Time-complexity

Time complexity is a measure used in computer science and algorithm analysis to estimate how the running time of an algorithm increases with the input size. It helps us understand the efficiency of an algorithm and how it scales as the problem size grows.

Time complexity is often expressed using Big O notation, which provides an upper bound on the growth rate of an algorithm's running time.

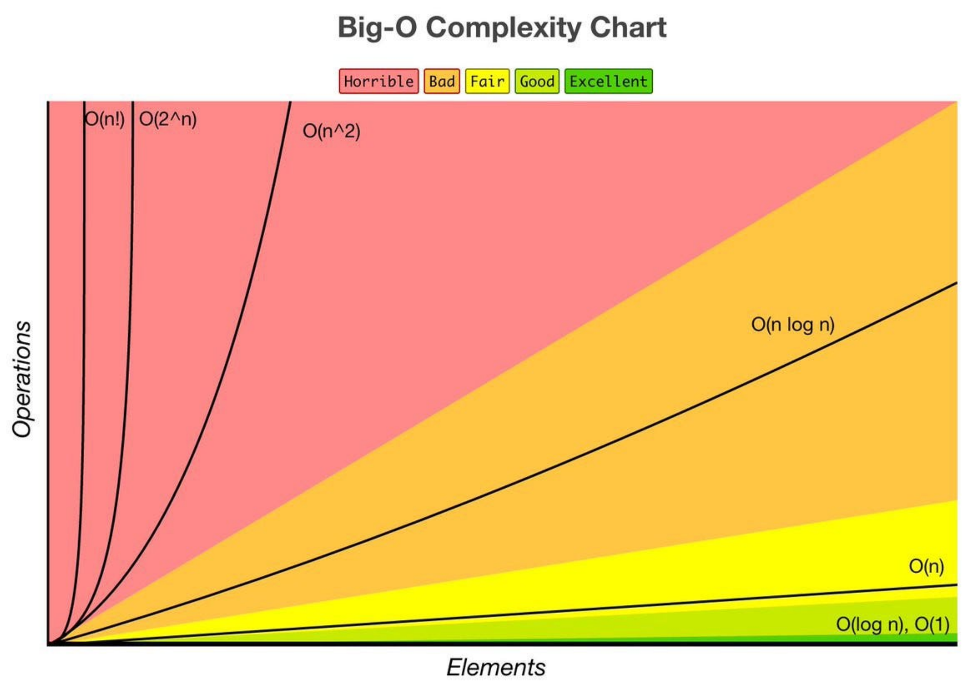
In Big O notation, the time complexity of an algorithm is denoted **O(f(n)), where f(n)** is a **function representing the growth rate of the algorithm's running time concerning the input size "n."**

concept
the amount of time taken to run an algorithm **(estimated)**, as a function of the **length of the input**

```

The complexity is a theoretical measurement to get an idea of how an algorithm
slow down on
bigger inputs compared to smaller inputs or compared to other algorithms.

: AbcAeffchen

```

The growth rate algorithm

The growth rate for an algorithm is the rate at which **the cost of the algorithm grows as the size of its input grows**

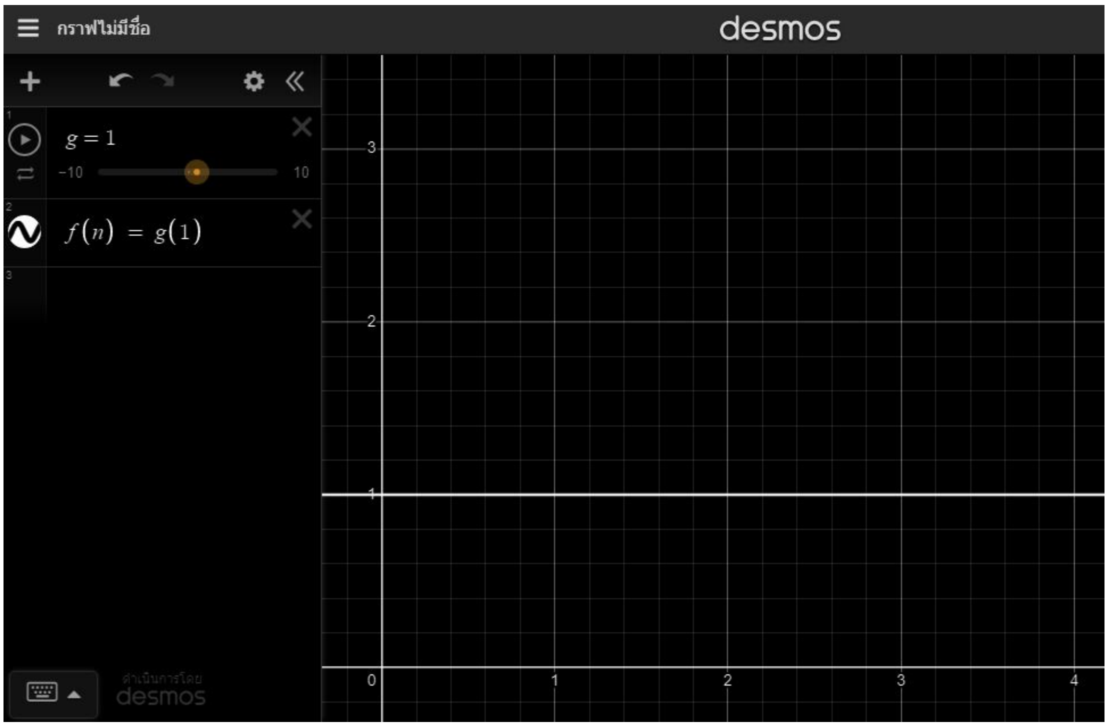


https://miro.medium.com/max/1400/1*j8fUQjaUlmrQEN_udU0_TQ.jpeg

O(1) - Constant

Constant Time Complexity: The algorithm's running time does not depend on the input size. It performs the same number of operations regardless of the input size.

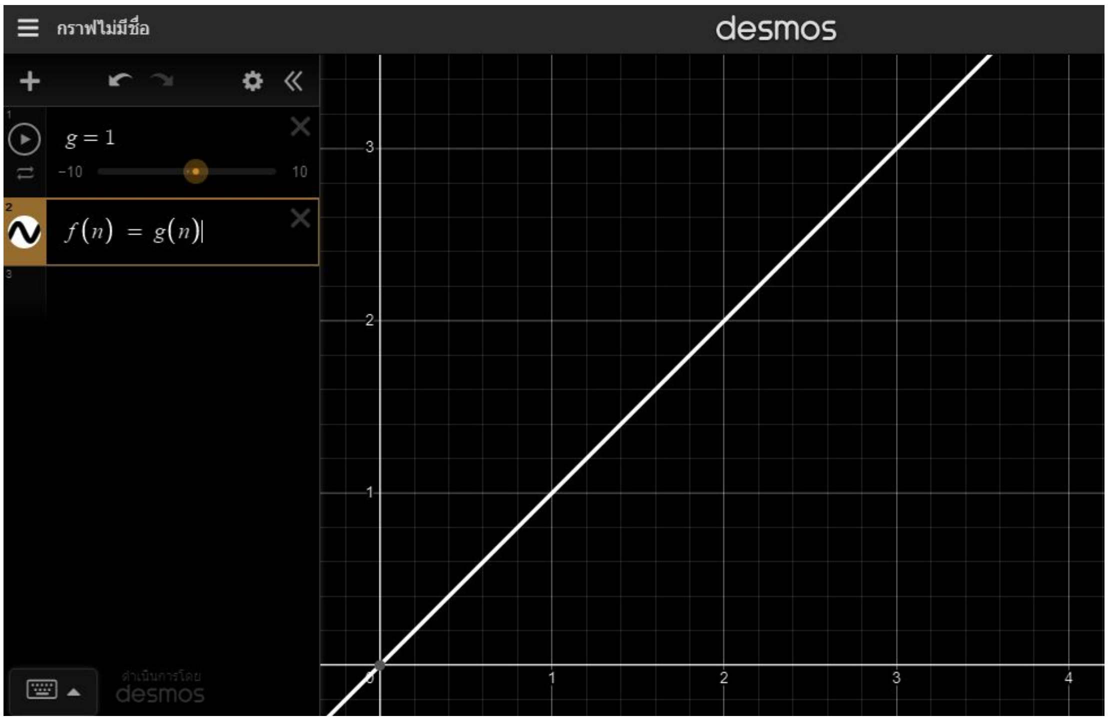
```
    s = [ -1000000, -999999, ..., 0, ..., 999999, 10000000 ]

    const len = (str) => {return str.length};

    const isBool = (value) => {return value ? true : false};

    const isString = (value) => {return isNaN(value) ? true : false};

    const searchItem = (index) => {return [1, 'a', 2, 'b', 100000000][index]};
```



O(n) - Linear

Linear Time Complexity: The algorithm's running time grows linearly with the input size. The number of operations increases linearly with the input.

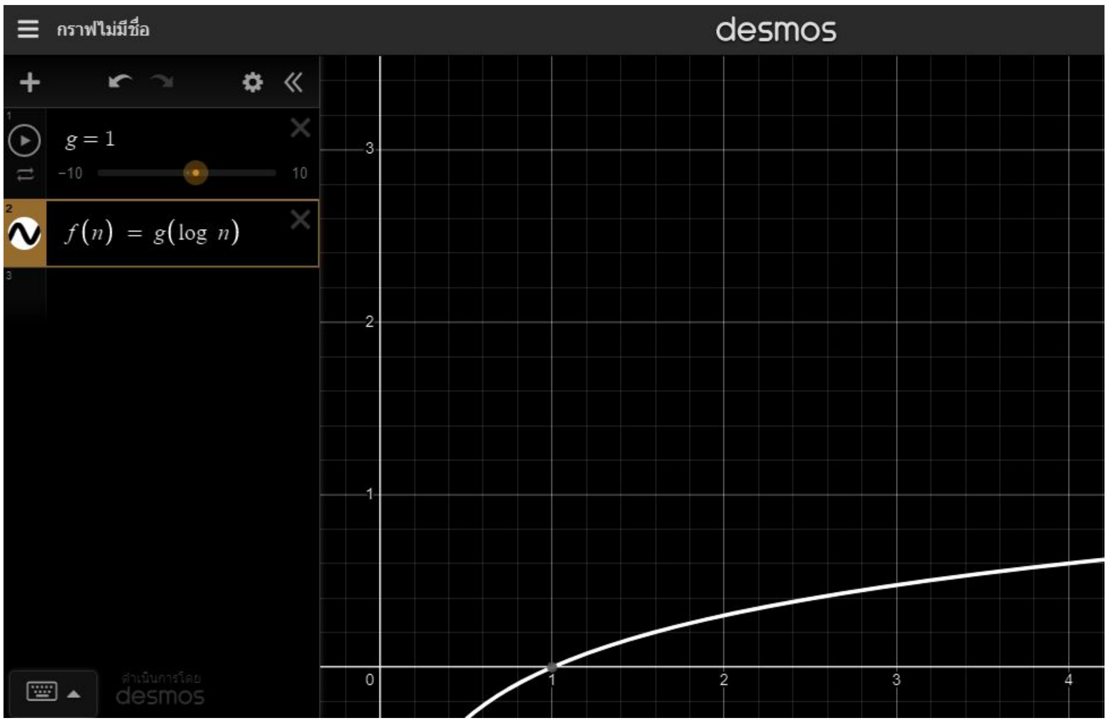
```
for(let i = 0; i < 10; i++){ // length = 10
    console.log(i) // 0 1 2 3 4 5 6 7 8 9
}


for(let i = 10; i >= 0; i--){ // length = 11
    console.log(i) // 10 9 8 7 6 5 4 3 2 1 0
}


for(let i = 10; i > 0; i-=2){ // length = 9
    console.log(i) // 10 8 6 4 2
}


for(let i = 1; i <= 10; i+=4){ // length = 10
    console.log(i) // 1 5 9
}
```



O(log n) - Logarithmic

Logarithmic Time Complexity: The algorithm's running time grows logarithmically with the input size. Algorithms that **using multiplication or division**

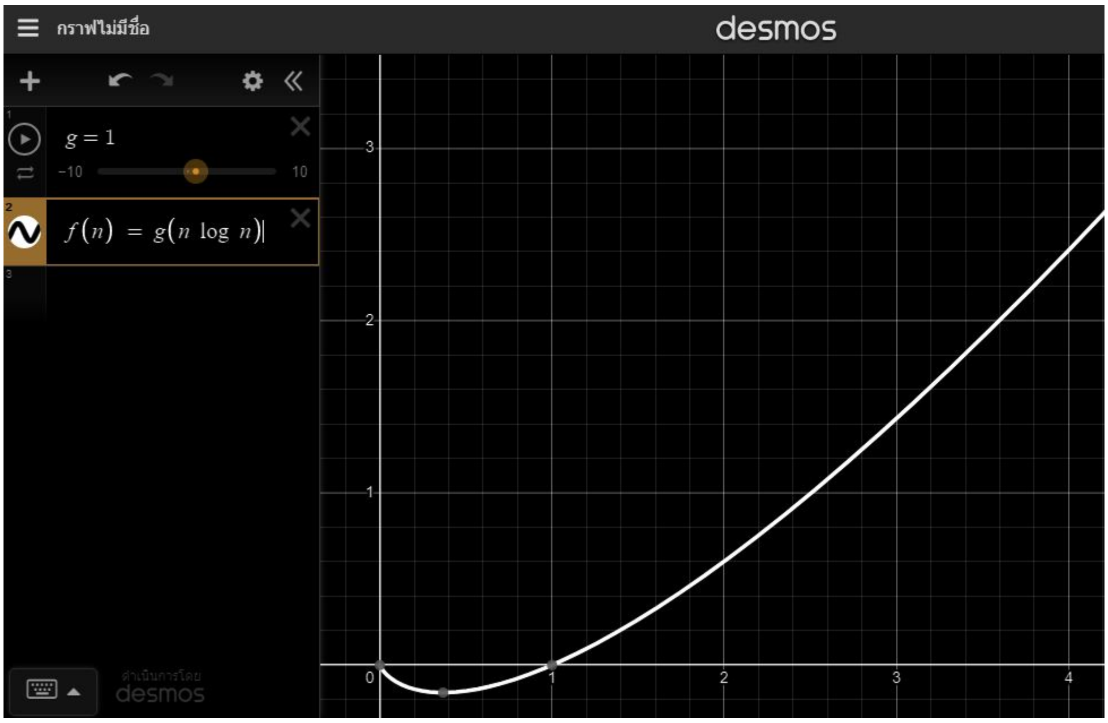
```
let n = 16;
let count = 0;
function logFunc(n){
    if(n <= 1){ return `Done in ${count} times`; }
    n = Math.floor(n / 2);
    count++;
    return logFunc(n);
}

function binarySearch(arr, target){
    let left = 0;
    let right = arr.length - 1;

    while(left <= right){
        let mid = Math.floor((left + right) / 2);

        if(arr[mid] === target){
            return mid; // Found the target at index mid
        } else if(arr[mid] < target){
            left = mid + 1; // Search the right half
        } else {
            right = mid - 1; // Search the left half
        }
    }
    return -1; // Target not found in the array
}
```


กราฟไม่มีชื่อ
desmos
$g = 1$
−10
10
$f(n) = g(\log n)$
ดำเนินการโดย desmos
3
2
1
0
1
2
3
4

O(n * log n) - Log Linear

Linearithmic Time Complexity: The algorithm's running time grows between linear and logarithmic. Common in many efficient sorting and searching algorithms like Merge Sort and Quick Sort.

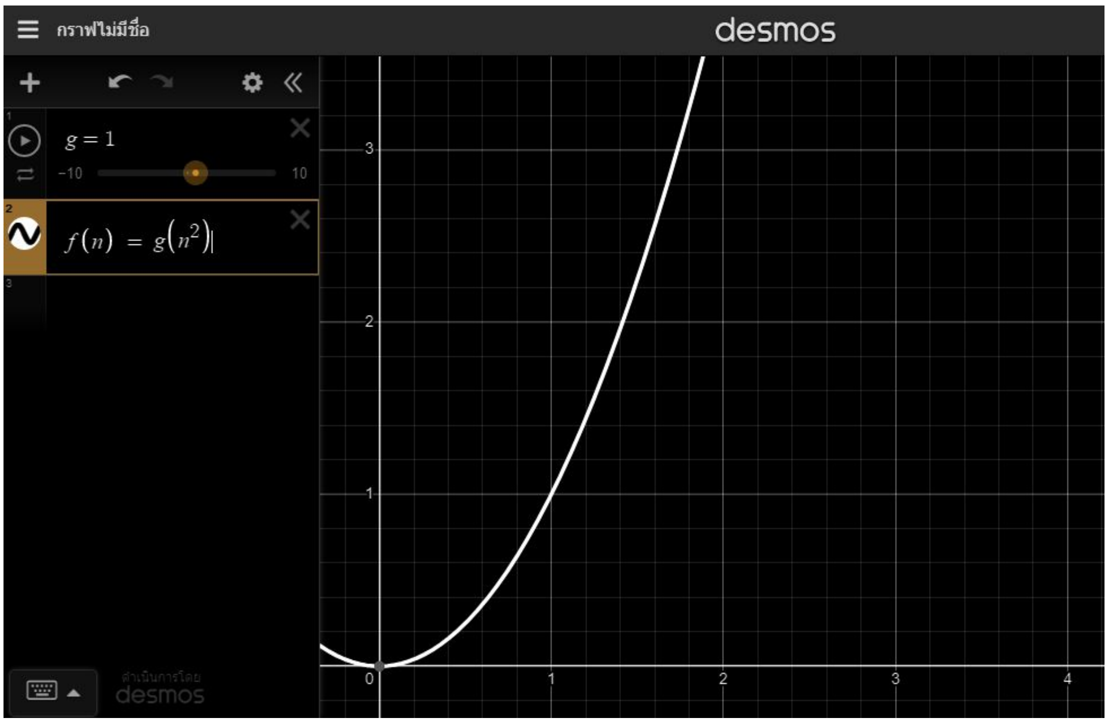
```
    for(let i = 0; i < 5; i++){
        for(let j = 1; j < 5; j*=2){
            console.log(i, j);
            // 0 1
            // 0 2
            // 0 4
            // 1 1
            // 1 2
            // 1 4
            // 2 1
            // 2 2
            // 2 4
            // 3 1
            // 3 2
            // 3 4
            // 4 1
            // 4 2
            // 4 4
        }
    }
```



O(n^2) - Quadratic

Quadratic Time Complexity: The algorithm's running time grows quadratically with the input size. It generally involves nested loops.

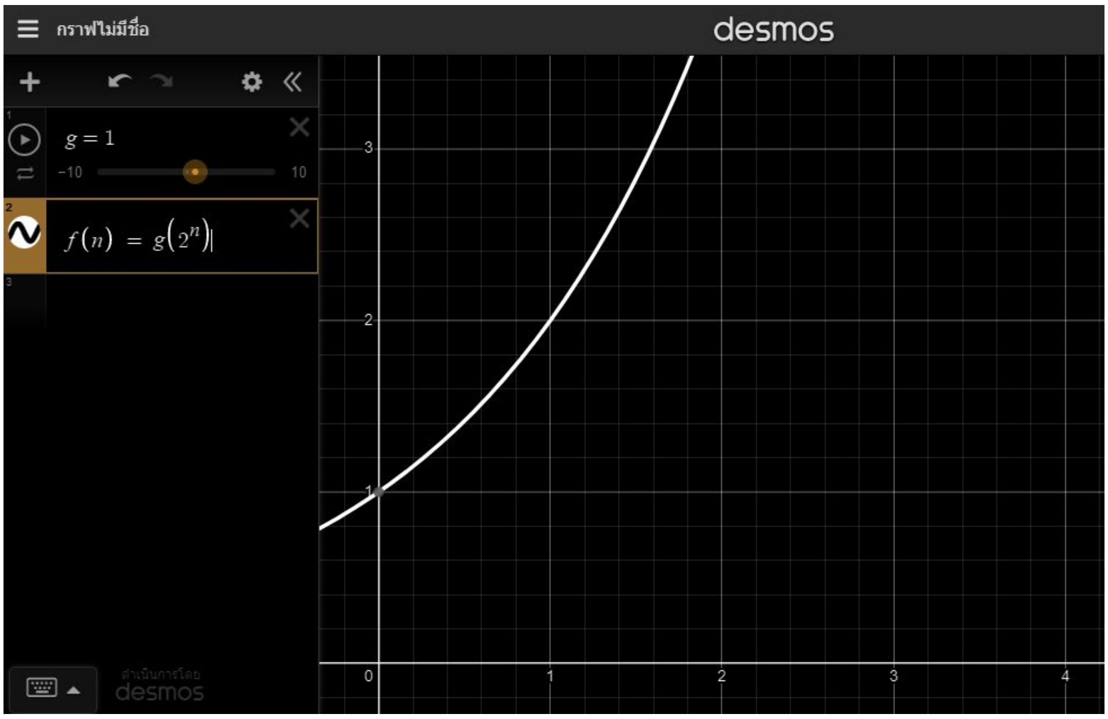
```
    for(let i = 0; i < 5; i++){
        for(let j = 0; j < 4; j++){
            console.log(i, j)
            // 0 0
            // 0 1
            // 0 2
            // 1 0
            // 1 1
            // 1 2
            // 2 0
            // 2 1
            // 2 2
            // 3 0
            // 3 1
            // 3 2
            // 4 0
            // 4 1
            // 4 2
        }
    }
```



O(2^n) - Exponential

> The pattern to watch for is that if a recursive function makes more then one call, the complexity is often **O(branches^{depth}), where branches refers the number recursive calls made** and **the depth refers to the depth of the tree this creates.** : Joseph Trettevik

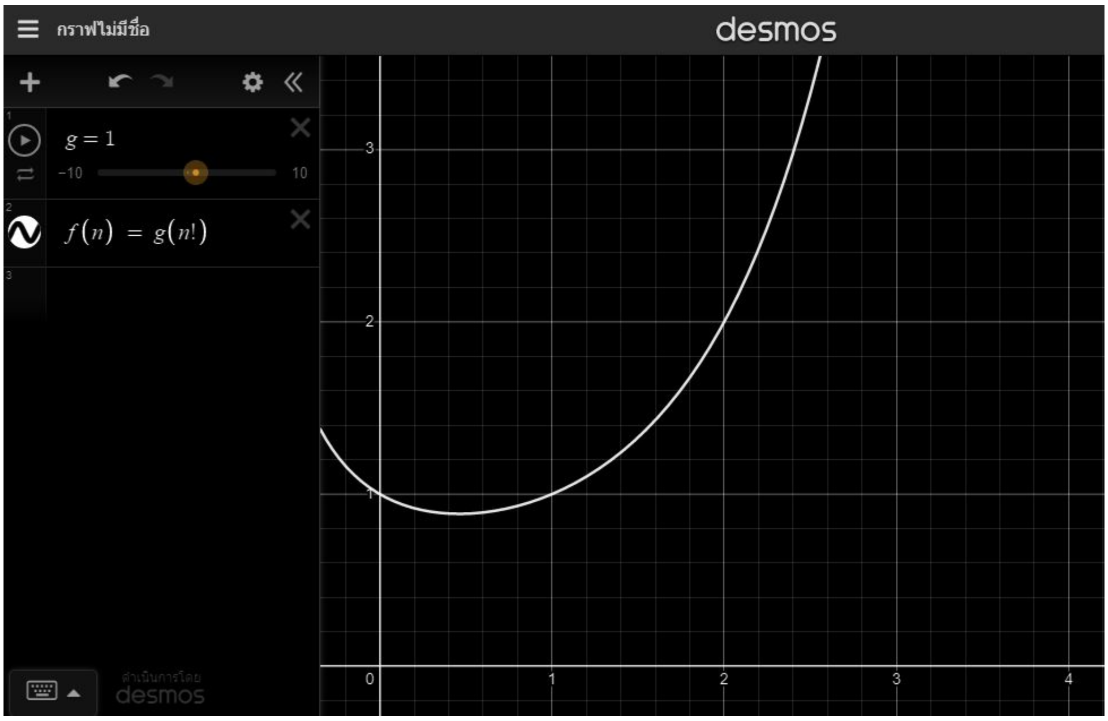
```
Power set
s = [1, 3, 5]
n(s) = 3
P(n(s)) = 2^3 = 8

s = [5, 8, 12, 1, 6]
n(s) = 5
P(n(s)) = 2^5 = 32

function fibo(n){
    if(n === 1 || n === 0){ return n; }
    return fibo(n - 1) + fibo(n - 2); // 2 branches
}
fibo(1): 1          execute 1 times              depth lv. 0 = 1
fibo(2): 1          execute 3 times              depth lv. 1 = 2
fibo(3): 2          execute 5 times              depth lv. 2 = 4
fibo(10): 55        execute 177 times            depth lv. 9 = 512
//               _____Execute time______     ________Depth level________
// lv 0          |          f(3)       |     |          f(3)           |
//               |        /       \    |     |       /         \       |
// lv 1          |     f(2)      f(1)  |     |     f(2)        f(1)    |
//               |    /   \            |     |    /   \        /   \   |
// lv 2          |  f(1) f(0)          |     |  f(1)  f(0)  f(0) f(-1) |
//               |__stop at condition__|     |_____count leaf node_____|
```



O(n!) - Factorial

> If there are N cities, **the brute force method will try each and every permutation** of these N cities. #codaddict

```
let allExecute = 0;
let t0 = 0;

function nFactorial(n){
    t0 = performance.now();
    let num = n;
    allExecute++;
    if(n === 0){
        return 1;
    };

    for(let i = 1; i <= n; i++) {
        num = n * nFactorial(n - 1);
    }
    return num;
}
let t1 = performance.now();

nFactorial(1): 1      0.045 milliseconds    1 branches      execute 2 times
nFactorial(2): 2      2.714 milliseconds    2 branches      execute 5 times
nFactorial(5): 120    3.356 milliseconds    120 branches    execute 326 times
nFactorial(7): 5040   12.75 milliseconds    5040 branches   execute 13700 times
```



Primitive Operations (POs)

```
In computer science and algorithm analysis, primitive operations refer to the
basic and fundamental operations that an algorithm or program performs. These
operations are considered to have constant time complexity and are used to measure
the efficiency of an algorithm in terms of its execution time.

The choice of primitive operations might vary depending on the context, but some
common examples of primitive operations include:

1. Arithmetic Operations: Addition, subtraction, multiplication, and division of
simple data types like integers, floating-point numbers, etc.
2. Assignment: Assigning a value to a variable.
3. Comparison: Comparing two values to check for equality, inequality, greater
than, less than, etc.
4. Memory Access: Reading from or writing to a specific memory location.
5. Control Flow: Basic control structures like branching (if-else statements) and
looping (for, while, do-while loops).
6. Function Calls: Invoking a function or method.
7. Pointer Operations: Operations on pointers such as dereferencing or getting the
address of a variable.
8. Array Access: Accessing elements in an array using an index.

When analyzing the time complexity of an algorithm, it is often assumed that these
primitive operations take constant time (O(1)) because they are considered
elementary and do not depend on the size of the input data. However, it's
essential to consider that the actual time taken by these operations might vary
depending on hardware architecture, compiler optimizations, and other low-level
factors.

By counting the number of primitive operations executed by an algorithm as a
function of the input size, we can determine its time complexity and understand
how the running time scales with larger inputs. This analysis helps in comparing
and selecting the most efficient algorithm for a given problem.

ChatGPT
```

> My summary : Count **No. of operations** that the algorithm is performing **at worst-case.** You may found a variety of counting methods. **Keep reminding that counting is only an approximation for make the performance of the function can be measured.**

Counting concept

POs (Primitive Operations)	expression	constant factor
Assigned a value to variable	v = 5	1
Calling a method	function()	1
Performing an arithmetic operations (+ - * / %)	1 + 2	1
Comparing two numbers	1 < 2 , 1 === 2	1
Indexing into an array	array[i]	1
Returning from a method	return	1

Single line code

```
let num = 0;            // POs is assignment = 1
console.log(num)        // POs is only log = 1
let sum = 1 + 2         // POs is arithmetic operations and assign = 2
console.log(1 < 2)      // POs is comparison and log = 2
```

Iteration loop

has 3 tasks to processing ***initial value***, ***condition***, ***updation***

```
for(initial value; condition; updation){
    /*body statement*/
}
```

```
initial value                           initial value
while(condition){                       do{
    /*body statement*/                     /*body statement*/
    updation                               updation
}                                       } while(condition);
```

Excepted These loops are execute n times, there are no updation and condition.

```
for(value of items){                    for(value in items){
    /*body statement*/                      /*body statement*/
}                                       }
```

Calculate iteration

1. **Initial value** is init value to starting count each loop.
2. **Condition** is **the determinants No. of iterations from initial value until whatever condition has setted.** If working in a large system or with a database, perhaps we cannot know correct of No. of iterations so for simplicity will **called that n times** (or any) and **n** will related to the condition that executed is true and at the last of every single iteration the condition always executed to be **false 1 time for exit the iterations.** So general form is **n + 1**

> Note: every example in this topic
>
> - use update just only i++ and i--
> - use absolute value because it's best way to counting for this case
> - for to show formula structure and how to use it In general will use updation

- **if either of initial value or condition is 0**
  - if condition use **>** or **<** : No. of iterations is **n + 1** | n is initial value or condition not equals to 0 and turns negative sign to positive sign.
    - i = 0; i < 3; i++
    - i = 3; i > 0; i--
    - i = -3; i < 0; i++
    - i = 0; i > -3; i--

    > All of this No. of iterations is **3 + 1**

  - if condition use ≥ or ≤ it's mean No. of iterations is **(n + 1) + 1** | n is initial value or condition not equals to 0 and turns negative sign to positive sign.
    - i = 0; i <= 3; i++
    - i = 3; i >= 0; i--
    - i = -3; i <= 0; i++
    - i = 0; i >= -3; i--

    > All of this No. of iterations is **(3 + 1) + 1**

- **if either of initial values or condition ≠ 0 No. of iterations is (n - initial value) + 1**
  - if condition use ≥ or ≤ the condition have to shift ceiling so **n** is
    - condition + 1 : if updation is plus.
    - condition - 1 : if updation is minus.

	-3	-2	-1	0	1	2	3	4	No. of interations
i = 1; i < 3; i++	-	-	-	-	✓	✓	✗	-	|3 - 1| + 1
i = -3; i <= 2; i++	✓	✓	✓	✓	✓	✓	✗	-	|(2 + 1) - (-3)| + 1
i = 4; i >= 1; i--	-	-	-	✗	✓	✓	✓	✓	|(1 - 1) - 4| + 1
i = 1; i >= -2; i--	✗	✓	✓	✓	✓	-	-	-	|((-2) - 1) - 1| + 1

3. **Updation** determines the type of function it is implied when value **≠ 0 or 1** because updation use for **divisor of n. And if result of division turns out to decimal use only the integer then plus 1**

   - If updation is **plus or minus called linear growth.** Formula is

$$POs \cdot \left(\frac{condition - initial\ value}{updation}\right) + 1$$

example	**i = 0; i < 10; i+=2**	**i = 2; i <= 10; i+=2**	**i = 10; i > 2; i-=2**	**i = 10; i >= 2; i-=2**
initial and condition	0 < 10	2 <= 10	10 > 2	10 >= 2
updation value	2	2	-2	-2
No. of iterations	$(\frac{10 - 0}{2}) + 1$	$(\frac{(10 \mathbf{+ 1}) - 2}{2}) + 1$	$(\frac{2 - 10}{-2}) + 1$	$(\frac{(2 \mathbf{- 1}) - 10}{-2}) + 1$
is decimal	-	4 + 1	-	4 + 1
true	0, 2, 4, 6, 8	2, 4, 6, 8, 10	10, 8, 6, 4	10, 8, 6, 4, 2
false	10	12	2	0

   - If updation is **mutiply or division called logarithmic growth.** Use **updation to be logarithm base** and the either of initial value or condition where **largest value to be dividend and smallest to be divisor.** Formula is

$$\log_{updation} \frac{largest\ value}{smallest\ value}$$

     - if condition use ≥ or ≤ the condition have to shift ceiling so n is
       - condition * updation : if updation is multiplication.
       - condition / updation : if updation is division.

example	**i = 2; i < 32; i*=2**	**i = 2; i <= 32; i*=2**	**i = 12; i > 3; i/=2**	**i = 12; i >= 3; i/=2**	**i = 12; i >= 3; i/=2**
initial and condition	2 < 32	2 <= 32	12 > 3	12 >= 3	12 >= 3
updation value	2	2	2	2	2
No. of iterations	$\log_2 \frac{32}{2}$	$\log_2 \frac{32 \mathbf{* 2}}{2}$	$\log_2 \frac{12}{3}$	$\log_2 \frac{12}{3 \mathbf{/ 2}}$	$\log_{\frac{1}{2}} \frac{3}{12 \mathbf{/ \frac{1}{2}}}$
is decimal	-	-	-	-	-
true	2, 4, 8, 16	2, 4, 8, 16, 32	12, 6	12, 6, 3	12, 6, 3
false	32	64	3	1.5	1.5

Lesser than

	i = 0; i < n; i++	i = 3; i < n; i+=2	i = -3; i < n; i+=5	i = 1; i < n; i*=2
condition	$(\frac{n - (0)}{1}) + 1$	$(\frac{n - (+3)}{2}) + 1$	$(\frac{n - (-3)}{5}) + 1$	$(\log_2 \frac{n}{1}) + 1$
updation	$2 \cdot \frac{n - (0)}{1}$	$2 \cdot \frac{n - (+3)}{2}$	$2 \cdot \frac{n - (-3)}{5}$	$(\log_2 \frac{n}{1})$

Lesser than or equal

	i = 0; i <= n; i++	i = 3; i <= n; i+=2	i = -3; i <= n; i+=5	i = 2; i <= n; i*=4
condition	$(\frac{(n + 1) - (0)}{1}) + 1$	$(\frac{(n + 1) - (+3)}{2}) + 1$	$(\frac{(n + 1) - (-3)}{5}) + 1$	$(\log_4 \frac{n * 4}{2}) + 1$
updation	$2 \cdot \frac{(n + 1) - (0)}{1}$	$2 \cdot \frac{(n + 1) - (+3)}{2}$	$2 \cdot \frac{(n + 1) - (-3)}{5}$	$(\log_4 \frac{n * 4}{2})$

Greater than

	i = 5; i > n; i--	i = 4; i > n; i-=2	i = 3; i > n; i-=4	i = 7; i > n; i/=2
condition	$(\frac{n - (+5)}{-1}) + 1$	$(\frac{n - (+4)}{-2}) + 1$	$(\frac{n - (+3)}{-4}) + 1$	$(\log_2 \frac{7}{n}) + 1$
updation	$2 \cdot \frac{n - (+5)}{-1}$	$2 \cdot \frac{n - (+4)}{-2}$	$2 \cdot \frac{n - (+3)}{-4}$	$(\log_2 \frac{7}{n})$

Greater than or equal

	i = 5; i >= n; i--	i = 4; i >= n; i-=2	i = 3; i >= n; i-=4	i = 32; i >= n; i/=4
condition	$(\frac{(n - 1) - (+5)}{-1}) + 1$	$(\frac{(n - 1) - (+4)}{-2}) + 1$	$(\frac{(n - 1) - (+3)}{-4}) + 1$	$(\log_4 \frac{32}{n / 4}) + 1$
updation	$2 \cdot \frac{(n - 1) - (+5)}{-1}$	$2 \cdot \frac{(n - 1) - (+4)}{-2}$	$2 \cdot \frac{(n - 1) - (+3)}{-4}$	$(\log_4 \frac{32}{n / 4})$

Counting with POs

```
ex1 for(let i = 0;    i < n;      i++) { console.log(i); }
            ▼         ▼          ▼           ▼
    POs   =   1    +  (n + 1)   +  2(n)     +     n     =   4n + 2
```

```
ex2 for(let i = 1;         i <= n;          i++) { console.log(i); }
            ▼               ▼                ▼          ▼
    POs   =   1   +  (((n + 1) - 1) + 1)  +  2(n)   +   (n)   =   4n + 2
```

```
ex3 for(let i = -3;          i <= n;              i++) { console.log(i); }
            ▼                 ▼                ▼             ▼
    POs   =   1   +  (((n + 1) - (-3)) + 1)  +  2(n + 4)  +  (n + 4)   =   4n + 18
```

```
ex4 for(let i = 10; i > n; i--) { console.log(i); }
    POs   =   1 + (((n - 10) / (-1)) + 1) + 2((n - 10) / (-1)) + (n - 10) / (-1)
          =   4n + 42
```

```
ex5 for(let i = 10; i >= n; i-=2) { console.log(i); }
    POs   =   1
              + ((((n - 1) - 10) / (-2)) + 1) + 2((n - 1) - 10) / (-2)
              + ((n - 1) - 10) / (-2)
          =   -2n + 24
```

```
ex6 for(let i = 0; i < n; i++) {        // 1 + (n + 1) + (n)
        // Outermost body is n
        // Innermost is body of outermost so innermost iterate n times
        for(let j = 10; j > n; j--) {   // 1 + n(((n - 10) / (-1)) + 1) + 2n((n -
10) / (-1))
            console.log(j);             // n(n - 10) / (-1)
        }
    }
    POs  =  1 + (n + 1) + (n) + 1 + n(((n - 10) / (-1)) + 1)
            + 2n((n - 10) / (-1)) + n(n - 10) / (-1)
         =  (4n^2 - 43n - 3) / (-1)
```

> Note : let i count only 1 beacause that assigning 0 at start loop only and i++ is already includes assigning i after increment.

How many of i modulo n equals to 1 countMod(num)

1. SET count to be 0 (storing result count default is 0)
2. LOOP i to num - 1
   - IF num % i EQUAL TO 1
     - THEN STATEMENT count + 1
   - STATEMENT i + 1
3. END LOOP
4. RETURN count

```
let num = 7
function countMod(num){
    let count = 0;                          1
    for(let i = 0; i < num; i++){           1 + 1(n + 1) + 2(n)
        if(num % i === 1){                  2(n)
            count++;                        2(n) // 2(1) for best case
        }
    }
    return count;                           1
}
```

let count	let i	i < num	num % i = 1	count++	i++	return count
0	0	true	7 % 0 = NaN		true	
	1	true	7 % 1 = 0		true	
	2	true	7 % 2 = 1	true	true	
	3	true	7 % 3 = 1	true	true	
	4	true	7 % 4 = 3		true	
	5	true	7 % 5 = 2		true	
	6	true	7 % 6 = 1	true	true	
	7	false				true
1	**1**	**1(n + 1)**	**2(n)**	**2(n)**	**2(n)**	**1**

POs	=	1 + 1 + n + 1 + 2n + 2n + 2n + 1
	=	7n + 4
O(f(n))	=	O(n)

Calc mean

calcMean(set_data)

1. SET sum to be 0 (sum of each element)
2. SET len to be length of set_data
3. LOOP i to len - 1
    - OPERATION summation of sum and current data value
    - STATEMENT i + 1
4. END LOOP
5. RETURN sum / len

```
    let set_data = [4, 6, 1, 5];
    function calcMean(set_data){
        let sum = 0;                      1
        let len = set_data.length;        2
        for(let i = 0; i < len; i++){     1 + 1(n + 1) + 2(n)
            sum += set_data[i];           3(n)
        }
        return sum / len;                 2
    }
```

sum	len = set_data.length	let i	i < len	sum += set_data[i]	i++	return sum / len
0	4	0	true	0 + 4 = 4	true	
		1	true	4 + 6 = 10	true	
		2	true	10 + 1 = 11	true	
		3	true	11 + 5 = 16	true	
		4	false			true
1	**2**	**1**	**1(n + 1)**	**3(n)**	**2(n)**	**2**

POs	=	1 + 2 + 1 + n + 1 + 3n + 2n + 2
	=	6n + 7
O(f(n))	=	O(n)

Print 5 natural numbers from given x

naturalNumber(x)

1. IF x not be a natural number
   - THEN
     - STATEMENT console.log
     - RETRUN
2. LOOP i = 1 to x
   - console.log i
   - STATEMENT i + 1
3. END LOOP

```
    let x = 5
    function naturalNumber(x){
        if(x < 1){                                              1
            return console.log(`${x} is not a natural number`);     1 + 1
        };
        for(let i = 1; i < x; i++){                             1 + 1((n - 1)
+ 1) + 2(n - 1)
            console.log(i);                                     (n - 1)
        }
    }
```

x < 1	return console.log	let i	i < x	console.log	i++
false		0			
		1	true	1	true
		2	true	2	true
		3	true	3	true
		4	true	4	true
		5	false		
1	**2**	**1**	**1((n - 1) + 1)**	**(n - 1)**	**2(n - 1)**

POs	=	1 + 1 + 1 + 1 + n - 1 + 1 + n - 1 + 2n - 2
	=	4n + 1
O(f(n))	=	O(n)

```
    x = 5
    function naturalNumber(x){
        if(x < 1){                                          1
            return console.log(`${x} is not a natural number`);     1 + 1
        };
        for(let i = x; i < x + 5; i++){                     1 + 2(((n + 5)
 - 5) + 1) + 2(n - 5)
            console.log(i);                                 (n + 5) - 5
        }
    }
```

x < 1	return console.log	i	i < x + 5	console.log	i++
		0			
		1			
		2			
		3			
		4			
false		5	true	5	true
		6	true	6	true
		7	true	7	true
		8	true	8	true
		9	true	9	true
		10	false		
1	**2**	**1**	**2(((n + 5) - 5) + 1)**	**(n)**	**2(n)**

POs	=	1 + 2 + 1 + 2(((n + 5) - 5) + 1) + n + 2n
	=	5n + 6
O(f(n))	=	O(n)

Find cartesian product of two sets A and B

cartesianProduct(A, B)

1. SET A to be set
2. SET B to be set
3. SET size_a to be length of A
4. SET size_b to be length of B
5. SET C to be cartesian product of A * B (default is empty set)
6. LOOP
    - OUTER LOOP i to size_a - 1
        - INNER LOOP j to size_b - 1
            - STATEMENT C APPEND A[i] and B[j]
7. END LOOP
8. RETURN C

```
let A = [1, 2, 3];
let B = ['a', 'b', 'c'];
function cartesianProduct(A, B){
    let size_a = A.length;                     2
    let size_b = B.length;                     2
    let C = [];                                1
    for(let i = 0; i < size_a; i++){           1 + (n + 1) + 2(n)
        for(let j = 0; j < size_b; j++){       1 + n(n + 1) + 2(n(n))
            C.push([A[i], B[j]]);              3(n(n))
        }
    }
    return C;                                  1
}
```

```
function cartesianProduct(A, B){
    let size_a = A.length;                     2
    let size_b = B.length;                     2
    let C = [];                                1
    for(let i = 0; i < size_a;){               1 + (n + 1)
        for(let j = 0; j < size_b;){           1 + n(n + 1)
            C.push([A[i], B[j]]);              3(n(n))
            j++;                               2(n(n))
        }
        i++;                                   2(n)
    }
    return C;                                  1
}
```

size_A = A.length	size_B = B.length	let i	i < size_A	let j	j < size_B	C.push([A[i], B[j]])	i++	j++	return C
3	3	0	true	0	true	[1, 'a']		true	
		0		1	true	[1, 'b']		true	
		0		2	true	[1, 'c']		true	
		0		3	false		true		
		1	true	0	true	[2, 'a']		true	
		1		1	true	[2, 'b']		true	
		1		2	true	[2, 'c']		true	
		1		3	false		true		
		2	true	0	true	[3, 'a']		true	
		2		1	true	[3, 'b']		true	
		2		2	true	[3, 'c']		true	
		2		3	false		true		
		3	false						true
2	**2**	**1**	**(n + 1)**	**1**	**n(n + 1)**	**3(n(n))**	**2(n)**	**2(n(n))**	**1**

POs	=	2 + 2 + 1 + 1 +n + 1 + 1 + n(n + 1) + 3n(n) + 2n(n) + 2n + 1
	=	$6n^2 + 4n + 9$
O(f(n))	=	$O(n^2)$

Nested linear and logarithm loop

nestedLinearLogarithmic(n)

1. SET total to be 0
2. LOOP
    - OUTER LOOP i to n + 1
        - STATEMENT SET j to be 1
        - INNER LOOP j to n
            - STATEMENT INCREMENT TOTAL BY PLUS 1
            - STATEMENT INCREMENT j BY MULTIPLY 2
3. END LOOP
4. RETURN total

```
    let n = 3;
    function nestedLinearLogarithmic(n){
        let total = 0;                      1
        // Outer loop (O(n))
        for(let i = 0; i < n + 1; i++){     1 + 2((n + 1) + 1) + 2(n + 1)
            let j = 1;                      1(n + 1)
            // body of outermost is (n + 1) represent to m
            // + 1 of outermost is not false it just n + 1

            // Inner loop (O(log n))
            // j is initial value for innermost
            // if result from log is decimal then use only integer then + 1
            while(j < n){                   m((log_2 n / 1) + 1)
                total += 1;                 2m(log_2 n / 1)
                j *= 2;                     2m(log_2 n / 1)
            }
        }
        return total;                       1
    }
```

Explain

- At outermost is linear loop it's basic to count, see at previous example before.
- At innermost is logarithmic loop
- (n + 1) is the **No. of iterations of the outermost loop** that the innermost loop must iterate to. **Represent to be m**
    - **Condition.** $m((\log_2 \frac{3}{1}) + 1)$ result is decimal then use only integer then + 1 and the last (+ 1) is false.
    - **Updation.** $2m(\log_2 \frac{3}{1})$
        - **body not execute at condition false** so get rid off + 1
        - 2 is POs of code body it iterations related to condition then multiply to condition.

total	let i	i < n + 1	let j	j < n	total++	j*=2	i++	return
true	0	true	0					
			1	true	true	true		
			2	true	true	true		
			4	false			true	
	1	true	0					
			1	true	true	true		
			2	true	true	true		
			4	false			true	
	2	true	0					
			1	true	true	true		
			2	true	true	true		
			4	false			true	
	3	true	0					
			1	true	true	true		
			2	true	true	true		
			4	false			true	
	4	false						true
1	**1**	**2((n + 1) + 1)**	**1(n + 1)**	**(n + 1)((log_2 n / j) + 1)**	**2(n + 1)(log_2 n / j)**	**2(n + 1)(log_2 n / j)**	**2(n + 1)**	**1**

POs	=	1 + 1 + 2n + 4 + n + 1 + $\frac{n\log_2 n + \log_2 n}{j}$ + n + 1 + $\frac{4\log_2 n * (n + 1)}{j}$ + 2n + 2 + 1
	=	6n + $\frac{5\log_2 n * (n + 1)}{j}$ + 11
O(f(n))	=	O(n * log n)

Don't forget it when division or logarithm returns out to be decimal use only integer then + 1

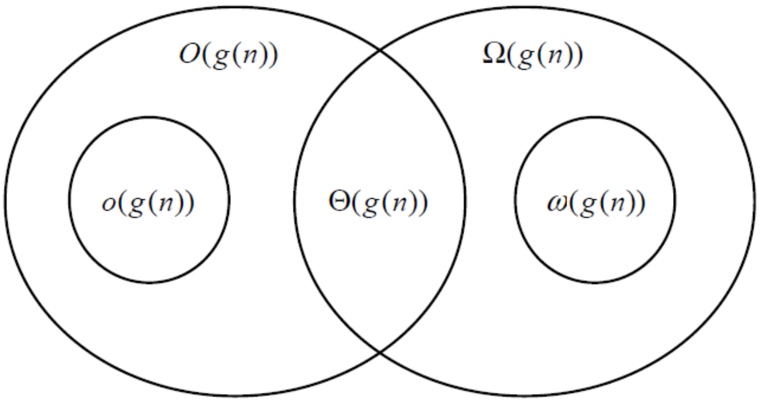
Measures

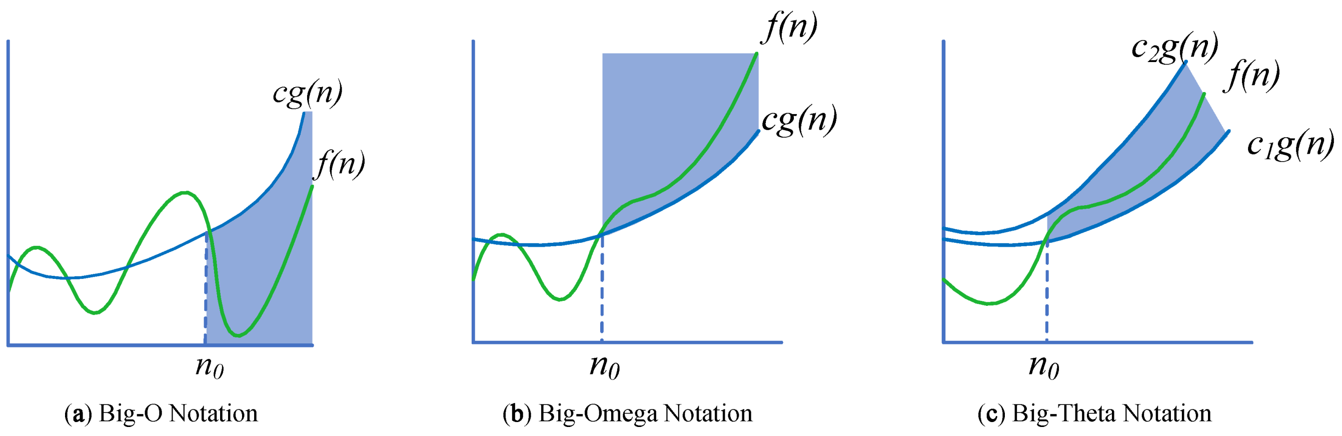
Lower bound ≤ f(n) ≤ Upper bound

It measures the time taken to **execute each statement or the number of primitive operations** performed by the algorithm are taken to be **related by a constant factor.** *It is not going to examine the total execution time of an algorithm.*

```
A function that is a boundary above the algorithms runtime function, when that
algorithm is given the inputs that maximize the algorithm's run time.
```

Classes





(**a**) Big-O Notation (**b**) Big-Omega Notation (**c**) Big-Theta Notation

https://www.mdpi.com/1996-1073/12/1/19/htm

constant C : **The sum of coefficient** that cause a new function from the original function to be **compare between f(n) and g(n)** | C is positive real constant

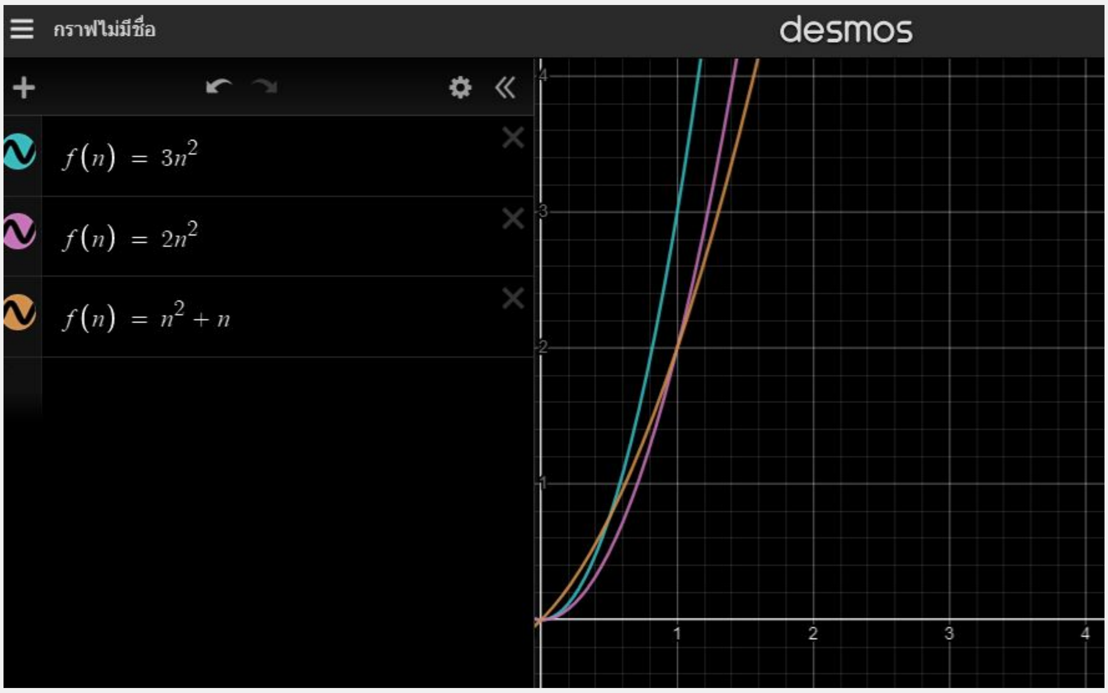
constant K : **Any value** that make the equation comparison of the asymptotic notations f(n) and g(n) | K is positive real constant

Upper bound - Big-O

- f(n) = O(g(n)) ⟺ ∃ ***C*** > 0, ∃ ***K*** > 0 : 0 ≤ **| *f(n)* |** ≤ ***C*** × g(n) | ∀ n ≥ ***K***

f(n) = O(g(n)) the notaion is read f of n is Big-O of g of n, iff there exits ***some positive constant c, k*** such that ***0 ≤ | f(n) | ≤ C × g(n) | ∀ n ≥ K***

- Prove $\lim_{n \to \infty}$ | f(n) / g(n) | < ∞



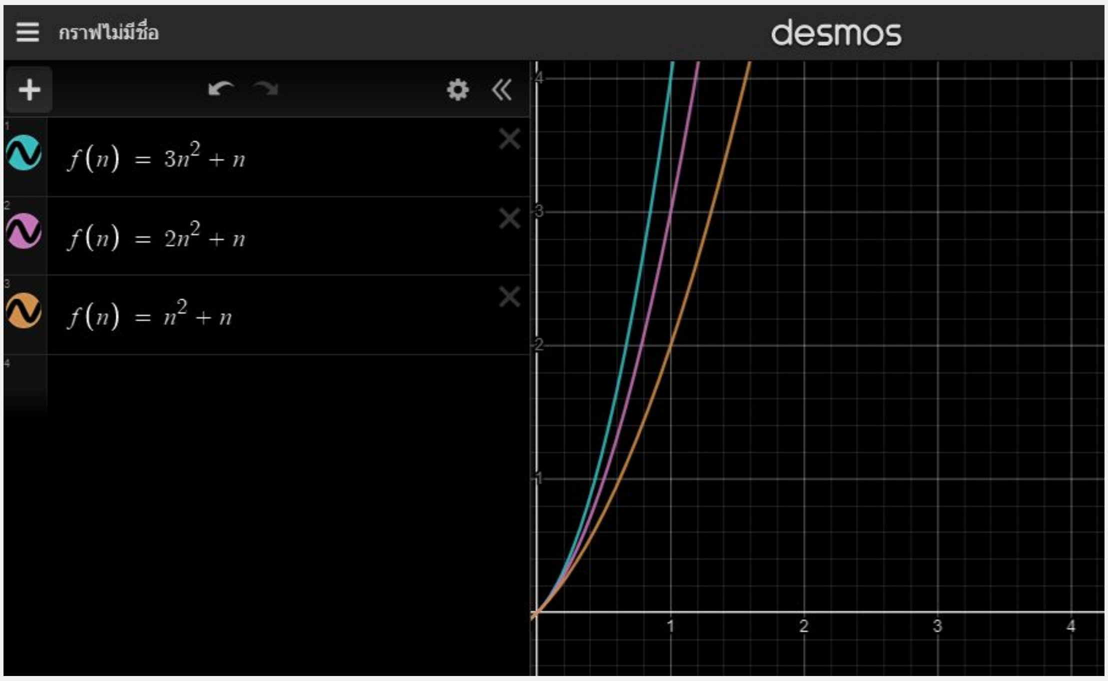
PROVE		**choose k = 1**		
f(n)	=	**n^2 + n**		
g(n)	=	**n^2**		
O(g(n))	=	O(**n^2**)		
cg(n)	=	**n^2 + n^2**	=	**$2n^2$**
c	=	2		
k	=	1	∴	**n ≥ 1**
| f(n) | ≤ cg(n)	=	**| n^2 + n | ≤ $2n^2$**		
∴ f(n)	=	**n^2 + n**	is	**O(n^2)**
Big-O	=	O(n^2)	∴	**n^2 + n ∈ O(n^2)**

Upper bound - Little-o

- f(n) = o(g(n)) ⟺ ∃ ***C*** > 0, ∃ ***K*** > 0 : 0 < **| *f(n)* |** < ***C*** × g(n) | ∀ n ≥ ***K***

f(n) = o(g(n)) the notaion is read f of n is Little-o of g of n, iff there exists positive ***some constants c, k*** such that ***0 < | f(n) | < C × g(n) | ∀ n ≥ K***

- Prove $\lim_{n \to \infty}$ | f(n) / g(n) | = 0



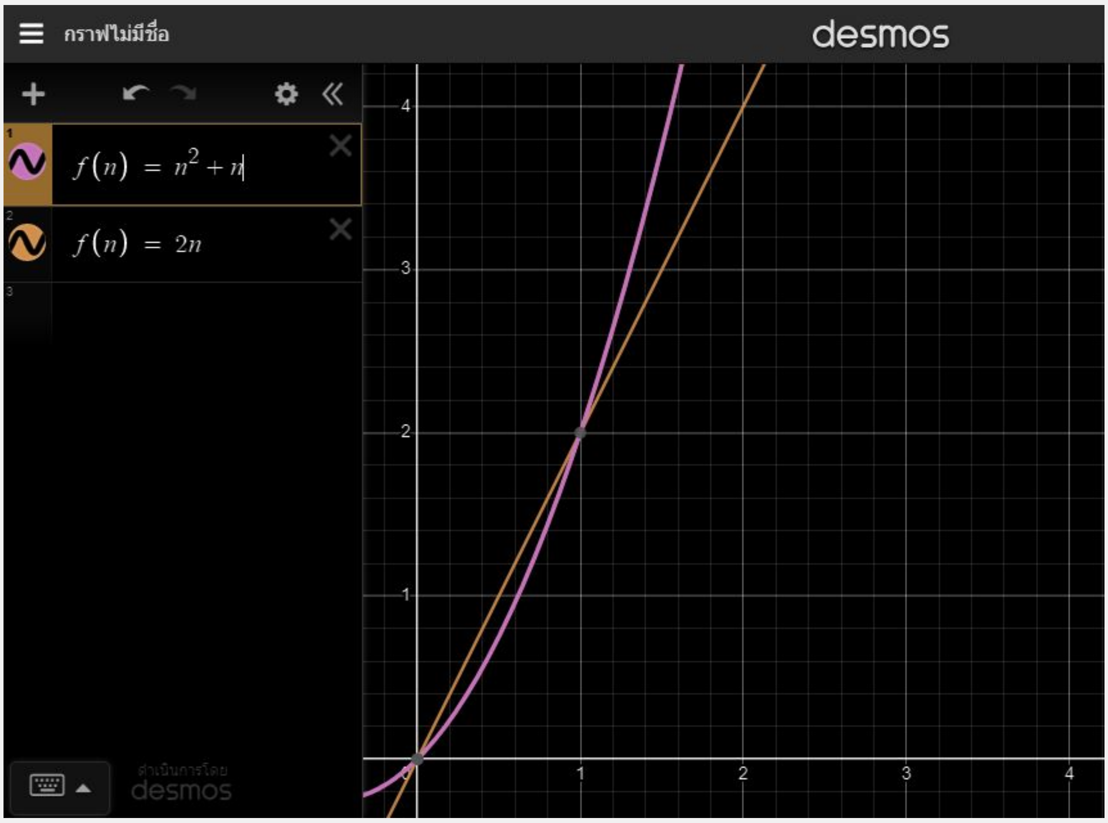
PROVE		**choose k = 1 and c = 3**		
f(n)	=	**n^2 + n**		
g(n)	=	**n^3 + n**		
o(g(n))	=	o(**n^3**)		
c	=	3		
cg(n)	=	3 * n^3 + n	=	**$3n^3$ + n**
k	=	1	∴	**n ≥ 1**
| f(n) | < cg(n)	=	**| n^3 + n | < $3n^3$ + n**		
∴ f(n)	=	**n^2 + n**	is	**o(n^3)**
little-o	=	o(n^3)	∴	**n^2 + n ∈ o(n^3)**

Lower bound - Big-Ω

- f(n) = Ω(g(n)) ⟺ ∃ ***C*** > 0, ∃ ***K*** > 0 : 0 ≤ ***C*** × g(n) ≤ **| *f(n)* |** | ∀ n ≥ ***K***

f(n) = Ω(g(n)) the notaion is read f of n is Big-Omega of g of n, iff there exits ***some positive constant c, k*** such that ***0 ≤ C × g(n) ≤ | f(n) |, ∀ n ≥ K***

- Prove $\lim_{n \to \infty} | f(n) / g(n) | > 0$



PROVE		**choose k = 1**		
f(n)	=	**3n^2**		
g(n)	=	**n^2 + n**		
Ω(g(n))	=	Ω(**n^2**)		
cg(n)	=	**n^2 + n^2**	=	**2n^2**
c	=	2		
k	=	1	∴	**n ≥ 1**
cg(n) ≤ | f(n) |	=	**2n^2 ≤ | 3n^2 |**		
∴ f(n)	=	**3n^2**	is	**Ω(n^2)**
Big-Ω	=	Ω(n^2)	∴	**3n^2 ∈ Ω(n^2)**

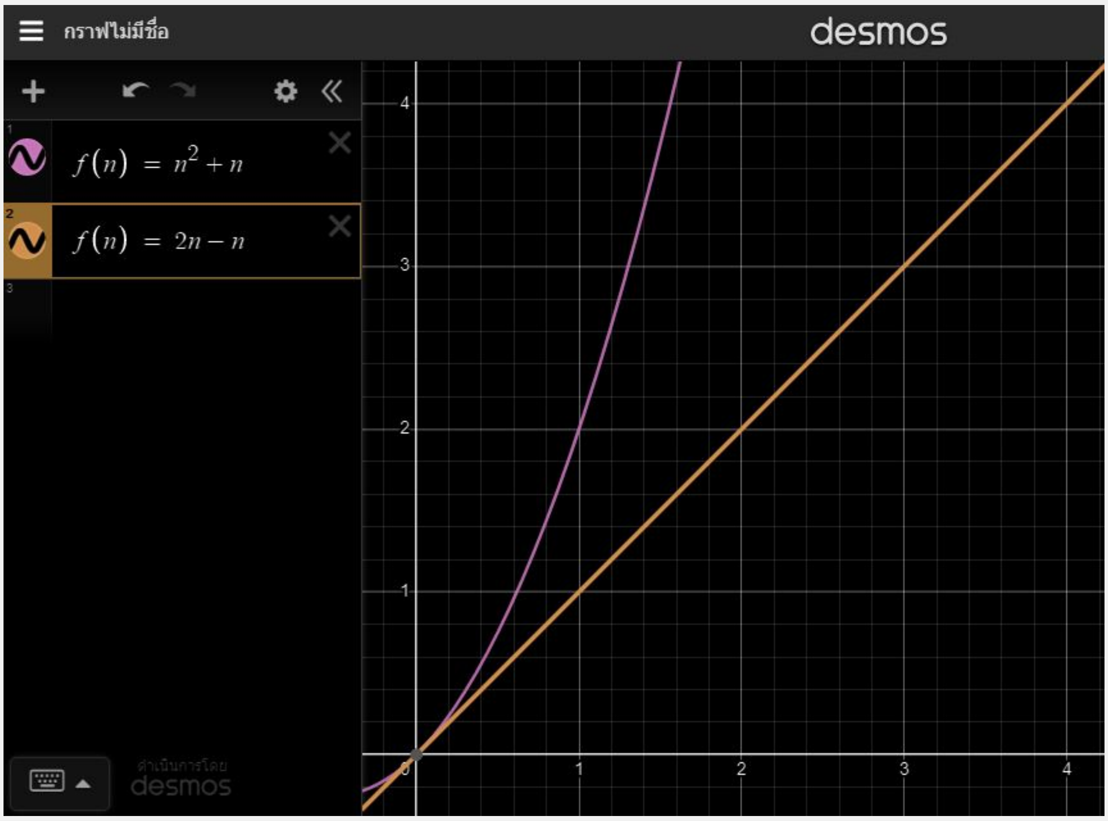
PROVE		**choose k = 1**		
				but $3n^2 \notin O(n^2)$

Lower bound - Little-ω

- $f(n) = \omega(g(n)) \iff \exists\, \boldsymbol{C} > 0, \exists\, \boldsymbol{K} > 0 : 0 < \boldsymbol{C} \times g(n) < \boldsymbol{|\, f(n)\, |}\ |\ \forall\ n \geq \boldsymbol{K}$

f(n) = ω(g(n)) the notaion is read f of n is Little-omega of g of n, iff there exits ***some positive constant c, k*** such that ***0 < C × g(n) < | f(n) |, ∀ n ≥ K***

- Prove $\lim_{n \to \infty} |\, f(n) / g(n)\, | = \infty$



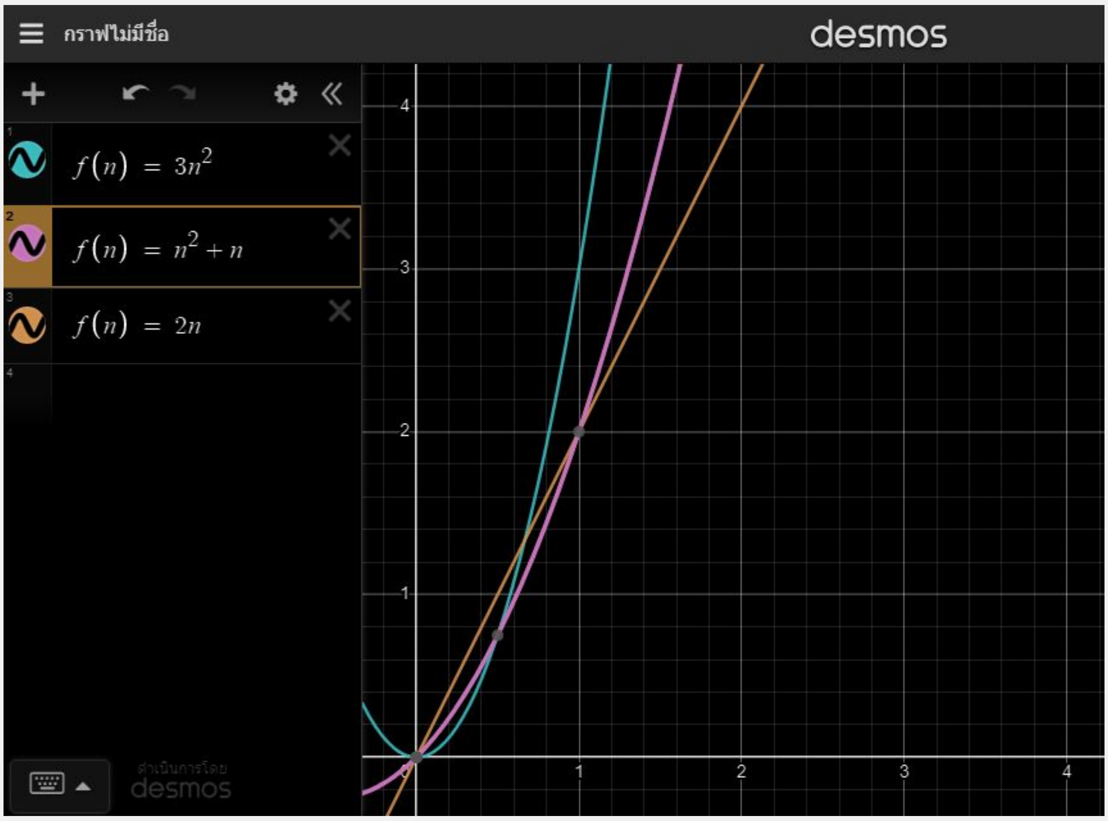
PROVE		choose k = 1		
f(n)	=	**$3n^2 + n$**		
g(n)	=	**$n^2 + n$**		
ω(g(n))	=	ω(**n^2**)		
cg(n)	=	**$n^2 + n^2$**	=	**$2n^2$**
c	=	2		
k	=	1	∴	**$n \geq 1$**
cg(n) < | f(n) |	=	**$2n^2 < \| 3n^2 + n \|$**		
∴ f(n)	=	**$3n^2 + n$**	is	**$\omega(n^2)$**
Little-ω	=	$\omega(n^2)$	∴	**$3n^2 + n \in \omega(n^2)$**
				but $3n^2 + n \notin o(n^2)$

Upper and Lower bound - Big-Θ

- f(n) = Θ(g(n)) $\iff \exists$ ***C_1, C_2*** > 0, $\exists$ ***K*** > 0 : 0 ≤ ***C_1*** × g(n) ≤ **| *f(n)* |** ≤ ***C_2*** × g(n) | ∀ n ≥ ***K***

f(n) = Θ(g(n)) the notaion is read f of n is Big-Theta of g of n, iff there exits
some positive constant c, k such that ***0 ≤ | f(n) | ≤ C_2 × g(n) | ∀ n ≥ K***
some positive constant c, k such that ***0 ≤ C_1 × g(n) ≤ | f(n) |, ∀ n ≥ K***

- Prove $\lim_{n \to \infty}$ | f(n) / g(n) | ∈ R > 0



PROVE	Lower		choose k = 1		Upper
f(n)			**n^2 + n**		
notation	Big-Ω				Big-O
g(n)	**n**				**n^2**
	Ω(n)				O(n^2)
c	2				3
n ≥ k	**n ≥ 1**				**n ≥ 1**
cg(n)	**2n**				**$3n^2$**
c_1g(n) ≤ | f(n) | ≤ c_2g(n)	**2n**	≤	**n^2 + n**	≤	**$3n^2$**