In [ ]:
```
# where do people drop out ?
# Should I watch old videos ?
# Learn how to learn
# State of mind: Flow
# What are you afraid of ? Why ? What's holding you back ?
# Surround yourself with positive people.
# What language should I use.
```

Introduction to Complexity: Time and Space

- Notion of time and CPU
- Adding two integers
- Adding two arrays of integers
- GPU

In [1]:
```
a1 = [1,2,3]
a2 = [4,5,6]

N

a3 = [5,7,9]

O(N)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
C:\Users\LEANGA~1\AppData\Local\Temp/ipykernel_37348/3041106453.py in <module>
      2 a2 = [4,5,6]
      3 
----> 4 N
      5 
      6 a3 = [5,7,9]

NameError: name 'N' is not defined
```

Finding an element in array

- By Value
- By Index

In [ ]:

Time complexity

In computer science, the time complexity is the computational complexity that describes the amount of **computer time** it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. Thus, the amount of time taken and the number of elementary operations performed by the algorithm are taken to be related by a constant factor.

Source: https://en.wikipedia.org/wiki/Time_complexity (https://en.wikipedia.org/wiki/Time_complexity)

In [ ]:

Why Complexity Analysis

Figure out an estimate of how the execution time gets impacted as the size of input data increases.
Rate of growth

In [ ]:

Example

Finding first duplicate in an array

- Simple approach
- Approach with a constraint that number of elements is limited to range 1-100

In [ ]:
```
Array of numbers. numbers are in integer range. Array contains exactly 1 duplicate value
Find out the duplicate element

[1 2 3 4 5 3]
[1 1]
[-1 2 3 3 4 -5]
```

In [ ]:
~~~
```C++

int arr[10];

class Solution {
public:
    int containsDuplicate(vector<int>& nums) {

        // O(N^2)
        for (int i = 0; i < nums.size(); i ++) { // O(N)

            // O(N)
            for (int j = i + 1; j < nums.size(); j++) { // O(N)
                if (nums[i] == nums[j]) { // O(1)
                    return nums[i];
                }
            }
        }
    }
};


```
~~~

In [3]:
```
print(1*3)
```

```
3
```

Calculating time complexity

- why to drop off smaller terms ?

In [2]:
```
# Comparing contribution of different steps in TC calculation
def n_square(n):
    return n*n*0.5

def n_(n):
    return n

def const():
    return 3/2

def poly(n):
    return n_square(n) + n_(n) + const()

for n in [1, 10, 100, 1000, 10000, 100000]:
    print("%6d poly=%-10.0f n^2=%-10.1f n=%-10.0f const=%-10.1f" % (n, poly(n), n_square(n), n_(n), const()))

# O(n^2) -> O n square
```

```
     1 poly=3          n^2=0.5        n=1          const=1.5
    10 poly=62         n^2=50.0       n=10         const=1.5
   100 poly=5102       n^2=5000.0     n=100        const=1.5
  1000 poly=501002     n^2=500000.0   n=1000       const=1.5
 10000 poly=50010002   n^2=50000000.0 n=10000      const=1.5
100000 poly=5000100002 n^2=5000000000.0 n=100000     const=1.5
```

In [ ]:

- O(1)
- O(log N)
- O(N)
- O(N log N) => O(N * log N)
- O(N^2)
- O(2^N)
- O(N!)

where N is the size of input/data`

```
In [ ]:
```

Example-0

```
n

for (i = 0 ; i < n; i++) {
    cout << i;
}
```

Example-1

```
n = 5

for (i = 0; i < n; i++) {

    for(j = 0; j < n; j++) {
        // some op // O(1)
    }
}

0 (0-n), 1(0-n), ... n(0-n)
n^2
```

Example-2

```
for (i = 0; i < n; i++) {

    for(j = 0; j <= i; j++) {
        // some op
    }
}

O(n^2)
1 + 2 + 3 + 4 + 5.... n


N(N+1)/2 => 0.5N^2 + 0.5N
O(N^2)
```

Example-3

```
for (i = 0; i < n; i++) { // N

    for(j = 0; j < n; j++) { // N

        for (k = 0; k < n; k++) { // N
                // some op
        }

    }
}

O(N^3)
```

Example-4

```
for (i = 0; i < n; i++) { // N

    for(j = 0; j < n; j++) { /// N
        // some op1
    }

    for (k = 0; k < n; k++) { // N
        // some op2
    }

}


N*2N => 2N^2 => O(N^2)
```

Example-4.1

```
for (i = 0; i < n; i++) { // N
    FUNCTION() // o(1)
}
O(N)


for (i = 0; i < n; i++) { // N
    FUNCTION() // o(n)
}
O(N^2)
```

Example-5

```
for (i = 0; i < n; i++) { // N

    // N^2
    for(j = 0; j < n; j++) { // N
            for (k = 0; k < n; k++) {  // n
                // some op1
            }
    }

    // N
    for (k = 0; k < n; k++) {
        // some op2
    }

}
```

N*(N^2 + N) = N^3 + N^2 => O(N^3)

Example-6

```
for (i = 0; i < n; i*=2) {
    // some op1
}

Infinite
```

Example-6.1

```
for (i = 1; i < n; i*=2) {
    // some op1
}

O (log N)
```

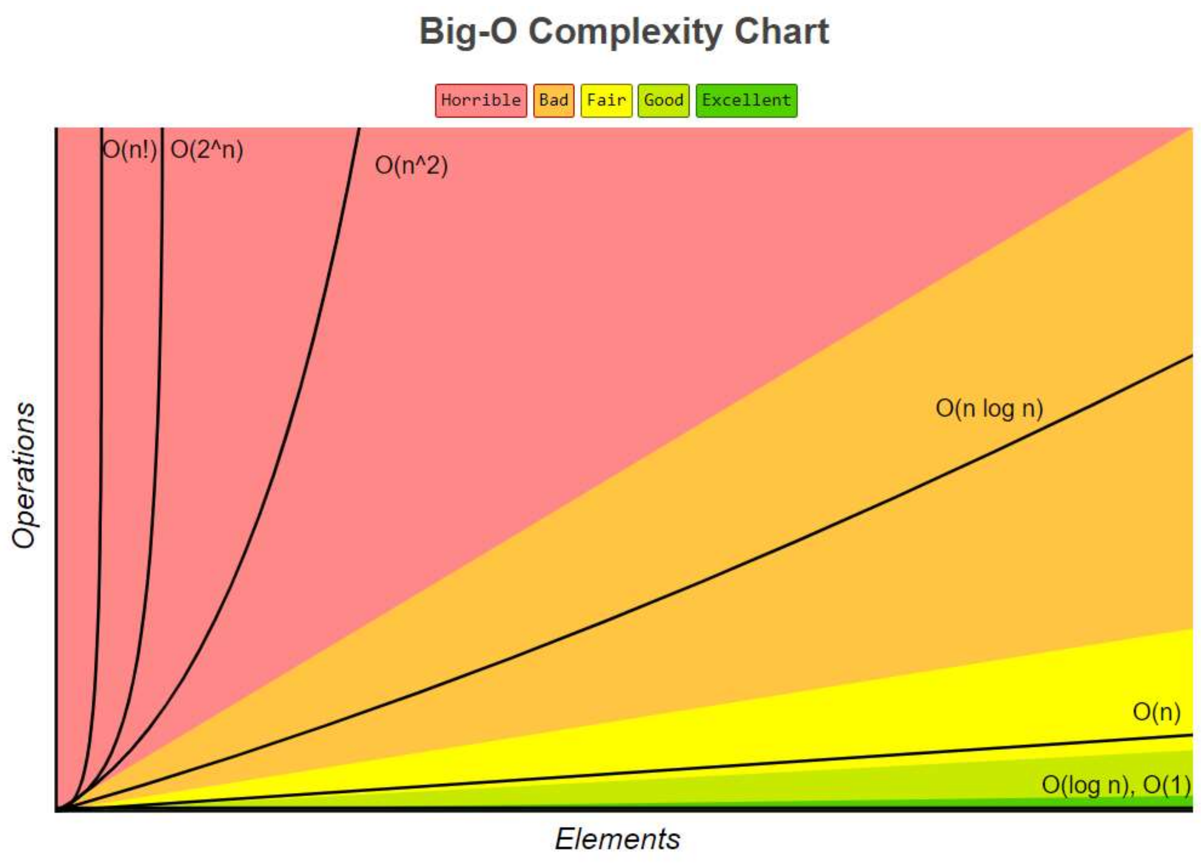
Example-7

```
for (i = 0; i < n; i+=2) {
    // some op1
}
20

0 2 4 6 8 10
N/2 = (1/2) * N = O(N)
```

Example-8

```
for (i = 0; i < n; i+=1) { // N
    for (j = 1; j < n; j=j*2) { // log N
        // some op1
    }
}

O(N * log N)
```

**Example-



Source: [Big-O Cheetsheet (https://www.bigocheatsheet.com/)](https://www.bigocheatsheet.com/)

```
In [ ]:
```

```
In [ ]:
```

What is a data structure

Data structure defines the way data is organized in the system/program memory.
DS provides interfaces for insertion, deletion and updation of data.
DS defines the way data is going to be arranged in memory and hence it also defines how the above operations are going to be implemented.
Example: Array, Stack, Queue, Tree, Graph, Hashmap etc.

```
In [ ]:
```

Understanding Hash Map data structure

```
In [ ]:
```

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)