What is an Algorithm?

A: An algorithm is the step-by-step instruction to solve a given problem.

Let us consider the problem of preparing an omelet. For preparing omelet, general steps we follow are:

1. Get the frying pan
2. Get the foil
    a. Do we have oil?
        i. If yes, put it in the pan.
        ii. If no, do we want to buy oil?
            a. If yes, then go out and buy.
            b. If no, we can terminate.
3. Turn on the stove, etc…

Why Analysis of Algorithms?

A: Algorithm analysis helps us determining which of them is efficient in terms of time and space consumed.

For Example: To go from one city to another, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle. Depending on the availability and convenience we choose the one that suits us. Similarly, in computer science multiple algorithms are available for solving the same problem.

Email: aatish.azad7@gmail.com

Goal of Analysis of Algorithms

A: The goal of analysis of algorithms is to compare algorithms (or solutions) mainly in terms of running time but also in terms of others factors (e.g, memory, developers' efforts, etc.)

What is Running Time Analysis?

A: It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is the number of elements in the input and depending on the problem type the input may be of different types. The following are the common types of inputs.

- Size of the Array
- Number of elements in a matrix
- Number of bits in binary representation of the input

How to Compare Algorithms?

A: To compare algorithms, there are ways to do that:

Execution times? *Not a good measure* as execution times is specific to a particular computer.

Number of statements executes? *Not a good measure, because* the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal Solution

Let's see an example:

Suppose we want to write a function that calculates the sum of all numbers from 1 up to (and including) some number n.

```
public static Long addUpTo(Long number) {
    Long total = 0L;
    for(Long i = 0L;i<=number;i++) {
        total += total;
    }
    return total;
}
```

Or

```
public static Long addUpToQuick(Long number) {
    return number * (number + 1)/2;
}
```

So, In the given two example. We are having two methods: addUpTo() and addUpToQuick(). Now, the question is , Which one is better?

But, then **What better means?**

Does it means "**Faster**"?

Does it means "**Less memory-intensive**"?

Does it means "**More readable**"?

In short, all of the above points are valid concerns. But, most will agree that, the first two (speed and memory usage) are more important.

We'll focus on evaluating speed. We can calculate the speed of an algorithm by measuring the time duration taken by the algorithm.

Example:

```
public static void countingDuration() {
    Long number = 900000000L;
    Instant start = Instant.now();
    addUpTo(number);
    Instant end = Instant.now();
    Long duration = Duration.between(start, end).toMillis();
    Double seconds = duration/1000.00;
    System.out.println("addUpTo: " + seconds + " Seconds");

    start = Instant.now();
    addUpToQuick(number);
    end = Instant.now();
    duration = Duration.between(start, end).toMillis();
    seconds = duration/1000.00;
    System.out.println("addUpToQuick: " + seconds + " Seconds");
}
```

Now, it may seems like we might just calculate the duration of an algorithm to label its performance. But, **the problem with time is:**

- **Different machine will record different times.**
- **The same machines will record different times.** Because, the results may vary depending on the number of application used by the machine.
- **For fast algorithms, speed measurements may not be precise enough?** For example, we have two or three or four algorithms, which are superfast. But there will be the one which will be the fastest among all. We can't find the fastest among superfast algorithm with our time function because they always vary and they are not precise.
- **It's a bit difficult to wait** for two algorithms to complete their execution. For example, what if one algorithm takes two hours and

the other will take five hours. We can't wait for hours or day to find out which algorithm is best and more efficient.

So, what we want is to assign a general value for an algorithm without having to go through the above problem.

So, now the question is "*If not time than what*"?

So, rather than counting seconds which are so variable, let's **count the number of operations the computer has to perform**!

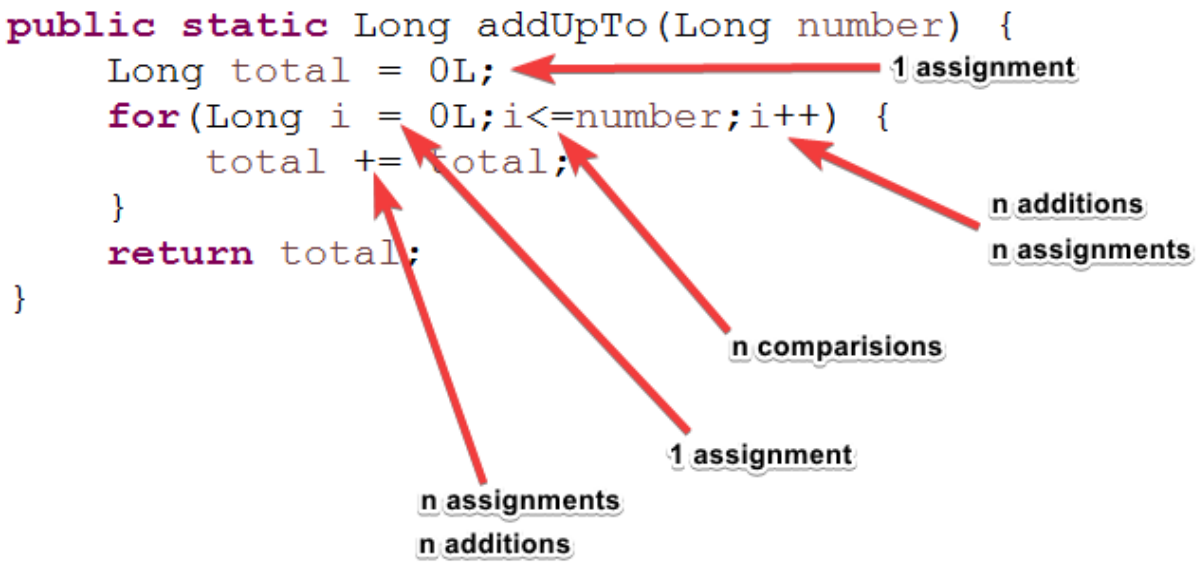
For example, it becomes very easy to compare two algorithms by comparing the number of operations they are doing. Like, this algorithm has five operations and the other has 7. **It doesn't matter what the specifications of the computer are**. Time we might get different but the time is always be determined by the number of operations. So, we can use that, **rather than comparing the exact time, we can focus on the number of single operation a computer has to perform.**

Counting Operations:

```
public static Long addUpToQuick(Long number) {
    return number * (number + 1)/2;
}
```

So, there are 3 operations. And it doesn't really matter what number is given as input. Because there are only three calculations that are happening. So, 3 simple operations, regardless of the size of n.

Let's compare it with the other solution:



n additions, n assignments and n comparisons means, the operation are dependent on the input given to the methods, because they are in a loop and they will execute n number of time for n value.

So, we can say, the above algorithm has: 5n + 2 operations. This is one way of doing it.

Counting is hard:

- Depending on what we count, the number of operations can be as low as 2n or as high as 5n + 2
- But, regardless of the exact number, the number of operations grows roughly proportionally with n.

What is the Rate of Growth?

A: The rate at which the running time increases as a function of input. It is called as **Rate of Growth.**

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
logn	Logarithmic	Finding an element in an sorted array
n	Linear	Finding an element in an unsorted array
nlogn	Linear Logarithmic	Sorting n items by 'divide-and-conquer'- Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Types of Analysis

To analyze the given algorithm we need to know what input the algorithm takes less time (performing well) and on what inputs the algorithm takes long time.

We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for the case where it takes less time and other for the case where it takes the more time.

In general the first case is called the ***best case*** and second case is called the ***worst case*** of the algorithm. To analyze the algorithm we need some kind of syntax and that forms the base for asymptotic analysis/notation.

There are three types of analysis:

- **Worst case**
  - Defines the input for which the algorithm takes long time.
  - Input is the one for which the algorithm runs the slower.
- **Best case**
  - Defines the input for which the algorithm takes lowest time.
  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm.
  - Assumes that the input is random

Lower Bound <= Average Time <= Upper Bound

- Proving an upper bound means you have proven that the algorithm will use no more than some limit on a resource.
- Proving a lower bound means you have proven that the algorithm will use no less that some limit on a resource.
- "Resource" in this context could be time, memory, bandwidth, or something else.

Asymptotic Notation

Having the expressions for the best, average case and worse cases, for all the three cases we need to identify the upper and lower bounds. To represent these upper and lower bounds we need some kind of syntax and

that is we need some kind of notation. Let us assume that the given algorithm is represented in the form of function f(n).

Big-O Notation

Big O Notation is used to describe the performance or complexity of an algorithm. Big O specifically describe the worst-case scenario, and can be used to describe the execution time required or the space used (e.g in memory or on disk) by an algorithm.

As a programmer first and a mathematician second. I found the best way to understand Big O thoroughly was to produce some examples in code. So, below are some common orders of growth along with descriptions and examples where possible.

O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

```
bool IsFirstElementNull(IList<string> elements)
{
    return elements[0] == null;
}
```

O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favors the worst-case performance scenario; a matching string could be found during any iteration of the *for* loop and the function would return early, but Big O notation will always assume the

upper limit where the algorithm will perform the maximum number of iterations.

```
bool ContainsValue(IList<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }

    return false;
}
```


O(N^2)

O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithm that involve nested iterations over the data set. Deeper nested iterations

will result in $O(N^3)$, $O(N^4)$ etc.

```
bool ContainsDuplicates(IList<string> elements)
{
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]) return true;
        }
    }

    return false;
}
```

$O(2^N)$

$O(2^N)$ denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an $O(2^N)$ function is exponential – starting off very shallow, then rising meteorically. An example of an $O(2^N)$ function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

Binary Search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower that the target value is lower that the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

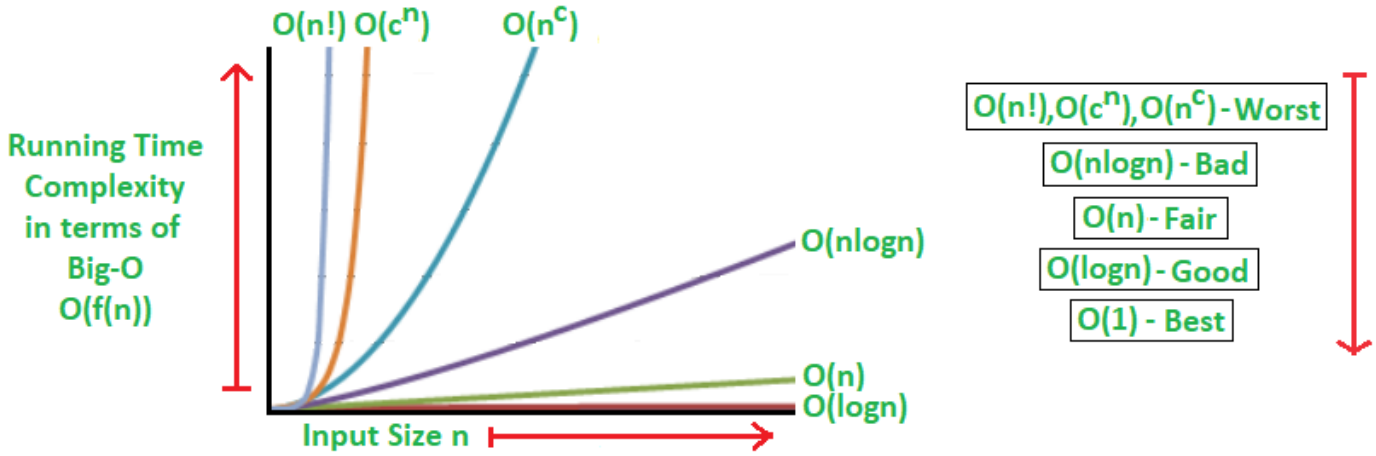
This type of algorithm is described as **O(log N)**. The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase. E.g. an input data set containing 10 items takes one second, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

Omega-Ω Notation

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given function (algorithm) are same. The average running time of algorithm is always between lower bound and upper bound. If the upper bound (O) and lower bound (**Ω)** give the same result then Θ notation will also have the same rate of growth.



Let talk about Space Complexity:

It means the amount of memory taken up for an algorithm.

We can also use big(O) notation to analyze space complexity: How much additional memory do we need to allocate in order to run the code in our algorithm?

What about the inputs?

If the n grows, then obviously the size of the inputs will grow, but we're going to ignore the input part.

It is also called as **Auxiliary space complexity:** It refers to space required by the algorithm not including the space taken up by the inputs.

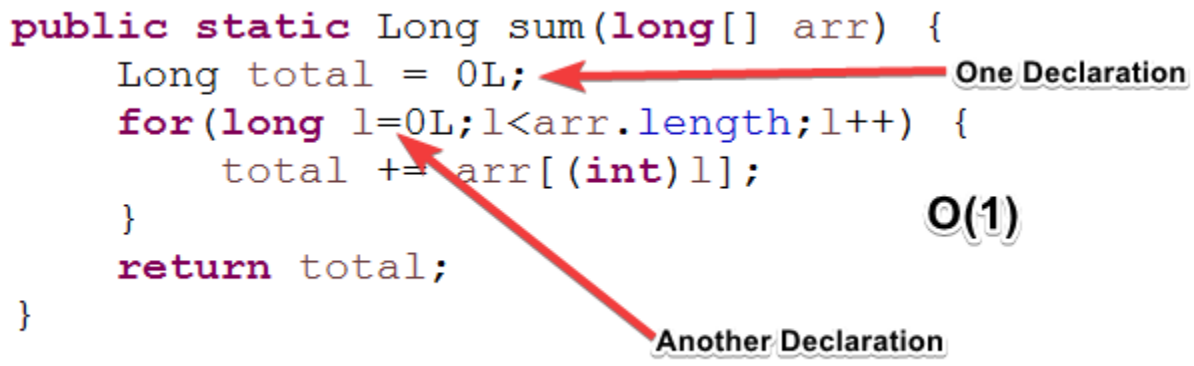
Note: When we talk about space complexity, technically we'll be talking about auxiliary space complexity.

Basically we are focusing on what happen inside the algorithm and not the input size.

Rules for Space Complexity:

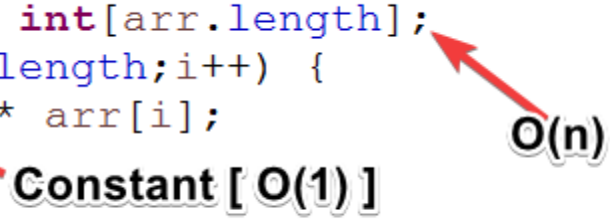
- Primitive data types have constant space.
- String require O(n) space (Where n is the String length)
- Collection types are generally O(n), where n is the length (for list type)
- The number of fields(for Object)

Let's take an example:



So, in this example we are concern about space, not time. Here we have **O(1)** space.

```
public static int[] doubleIt(int[] arr) {
    int[] newArr = new int[arr.length];
    for(int i=0;i<arr.length;i++) {
        newArr[i] = 2 * arr[i];
    }
    return newArr;
}
```



The Big O Notation in Space complexity for the above example is **O(n) .**

Guidelines for Asymptotic Analysis

Loops: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
//executes n times
for(i=1;i<=n;i++){
    m = m+2; //constant time, c
}
m= m+2; //contant time, c
```

Total time = a constant c * n = c n=O(n).

Nested loops: Analyze from inside out. Total amount time is the product of the sizes of all the loops.

```
//outer loop executed n times
for(int i = 1; i<=n;i++){
    //inner loop executed n times
    for(int j = 1; j <= n;j++){
        k = k + 1; //constant time
    }
}
```

Consecutive Statements: Add the time complexities of each statement.

```
x = x + 1; //constant time
//executed n times
for(i = 1; i < n; i++){
    m = m + 2; //constant time
}

//outer loop executed n times
for(i = 1; i <= n; i++){
    //inner loop executed n time
    for(j = 1; j <= n; j++){
        k = k + 1; //constant time
    }
}
```

Total time = $c_0 + c_1n + c_2n^2 = O(n^2)$

If-then-else statements: Worst-case running time: the test, plus either the then part or the else part (whichever is the larger).

```
//test: constant
if(length()==0){
    return false; //then part: constant
}
else{ //else part: (constant + constant) * n
    for(int n = 0; n < length(); n++){
        //another if: constant + constant (no else part)
        if(!list[n].equals(otherList.list[n]))
            //constatnt
            return false;
    }
}
```

Logarithmic complexity: An algorithm is O(logn) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
for(i = 1;i < n;)
i = i * 2;
```


Email: aatish.azad7@gmail.com