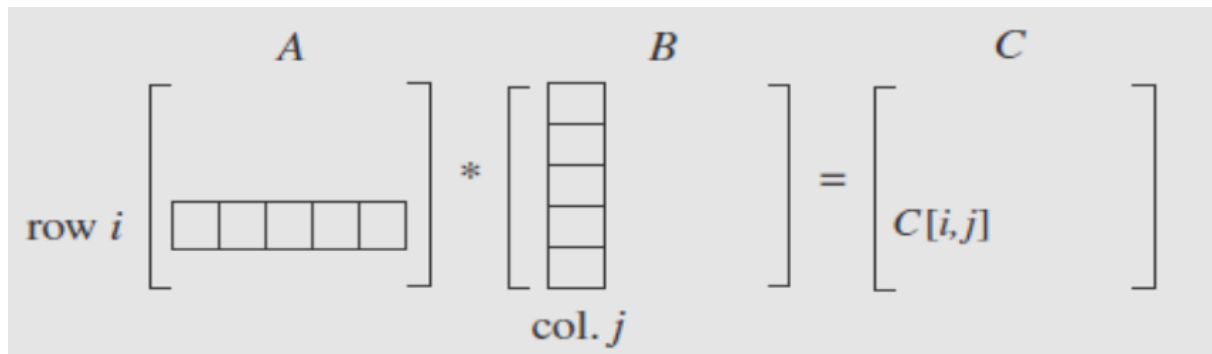
return true

Algorithm analysis

- The natural measure of the input's size here is again n .
- Since the innermost loop contains a single operation, we should consider it as the algorithm's basic operation.
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.
- One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits $i + 1$ and $n - 1$; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and $n - 2$.

$$\begin{aligned}
 C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) \\
 &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
 &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2).
 \end{aligned}$$

EXAMPLE 3: Consider matrix multiplication. Given two $n \times n$ matrices A and B , find the time efficiency of the definition-based algorithm for computing their product $C = AB$. By definition, C is an $n \times n$ matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and the columns of matrix B :



where $C[i, j] = A[i, 0]B[0, j] + \dots + A[i, k]B[k, j] + \dots + A[i, n-1]B[n-1, j]$ for every pair of indices $0 \leq i, j \leq n-1$.

ALGORITHM MatrixMultiplication(A[0..n – 1, 0..n – 1], B[0..n – 1, 0..n – 1])

//Multiplies two square matrices of order n by the definition-based algorithm

//**Input:** Two n × n matrices A and B

//**Output:** Matrix C = AB

for i ← 0 **to** n – 1 **do**

for j ← 0 **to** n – 1 **do**

 C[i, j] ← 0.0

for k ← 0 **to** n – 1 **do**

 C[i, j] ← C[i, j] + A[i, k] * B[k, j]

return C

Algorithm analysis

- An input's size is matrix order n.
- There are two arithmetical operations (multiplication and addition) in the innermost loop.
- But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm.
- Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound n – 1.
- Therefore, the number of multiplications made for every pair of specific values of variables i and j is ,

$$\sum_{k=0}^{n-1} 1$$

The total number of multiplications M(n) is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Now, we can compute this sum by using formula (S1) and rule (R1)

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

The running time of the algorithm on a particular machine m, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

If we consider, time spent on the additions too, then the total time on the machine is

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

MATHEMATICAL ANALYSIS FOR RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 1: Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n . Since $n! = 1 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$, for $n \geq 1$ and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

ALGORITHM $F(n)$

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0
    return 1
else
    return  $F(n-1) * n$ 
```

Algorithm analysis

- For simplicity, we consider n itself as an indicator of this algorithm's input size. i.e. 1.
- The basic operation of the algorithm is multiplication, whose number of executions we denote $M(n)$. Since the function $F(n)$ is computed according to the formula $F(n) = F(n-1) \cdot n$ for $n > 0$.
- The number of multiplications $M(n)$ needed to compute it must satisfy the equality

$$M(n) = \underset{\substack{\uparrow \\ \text{To compute} \\ F(n-1)}}{M(n-1)} + \underset{\substack{\uparrow \\ \text{To multiply} \\ F(n-1) \text{ by } n}}{1} \quad \text{for } n > 0$$

$M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by n .

Recurrence relations

- The last equation defines the sequence $M(n)$ that we need to find. This equation defines $M(n)$ not explicitly, i.e., as a function of n , but implicitly as a function of its value at another point, namely $n-1$. Such equations are called recurrence relations or recurrences.
- Solve the recurrence relation $M(n) = M(n-1) + 1$,

- To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ return 1.

This tells us two things. First, since the calls stop when $n = 0$, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when $n = 0$, the algorithm performs no multiplications.

Method of backward substitutions

$$\begin{aligned}
 M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1 \\
 &= [M(n-2) + 1] + 1 \\
 &= M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1 \\
 &= [M(n-3) + 1] + 2 \\
 &= M(n-3) + 3 \\
 &\dots \\
 &= M(n-i) + i \\
 &\dots \\
 &= M(n-n) + n \\
 &= n.
 \end{aligned}$$

Therefore $M(n)=n$

Brute Force Algorithms :

Definition

Brute Force is a straightforward approach to solving a problem, directly based on the problem statement and definitions of the concepts involved.

It relies on the computational power of a computer to systematically try all possible solutions without requiring complex intellectual design. Often described as the "Just do it!" approach.

General Algorithm :

A brute-force algorithm solves a problem by:

1. Directly applying the problem's definition or statement.
2. Enumerating all possible solutions or combinations.
3. Checking each solution to find the correct one.

Example: For computing a^n , multiply a by itself n times (i.e., $a \times a \times \dots \times a$ times).

Exponentiation Problem: Compute a^n for a nonzero number a and nonnegative integer n .

Brute-force approach: Start with 1 and multiply by a , n times.

Pseudocode:

```
result ← 1
for i ← 1 to n do
    result ← result × a
return result
```

Other examples:

- Consecutive integer checking for **GCD**.
- Definition-based **matrix multiplication**.

Analysis :

Time Complexity: Depends on the problem. For example:

Exponentiation: ($O(n)$), where (n) is the exponent.

Matrix multiplication: ($O(n^3)$) for ($n \times n$) matrices.

Space Complexity: Typically minimal, as brute-force algorithms store only the current solution being tested (e.g., ($O(1)$) for exponentiation).

Advantages

Simplicity: Easy to understand and implement.

Applicability: Works for a wide variety of problems.

No Size Limitation: Can handle problems of any instance size.

Reliability: Guarantees a solution if one exists (for exhaustive search).

Useful for Small Instances: Efficient enough for small inputs or one-time problem-solving.

Limitations

Inefficiency: Often has poor time complexity (e.g., exponential for combinatorial problems).

Scalability Issues: Impractical for large inputs.

Lack of Cleverness: Does not exploit problem-specific properties to optimize.

Resource Intensive: Requires significant computational power for large problems.

Sequential Search

Definition :

- **Sequential Search** is a brute-force algorithm for searching an element (key) in a list by comparing the key with each element sequentially until a match is found or the list is exhausted.
- Also known as linear search, it is the simplest searching algorithm.

Algorithm :

- **Pseudocode:**

```
ALGORITHM SequentialSearch(A[0..n-1], K)
  for i ← 0 to n-1 do
    if A[i] = K then
      return i
  return -1
```

Example

Problem: Search for ($K = 5$) in array ($A = [3, 1, 5, 7, 2]$).

Compare (K) with ($A[0] = 3$) (no match).

Compare (K) with ($A[1] = 1$) (no match).

Compare (K) with ($A[2] = 5$) (match, return index 2).

Analysis

Time Complexity:

Worst Case: ($O(n)$), when the key is at the end or not present.

Average Case: ($O(n)$), assuming the key is equally likely to be at any position.

Best Case: ($O(1)$), when the key is at the first position.

Space Complexity: ($O(1)$) (excluding input array), as it uses only a few variables.

Advantages

- **Simplicity:** Easy to implement and understand.
- **No Preprocessing:** Works on unsorted lists without requiring sorting.
- **General Applicability:** Works for any list, regardless of order or size.
- **Guaranteed Result:** Finds the key if it exists or confirms its absence.

Limitations

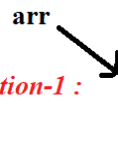
- **Inefficiency:** ($O(n)$) time complexity, poor for large lists.
- **No Optimization for Sorted Lists:** Standard version doesn't exploit sorted order effectively.
- **Scalability:** Not suitable for large datasets compared to faster algorithms like binary search.

Linear Search

* Element to Search : 8

arr

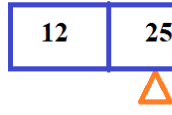
Iteration-1 :



12	25	8	10	32
----	----	---	----	----

⇒ $12 \neq 8$ (Does not Match. So Continue...)

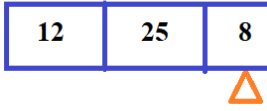
Iteration-2 :



12	25	8	10	32
----	----	---	----	----

⇒ $25 \neq 8$ (Does not Match. So Continue...)

Iteration-3 :



12	25	8	10	32
----	----	---	----	----

⇒ $8 = 8$ (It Matches. So it returns index of 8 i.e. 2)