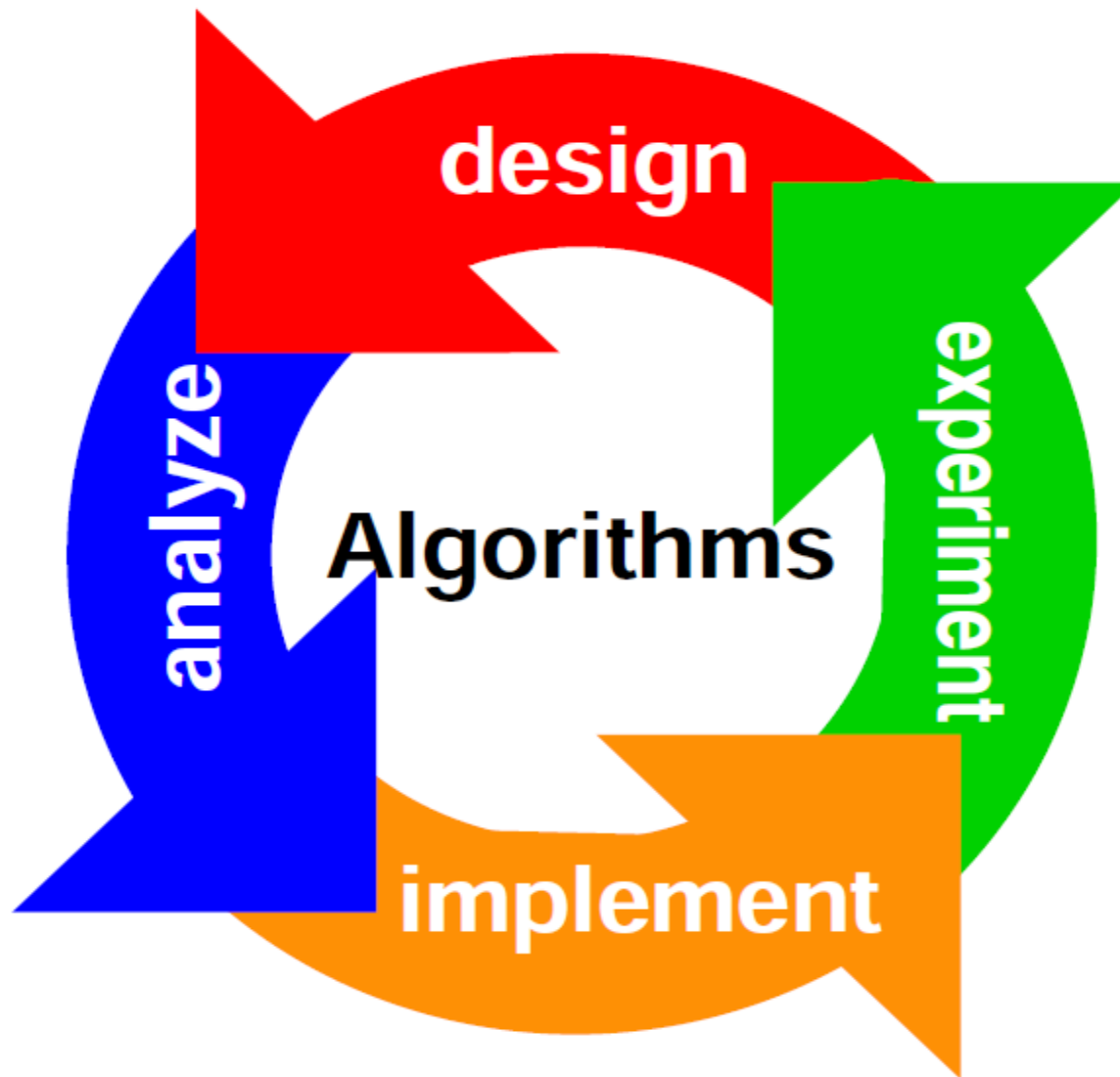


Algorithm



History of algorithm...

- The word algorithm comes from the **9th-century** mathematician Muhammad ibn Mūsā al-Khwārizmī, latinized as Algoritmi.



Who invented algorithm in computer science?

- **Alan Turing** first formalized the concept of the algorithm in 1936 with his infamous Turing machine. The addition of Alonzo Church's lambda calculus paved the way for modern computer science.

Definition of algorithm.

- **An algorithm refers to a procedure for solving a problem; this procedure typically plays out in a finite number of steps and they frequently involve repetition of an operation.**
- **algorithms are not just used in mathematics and computer science but are also used in daily life.**
- **So an informal definition of algorithm would describe it as a set of rules that precisely defines a sequence of operations.**

Agenda

- What Is an Algorithm?
- Characteristic of an Algorithm
- How to Write an Algorithm?
- Algorithm Analysis
- Algorithm Complexity
- Pros and Cons of an Algorithm
- Algorithm vs Programming





What is an Algorithm?




What is an Algorithm?



She is Shreya, and she wants to prepare tea. To do so, she is following a series of steps.

What is an Algorithm?

Suggested: Data Structures & Algorithms [2022 Updated] 

STEP

1

Take a pan, fill it with water, and place it on the gas stove.

STEP

2

After the water has come to a boil, add the tea leaves and sugar.

STEP

3

Allow for full leaf expansion and milking after that.



What is an Algorithm?

STEP

4

Then wait for the tea to come to a boil.

STEP

5

Turn off the gas after it has finished cooking.

STEP

6

You're all set with your tea now.

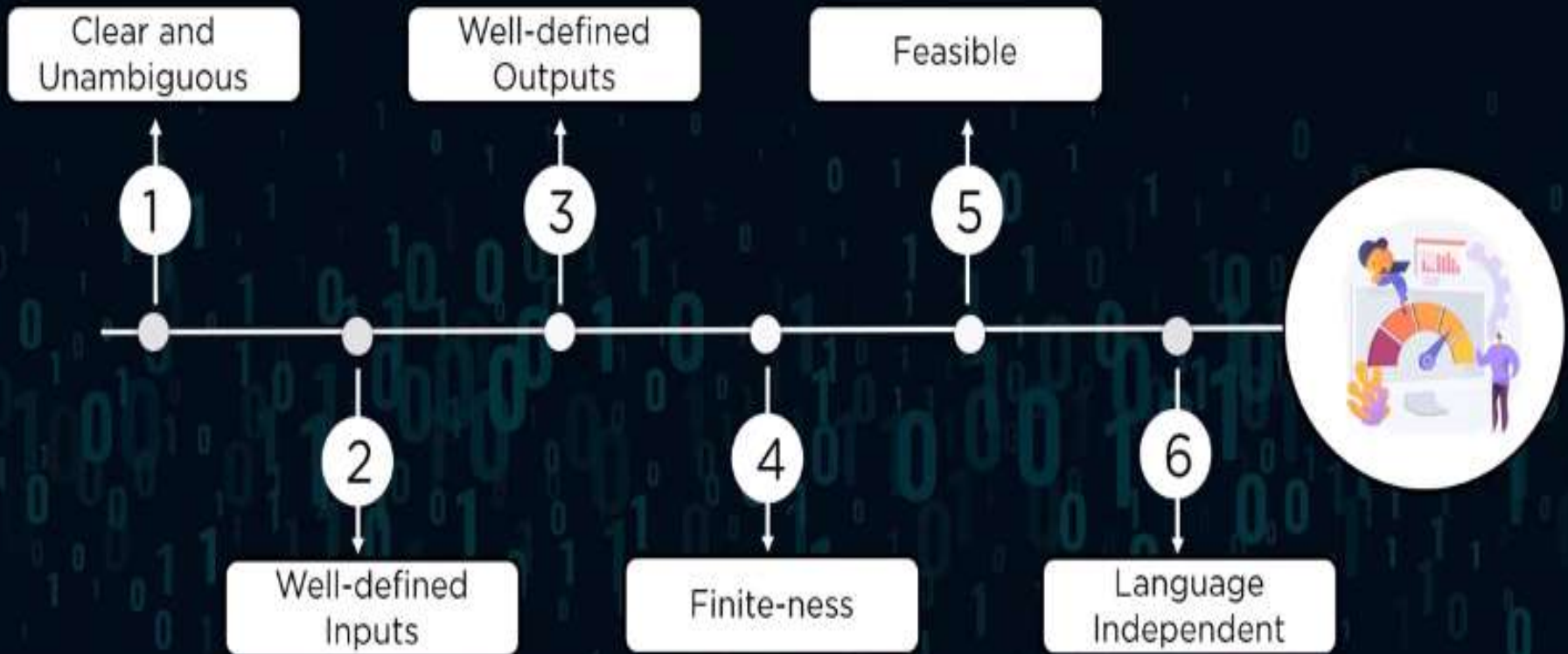


What is an Algorithm?

A method or set of rules that must be followed when performing problem-solving operations. As a result, an algorithm is a collection of rules or instructions that govern how a work is to be conducted step-by-step to achieve the desired results is termed as an algorithm.



Characteristics of an Algorithm



- **Clear and Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.

- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.
- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon with the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected

How to write an Algorithm?

- Writing algorithm does not have any well-defined standards.
- Algorithms are never created to support a specific programming language.
- As we all know, basic code features such as loops (do, for, while all programming languages share), flow control (if-else), and so on.
- We usually create algorithms step by step; however, this isn't always the case.
- After the problem domain has been well-defined, algorithm writing is a procedure that is carried out.



How to write an Algorithm?

Algorithm to find the largest number among three numbers x, y and z.

Solution 1

step 1: start
Step 2: Read x,y,z
Step 3: if $x > y$ then go to step 4
 else go to step 5
Step 4: if $x > z$ print x is largest
 else z is largest
Step 5: if $y > z$ print y is largest
 else z is largest
Step 6: stop



Algorithm Analysis

The algorithm may be studied on two levels: first, before it is made, and then, after it is created. The two analyses of an algorithm are as follows:



Algorithm Analysis

- Priori analysis refers to the theoretical analysis of an algorithm performed before its implementation.
- Before implementing the algorithm, other parameters, such as processor speed, might be considered, which does not affect the implementation component.



Priori Analysis

- A practical analysis of an algorithm is called posterior analysis, and it is utilizing any computer language to build the algorithm.
- The purpose of this analysis is to determine how much time and space the algorithm takes to operate.



Post Analysis

How to Design an Algorithm?

- The **problem** that is to be solved by this algorithm.
- The **constraints** of the problem that must be considered while solving the problem.
- The **input** to be taken to solve the problem.
- The **output** to be expected when the problem the is solved.
- The **solution** to this problem, in the given constraints.

HOW?

- **The problem that is to be solved by this algorithm:**
Add 3 numbers and print their sum.
- **The constraints of the problem that must be considered while solving the problem:** The numbers must contain only digits and no other characters.

- **The input to be taken to solve the problem:** The three numbers to be added.
- **The output to be expected when the problem this is solved:** The sum of the three numbers taken as the input.
- **The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or any other method.

Algorithm to add 3 numbers and print their sum:

START

- Declare 3 integer variables num1, num2 and num3.
- Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
- Declare an integer variable sum to store the resultant sum of the 3 numbers.
- Add the 3 numbers and store the result in the variable sum.
- Print the value of variable sum
- END


```
import java.util.Scanner;

public class SumOfNumbers2
{
    public static void main(String args[])
    {
        int x, y, sum;
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the first number: ");
        x = sc.nextInt();
        System.out.print("Enter the second number: ");
        y = sc.nextInt();
        sum = sum(x, y);

        System.out.println("The sum of two numbers x and y is: " + sum);
    }
}
```

```
//method that calculates the sum  
public static int sum(int a, int b)  
{  
  int sum = a + b;  
  return sum;  
}  
}
```

Types of Algorithms

Types of Algorithm.

- Algorithm types we will consider include:
 - Simple recursive algorithms
 - Backtracking algorithms
 - Divide and conquer algorithms
 - Dynamic programming algorithms
 - Greedy algorithms
 - Branch and bound algorithms
 - Brute force algorithms
 - Randomized algorithms

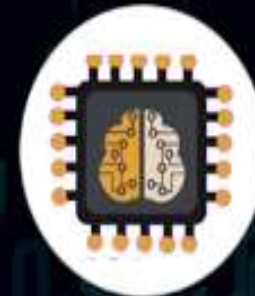
Algorithm Complexity

Two factors can be used to evaluate the algorithm's performance:



Time Complexity

The amount of time required to finish an algorithm's execution is known as its time complexity.



Space Complexity

The quantity of space required to solve a problem and produce an output is referred as space complexity

Algorithm Complexity



Time Complexity

- The big O notation expresses an algorithm's time complexity.
- The asymptotic notation used to depict the temporal complexity is big O notation.
- The time complexity is mainly determined by counting the number of steps required to complete the task.

Algorithm Complexity

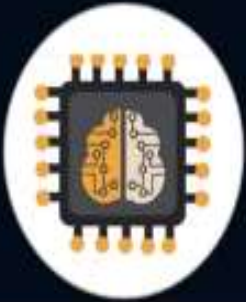
Example

```
mul = 1;  
// calculate the multiplication of n number.  
For i = 1 to n  
mul=mul *i //mul holds product of n numbers.  
Return mul;
```



The loop statement's time complexity will be at least n , and when the value of n increases, the time complexity will also increase. While the code's complexity will remain constant as its value is unaffected by the value of n , and it will obtain the result in a single step.

Algorithm Complexity



Space complexity is stated in big O notation, same as time complexity.

Space Complexity

To store
program
instruction

To store
constant value

To store
variable values

To keep track
of function
calls, jump
statements
etc.

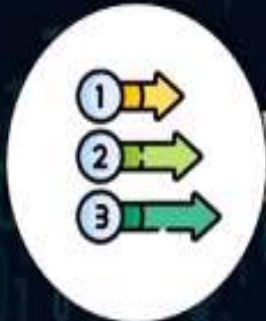
Space Complexity= Auxiliary Space + input Size

Pros of Algorithm

Algorithms are easy to understand.



Algorithm is a step-by-step solution to a problem.



Algorithm divides problem into smaller steps



Cons of Algorithm

Difficult to demonstrate branching and looping statements.



Writing an algorithm is a time-consuming process



What is Different between Algorithm and programming

Algorithm vs Programming

An algorithm is more like a concept, a technique to solve a problem.



Programming

vs

Algorithm



A programmer is related to executing one or more tasks by a computer

Algorithm vs Programming

An algorithm is designed, and we can analyze the algorithm.



Programming

vs

Algorithm



Programming is in the implementation phase, so we test the programs.

How to Write Algorithm.

Write an algorithm to add two numbers entered by user.

Step 1: Start

Step 2: Declare variables num1,

Step 3: Read values num1 and num2

Step 4: Add num1 and num2 and
 $sum \leftarrow num1 + num2$

Step 5: Display sum

Step 6: Stop

How do you write an algorithm for adding two numbers?

The algorithm to add two numbers.

- Declare variable (Two variable to store the number input by the user and one variable is used to store the output).
- Take the input of two numbers.
- Apply the formula for addition.
- Add two numbers.
- Store the result in a variable.
- Print the result.

Important of Algorithm

- **An algorithm is important in optimizing a computer program according to the available resources. .
Ultimately when anyone decide to solve a problem through better algorithms then searching for the best combination of program speed and least amount of memory consumption is desired.**

Why study algorithm

- The first reason why I would urge you to study algorithms is because a lot of them are quite general and a need for a certain one can come up many times over the course of one's career (if you choose to become, or are, some sort of developer). If you are stuck on a problem, there could be a published algorithm that could help you; however, if you haven't studied them too much and hence do not know that the particular one you need exists, you are going to waste time figuring out how to handle the situation, and then probably write a sub-optimal solution to the problem as well.

- The second reason is that it will increase your problem solving and logic skills, therefore meaning that studying them will benefit you even if you never need to implement any that you read about in your whole life. Steve Jobs once said that he thought everyone should learn how to program a computer because it teaches you how to think. I completely agree with this statement, and learning algorithms takes it even further. If you study and understand why problems have been solved certain ways, then that will have a massively positive effect on how you approach problems yourself in the future, both in software development and other areas of life.

- The third reason is that many developer interviews are technical in nature, and so you will likely be questioned on algorithms, data structures and so forth if that is your industry.



TASK

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

Space Complexity

Time Complexity

SPACE COMPLEXITY

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components:

- **Instruction Space** : Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space** : Its the space required to store all the constants and variables value.
- **Environment Space** : Its the space required to store the environment information needed to resume the suspended function.

TIME AND SPACE COMPLEXITY ANALYSIS

Constant Space Complexity

```
int square(int a)
{
    return a*a;
}
```

If any algorithm requires a fixed amount of space for all input values then that

space complexity is said to be Constant Space Complexity

Linear Space Complexity

```
int sum(int A[], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return sum;
}
```

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity

TIME COMPLEXITY

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity

```
int sum(int a, int b)
{
    return a+b;
}
```


Linear Time Complexity

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be Linear Time Complexity

```
int sum(int A[], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return sum;
}
```

ASYMPTOTIC NOTATION

Asymptotic notation of an algorithm is a mathematical representation of its complexity

For example, consider the following time complexities of two algorithms...

Algorithm 1 : $5n^2 + 2n + 1$

Algorithm 2 : $10n^2 + 8n + 3$

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

Big - Oh (O)

Big - Omega (Ω)

Big - Theta (Θ)

Time Complexity

Block of code executed for a constant time or repeated based on inputs

Assuming time taken for

printf - c1 seconds

scanf - c2 seconds

For loop

Initialization – c3 seconds

Condition checking – c4 seconds

Increment – c5 seconds

Total Time Taken = $\underline{C_1 + C_2 + C_3 + (n+1)*C_4 + n*C_5 + n * C_1}$

```
System.out.print("Enter n: \n");
```

```
int n = sc.nextInt();
```

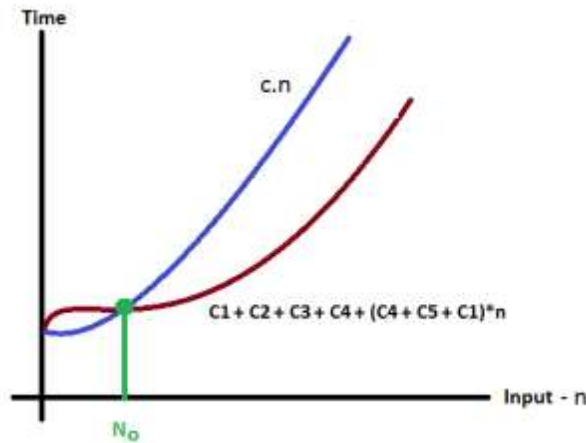
```
for(i = 1; i <= n; i++)
```

```
{
```

```
    System.out.print(i);
```

```
}
```

$$C_1 + C_2 + C_3 + (n+1)*C_4 + n*C_5 + n * C_1 = C_1 + C_2 + C_3 + C_4 + n(C_4 + C_5 + C_1)*n$$



$$= K_1 + K_2*n$$

$$\propto n$$

$$= O(n)$$

```
System.out.print("Enter n: \n");
```

```
int n = sc.nextInt();
```

```
for(i = 1; i <= n; i++)
```

```
{
```

```
    System.out.print(i);
```

```
}
```

Big Order Notation

- Time taken for
 - If with 1 condition – c6 seconds
 - Break statement – c7 seconds
- Best case time taken = **C3 + C4 + C6 + C1 + C7**
- Worst case time taken = **C3 + n*C4 + n*C5 + n*C6 + C1 + C7**
- A.K.A **upper bound time** for a code

- Big order notation $O()$
 - Upper bound time for a code
 - Eliminates constants C_3, C_4, \dots and approximates with function $O()$
 - If a code takes $O(n)$ time, it means

$$\text{Code Exec time} \leq k * n$$

- If a code takes $O(n)$ time, then the code takes $O(n^2)$ as well. True or false?

```
for(i = 1; i <= n; i++)  
{  
    if(a[i] == elem)  
    { System.out.print("found");  
      break;  
    }  
}
```


- Worst case time = $C_1 + C_2$;
= K
= $O(1)$

```
System.out.print("Enter n: \n");  
int n = sc.nextInt();
```

- Worst case time = $C_3 + 11 \cdot C_4 + 10 \cdot C_5 + 10 \cdot C_1$
= $C_3 + C_4 + 10 \cdot C_4 + 10 \cdot C_5 + 10 \cdot C_1$
= $C_3 + C_4 + 10 \cdot (C_4 + C_5 + C_1)$
= $K_1 + 10 \cdot K_2$
 $\leq K_{3.1}$
= $O(1)$

```
for(i = 1; i < 10; i++)  
{  
    System.out.print(i);  
}
```

- Shortcuts
 - Do not write terms that are repeated once
 - Ignore for loop init, condition checking and incrementing part
 - Only count the number of times expressions inside the loop are executing
 - Approximate **n** and **(n+1)** as **n**
 - $n \cdot C_4 + (n+1) \cdot C_5$ can be written as $n (C_4 + C_5)$
- Worst case time = $n (C_1) = O(n)$

```
int n = sc.nextInt();  
  
for(i = 1; i <= n; i++)  
{  
    System.out.print(i);  
}
```

- Just count how many times `printf("x")` is executed

- Worst case exec time

$$= (n + n + \dots + n) \cdot C_1$$

$$= n^2 \cdot C_1$$

$$= O(n^2)$$

```
int n = sc.nextInt();  
  
for(i = 1; i <= n; i++)  
{  
    System.out.print(i);  
}
```

i	1	2	3	...	n
#	n	n	n	...	n

- Just count how many times `printf("x")` is executed

- Worst case exec time

$$= (n + n + \dots + n) \cdot C_1$$

$$= n^2 \cdot C_1$$

$$= O(n^2)$$

```
for(i = 1; i <= n; i++)
{
    for(j = 1; j <= n; j++)
    {
        System.out.print("*");
    }
}
```

i	1	2	3	...	n
#	n	n	n	...	n

- Just count how many times `printf("x")` is executed

- Worst case exec time

$$= (1 + 2 + \dots + n) \cdot K$$

$$= \frac{n \cdot (n+1)}{2} \cdot K$$

$$= (n^2 + n) \cdot \frac{K}{2}$$

$$= K_1 \cdot n^2 + K_2 \cdot n$$

$$= O(n^2) + O(n)$$

$$= O(n^2)$$

```
for(i = 1; i <= n; i++)
{
    for(j = 1; j <= n; j++)
    {
        System.out.print("*");
    }
}
```

i	1	2	3	...	n
#	n	n	n	...	n

- Worst case exec time

$$= (1 + 2 + \dots + n) \cdot C_1$$

$$= \frac{n \cdot (n+1)}{2} \cdot C_1$$

$$= (n^2 + n) \cdot \frac{C_1}{2}$$

$$= K_1 \cdot n^2 + K_1 \cdot n$$

$$= O(n^2) + O(n)$$

$$= O(n^2)$$

```
for(i = 1; i <= n; i++)  
{  
    for(j = 1; j <= n; j++)  
    {  
        System.out.print("*");  
    }  
}
```

i	1	2	3	...	n
#	n	n	n	...	n

- Worst case exec time

$$\begin{aligned}
 &= (n + n + \dots + n) \cdot K + \\
 &\quad (n + n + \dots + n) \cdot K \\
 &= n^2 \cdot K + n^2 \cdot K = n^2 \cdot K \cdot 2 \\
 &= O(n^2)
 \end{aligned}$$

```

for(i = 1; i <= n; i++)
{
    for(j = 1; j <= n; j++)
    {
        System.out.print("*");
    }

    for(j = 1; j <= n; j++)
    {
        System.out.print("*");
    }
}
    
```

i	1	2	3	...	n
#1	n	n	n	...	N
#2	n	n	n	...	n

Algorithm	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$
Quick sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$
Merge sort	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$

Algorithm	Best	Average	Worst
Linear Search	$O(1)$	$O(n)$	$O(n)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$

Operations

$O(n^2)$

$O(n \log n)$

$O(n)$

Elements

How to calculate Time Complexity?

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N , as N approaches infinity. In general you can think of it like this:

Statement;

Above we have a single statement. Its Time Complexity will be Constant. The running time of the statement will not change in relation to N .

```
for(i=0; i < N; i++)  
{  
  
    statement;  
  
}
```

The time complexity for the above algorithm will be Linear. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)  
{  
  
    for(j=0; j < N;j++)  
    {  
  
        statement;  
  
    }  
  
}
```

This time, the time complexity for the above code will be Quadratic. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by $N * N$.


```
while(low <= high)  
{  
    mid = (low + high) / 2;  
    if (target < list[mid])  
        high = mid - 1;  
    else if (target > list[mid])  
        low = mid + 1;  
    else break;  
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field. Now, this algorithm will have a Logarithmic Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2 (N is high-low here).

This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)  
{  
int pivot = partition(list, left, right);  
quicksort(list, left, pivot - 1);  
quicksort(list, pivot + 1, right);  
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort.

Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times (where N is the size of list). Hence time complexity will be $N \cdot \log(N)$.

The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

BIG - OMEGE NOTATION (Ω)

Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.

Consider function $f(n)$ the time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \geq C \times g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$.

Then we can represent $f(n)$ as $\Omega(g(n))$.

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

Example

Consider the following $f(n)$ and $g(n)$..

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

$$g(n) = n$$

IF we want to represent $f(n)$ and $\Omega(g(n))$ then it must satisfy $f(n) \geq C g(n)$ for all values of $C > 0$ and $n \geq 1$

$$f(n) \geq C g(n)$$

$$\rightarrow 3n+2 \leq C n$$

Above condition is always TRUE for all values of $C = 1$ and $n \geq 1$.
By using Big – Omega notation we can represent the time complexity as follows...

$$3n + 2 = \Omega(n)$$

BIG - THETA NOTATION (Θ)

Big - Theta notation is used to define the average bound of an algorithm in terms of Time Complexity.

Consider function $f(n)$ the time complexity of an algorithm and $g(n)$ is the most significant term. If $C_1g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_0$, $C_1, C_2 > 0$ and $n_0 \geq$

1. Then we can represent $f(n)$ as $\Theta(g(n))$.

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

Example

Consider the following $f(n)$ and $g(n)$..

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

$$g(n) = n$$

If we want to represent $f(n)$ and $\Theta(g(n))$ then it must satisfy $C_1 g(n)$

$= f(n) \geq C_2 g(n)$ for all values of $C_1, C_2 > 0$ and $n_0 \geq 1$

$$C_1 g(n) \leq f(n) \geq \rightarrow C_2 g(n)$$

$$C_1 n \leq 3n+2 \geq C_2 n$$

Above condition is always TRUE for all values of $C_1 = 1, C_2 = 4$ and $n = 1$. By using Big – Theta notation we can represent the time complexity as follows...

$$3n + 2 = \Theta(n)$$


```
1.  int fun()
    {
        for(i = 1; i < n; i++)
        {
            cout << "SMART";
        }
        return 0;
    }
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
2. int fun()
{
    for(i = 1; i <= sqrt(n); i++)
    {
        cout << "SMART";
    }
    return 0;
}
```

(a) $O(n)$

(c) $O(\sqrt{n})$

(b) $O(\log n)$

(d) $O(n * n)$

```
3. int fun()
{
    for(i = 1; i*i <= n; i++)
    {
        cout << "SMART";
    }
    return 0;
}
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
4. int fun()
{
    for(i = 1; i <= n; i=i*2)
    {
        cout << "SMART";
    }
    return 0;
}
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
5. int fun()
{
    int n, i = 1, s = 1;
    while(s <= n)
    {
        i++;
        s = s + i;
        cout << "SMART";
    }
    return 0;
}
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
6.  int fun()
    {
        int n;
        while(n > 1)
        {
            n = n / 2;
        }
        return 0;
    }
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
7. int fun()
{
    for(i = 1; i < n; i++)
    {
        for(j = 1; j < n; j++)
        {
            cout << "SMART";
        }
    }
    return 0;
}
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$


```
8. int fun()
{
    for(i = 1; i < n; i++)
    {
        for(j = 1; j < n; j=j+i)
        {
            cout << "SMART";
        }
    }
    return 0;
}
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
9.  int fun()
    {
        int i, j;
        for(i = 1; i < n; i++)
        {
            for(j = 1; j < n; j=j*2)
            {
                cout << "SMART";
            }
        }
        return 0;
    }
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
10. int fun()
{
    int i, j;
    for(i = 1; i < n; i++)
    {
        for(j = n; j > 1; j--)
        {
            cout << "SMART";
        }
    }
}
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$

```
11. int fun()
    {
        for(i = 1; i < n; i++)
        {
            for(j = 1; j < n; j++)
            {
                cout << "SMART";
                break;
            }
        }
        return 0;
    }
```

(a) $O(n)$

(b) $O(\log n)$

(c) $O(\sqrt{n})$

(d) $O(n * n)$