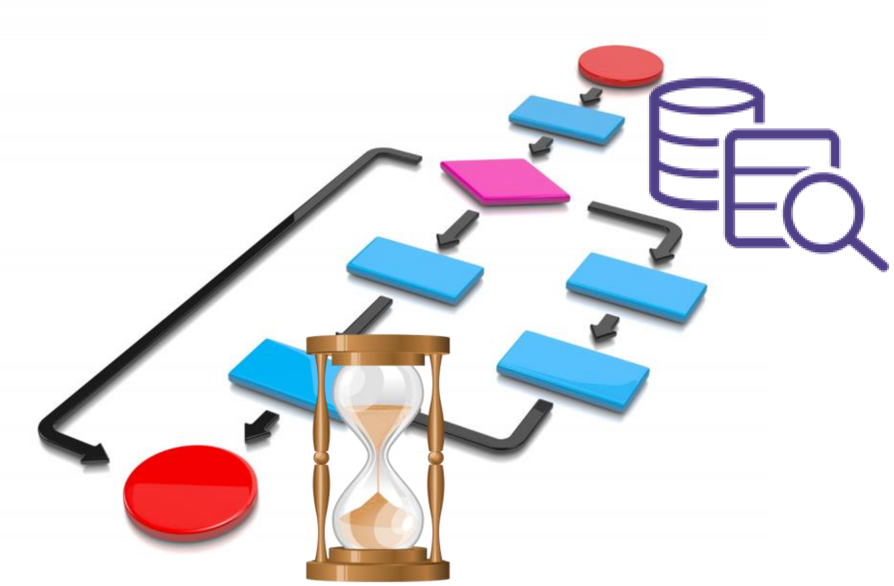
Lecture 2: Big-O Notation

Time Complexity and Space Complexity of an Algorithm



Ms. Togzhan Nurtayeva
Course Code: IT 235/A
Semester 3
Week 4-5
Date: 26.10.2023

Two criteria are used to judge algorithms:

i. Time complexity
ii. Space complexity

Time complexity of an algorithm is the amount of CPU time it needs to run completion.

Space complexity of an algorithm is the amount of memory it needs to run completion.

TIME:
- Operations
- Comparisons
- Loops
- Pointer references
- Function calls to outside

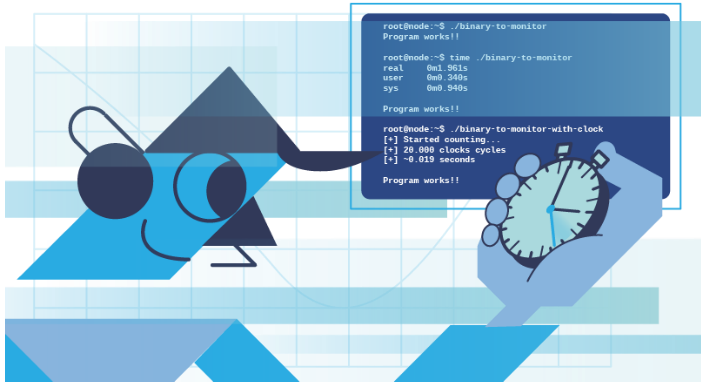
SPACE:
- Variables
- Data structures
- Allocations
- Function call

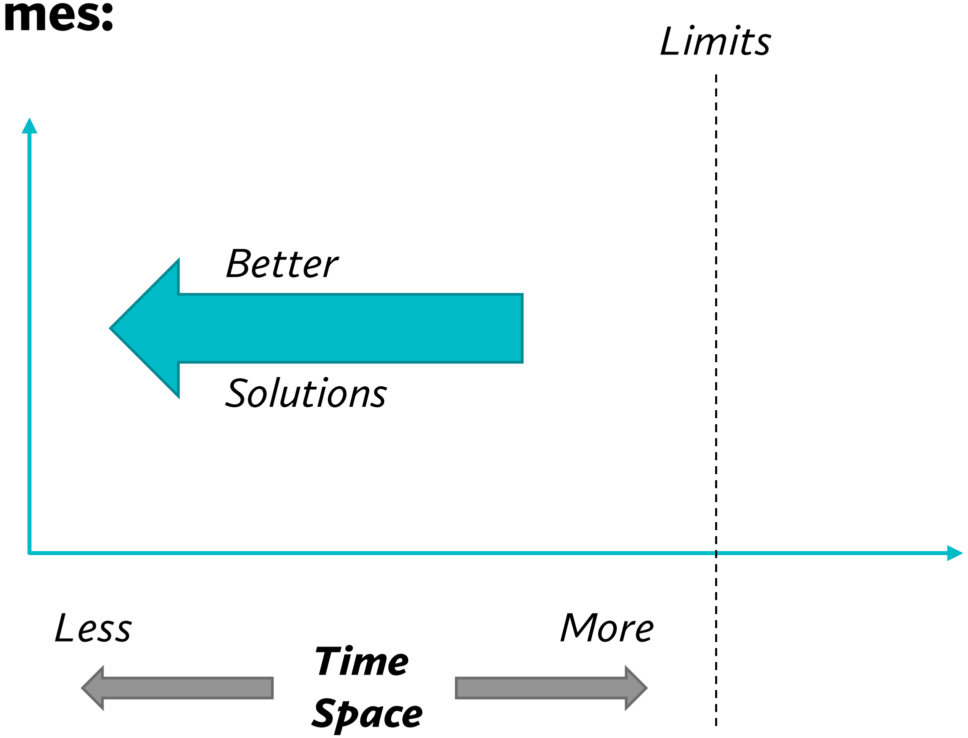
Time complexity of an algorithm is the amount of time (or number of steps) needed by a program to complete its task (to execute a particular algorithm).

The time taken for an algorithm is comprised of two times:

1. Compilation time

2. Run time





Compile Time

- Compilation time is the time taken to compile an algorithm
- While compiling it checks for the syntax ($int \rightarrow itn$) and semantic errors ($int\ 12 \rightarrow int\ 12.5$) in the program and links it with the standard libraries

Run Time

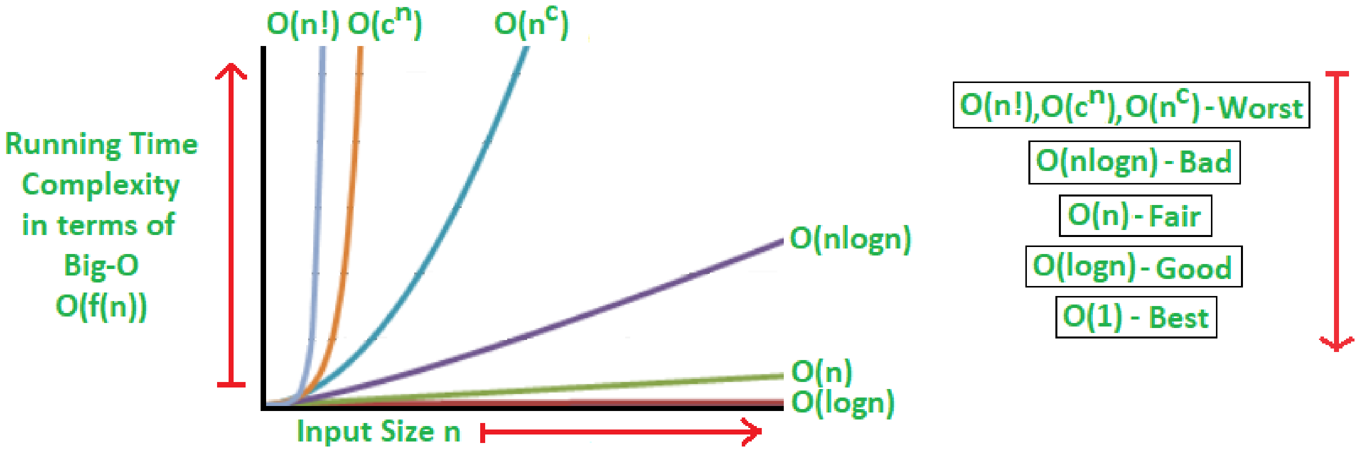
- It is the time to execute the compiled program
- The run time of an algorithm depends on the number of instructions present in the algorithm
- Note that run time is calculated only for executable statements and not for declaration statements

Types of Time Complexity

Time complexity of an algorithm is generally classified into three types:

1. **Worst Case** (Longest Time)
2. **Average Case** (Average Time)
3. **Best Case** (Shorter Time)

- ✓ Big Oh Notation: Upper bound
- ✓ Omega Notation: Lower bound
- ✓ Theta Notation: Tighter bound



Standard Analysis Techniques

- Constant time statements
- Analyzing Loops
- Analyzing Nested Loops
- Analyzing Sequence of Statements
- Analyzing Conditional Statements

Time and Space are dependent on these analysis

Basic Example for Time Complexity

// Input: int A[n], array of n integers
// Output: sum of all numbers in array A

```
int Sum (int A[], int N){

int s = 0;          1

for (int i = 0; i < N; i++)          4
  2                          3
s = s + A[i];
  5                    7
return s;                 6
}
          8
```

1, 2, 8: Once
3, 4, 5, 6, 7: Once per each iteration of for loop, N iteration

Total: $5N + 3$

The *complexity function* of the algorithm is:

$$f(N) = 5N + 3$$

Space Complexity of Algorithm

Space Complexity of a program is the amount of memory consumed by the algorithm until it completes its execution.

The space occupied by the program is generally by the following:

1. A *fixed amount of memory* occupied by the space for the program i.e. data types
2. Code and space occupied by the *variables* used in the program.
3. A *variable amount of memory* occupied by the *component variable* whose *size is dependent on the problem being solved*.
4. This *space increases or decreases* depending on whether *the program uses iterative or recursive procedures*.

$$\boxed{\text{Space Complexity} = \text{Auxiliary Space} + \text{Input Space}}$$

Types of Space Complexity

- **Type 1**: A **fixed part** that is a space required to *store certain data and variables*, that *are independent of the size of the problem*.

For example, simple variables and constant used, program size, etc.

- **Type 2**: A **variable part** is a space required by variables, *whose size depends on the size of the problem.*

For example, dynamic memory allocation, recursion stack space, etc.

Space Complexity S(P) of any algorithm **P is S(P) = C + SP (I)**

Where,

C is the fixed part
S(I) is the variable part

Algorithm: SUM (A, B)
Step 1 – Start
Step 2 – C ← A+B+10
Step 3 - Stop

Other Types of Space

Instruction Space: is the space in memory occupied by the *complied version* of the program. We consider this space as a *constant space for any value of n*. The instruction *space is independent of the size of the problem*.

Data Space: is the space in memory, which used to *hold the variables, data structures, allocated memory and other data elements*. The data space is *related to the size of the problem*.

Environment Space: is the space in memory used on *the run time stack* for each *function call*. This is related to the run time stack and holds the *returning address of the previous function*. *Stored return value and pointer on it*.

Basic Example for Space Complexity

Type	Size
bool, char, unsigned char, signed char, _int8	1 byte
_int16, short, unsigned short, wchar_t, _wchar_t	2 bytes
float, _int32, int, unsigned int, long, unsigned long	4 bytes
double, _int64, long double, long long	8 bytes

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like function calls, jumping statements, etc.

```
{
int z = a + b + c;
return (z);
}
```

(4(4)+4) = 20 bytes

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

4n +12

Difference Between

Space Complexity

Space Complexity is the space (memory) needed for an algorithm to solve the problem. An efficient algorithm take space as small as possible.

Time Complexity

Time Complexity is the time required for an algorithm to complete its process. It allows comparing the algorithm to check which one is the efficient one.

Time complexity of a program is a simple measurement of how fast the time taken by a program grows, if the input increases.

Method 1

```
function isPrime(n) {
 for (let i = 2; i < n; ++i) {
    if (n % i === 0) {
        return false;
    }
 }
 return true;
}
```

$$n - 2$$

Method 2

```
function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); ++i) {
      if (n % i === 0) {
          return false;
      }
  }
  return true;
}
```

$$\sqrt{n} - 2$$

The second method is **faster**. That's why time complexity is **important**. In real life we want software to be fast & smooth.

Space complexity of a program is a simple measurement of how fast the space taken by a program grows, if the input increases.

Method 1

```
function fibonacci(n) {
    const arr = [0, 1];
    for (let i = 2; i <= n; ++i) {
        arr.push(arr[i - 2] + arr[ i - 1]);
    }
    return arr[n - 1];
}
```

$O(n)$

Method 2

```
function fibonacci(n) {
    let x = 0, y = 1, z;
    if (n === 0) {
        return x;
    }
    if (n === 1) {
        return y;
    }
    for (let i = 2; i <= n; ++i) {
        z = x + y;
        x = y;
        y = z;
    }
    return z;
}
```

$O(1)$

The second method is **better**. There is no point in using more space to solve a problem if, we can do the same with **lesser space complexity**.

Calculating Time complexity of Algorithms

Calculations in different Cases

1. Loop

for ($i = 1$ to n){ // n
x = y + z; // *constant time*
}

$O(n)$

2. Nested Loop

for ($i = 1$ to n){ // n
for ($j = 1$ to n){ // n
x = y + z; // *constant time*
}
}

$O(n^2)$

constant time can be neglected

Calculations in different Cases

3. Sequential Statements

i) a = a+ b; $// \ constant \ time = c_1$

ii) for ($i = 1$ to n){ $// \ n$
x = y + z; $// \ constant \ time = c_2$
}

iii) for ($j = 1$ to n){ $// \ n$
c = d + e; $// \ constant \ time = c_3$
}

$\boxed{O(n)}$ $= c_1 + c_2 n + c_3 n = n$

4. If-else Statements

if (condition){
$// \ n$
}
else
{
$// \ n^2$
}

$\boxed{O(n^2)}$

5. Loops running constant times

for $(i = 1 \text{ to } c)$
{
  x = y + z;
}

int $i = 1$;
while $(i \leq c)$
{
  x = y + z;
}

$O(1)$

6. Loops running n times and incrementing/decrementing by constant

for $(i = 1\,; i \leq n; i = i + c)$
{
  x = y + z;
}

int $i = 1$;
while $(i \leq n)$
{
  i = i + c;
}

$O(n)$

```
for (int i = 1; i <= c*n; i = i + 1)
{
    Some O(1) expressions
}

// while loop version
int i = 1;
while (i <= c * n)
{
    Some O(1) expressions
    i = i + 1;
}
```

$O(n)$

7. Loops running n times and incrementing/decrementing by constant factor

for $(i = 1; i \leq n; i = i * c)$
{
  x = y + z;
}

int $i = 1$;
while $(i \leq n)$
{
  i = i / c;
}

$O(logn)$

8. Loops running n times and incrementing by some constant power

for $(i = 2\, ; i \leq n; i = pow\ (i, c))$
{
  x = y + z;
}

int $i = 2$;
while $(i \leq n)$
{
  i = pow(i , c);
}

$\boxed{O(log(logn))}$

$1 \rightarrow i = 2$
$2 \rightarrow i = 2^c$
$3 \rightarrow i = 2^{c^2}$
...
The loop will end when: $n = 2^{c^i}$

$\log 2(n) = \log 2\,(2^{c^i})$
$logn = c^i$

$\log c(logn) = logc(c^i)$
$i = logc(logn)$

EXAMPLES

More Examples

Algo1 ()
{
int i;
for (i $=$ 1 to n)
print ("Hello World");
}

$\boxed{O(n)}$

Algo2 ()
{
int i;
for (i $=$ 1 to n)
  for (j $=$ 1 to n)
print ("Hello World");
}

//nested loop

$\boxed{O(n^2)}$

Algo3 ()

```
{
int i;
for (i = 1; i < n; i = i * 2)
print ("Hello World");
}
```

$i \rightarrow 1, 2, 4, 8, 16, 32, \ldots n$

$2^0, 2^1, 2^2, 2^3, 2^4, 2^5 \ldots\ 2^k$

$n = 2^k$

$2^k = n$

$k = \log_2 n$

$O(\log_2 n)$

Algo4 ()

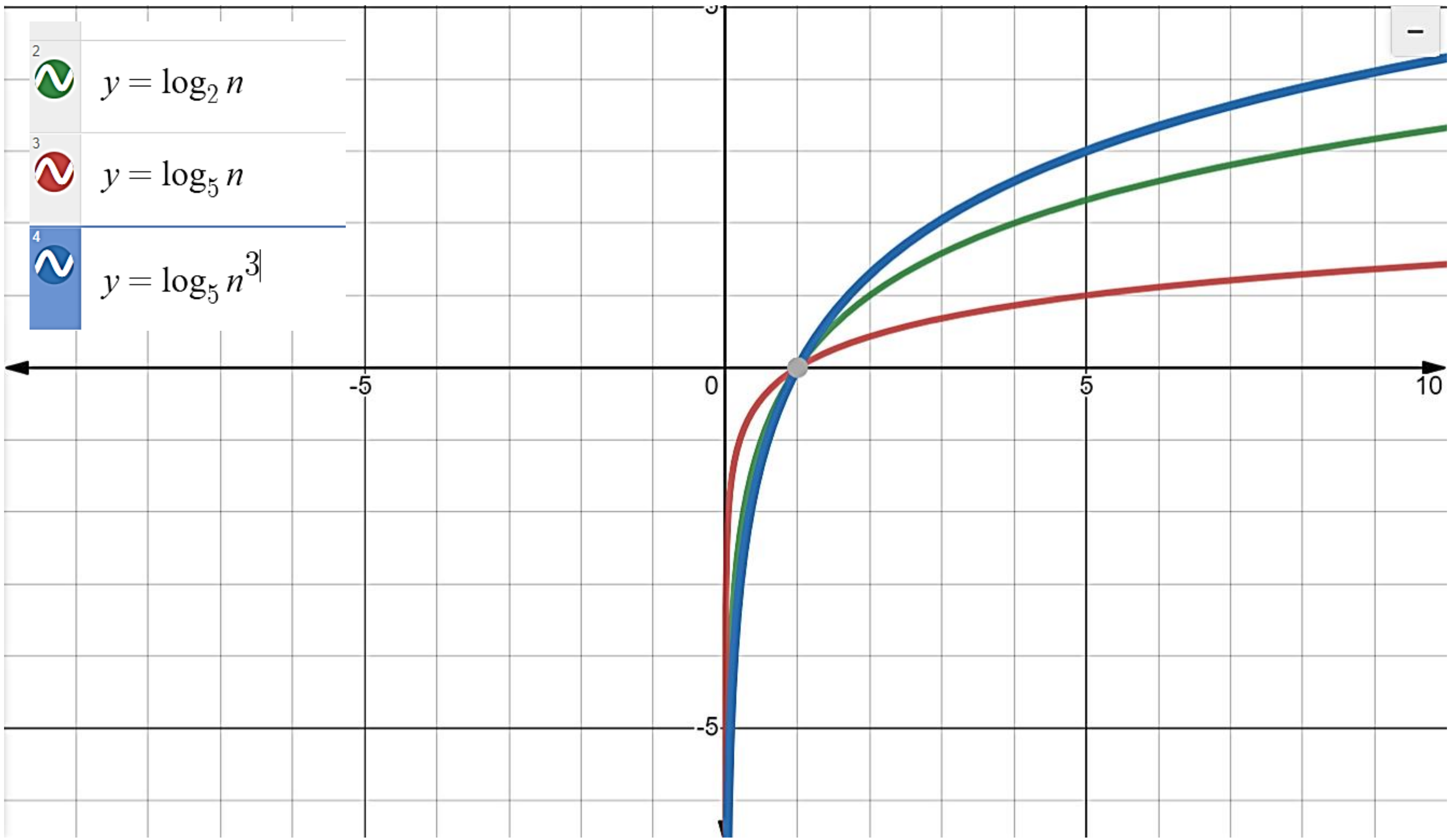
```
{
int i;
for (i = 1; i < n; i = i/5)
print ("Hello World");
}
```

$O(\log_5 n)$

Algo5 ()

```
{
int i;
for (i = 1; i < n^3; i = i * 5)
print ("Hello World");
}
```

$O(\log_5 n^3)$



2
$y = \log_2 n$
3
$y = \log_5 n$
4
$y = \log_5 n^3$
-5
0
5
10
-5

Algo6 ()

```
{
int i;
for (i = 1; i^2 <= n; i++)
print ("Hello World");
}
```

$$i^2 <= n$$
$$\sqrt{i^2} <= \sqrt{n}$$
$$i <= \sqrt{n}$$

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

Algo7 ()

```
{
int i = 1, k = 1;
while (k <= n)
{
    i++;
    k = k + i;
print ("Hello World");
}
}
```

// all operations inside any loop considered

k	1	3	6	10	15	...	n
i	1	2	3	4	5	...	z

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

$$z^2 + z = 2n$$

$$z = \sqrt{2n} = \sqrt{2} \cdot \sqrt{n}$$

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

Algo8 ()

```
{
int i, j, k;
for (i = n/2; i <= n; i++)          // n/2
  for (j = 1; j <= n/2; j++)        // n/2
    for (k = 1; k <= n; k = k * 2)  // log2 n
print ("Hello World");
}
```

$$\frac{n}{2} \cdot \frac{n}{2} \cdot \log_2 n = \frac{n^2 \log_2 n}{4}$$

$$\boxed{O(n^2 \log_2 n)}$$

Algo9 ()

```
{
int i = n;
while (i > 1)
{
print ("Hello World");
 i = i/2; // log2 n
}
}
```

$$\boxed{O(\log_2 n)}$$

Algo10 ()

```
{
int i, j, k;
for (i = n/2; i < n; i++)              // n/2
  for (j = 1; j <= n; j = 2 * j)       // log2 n
    for (k = 1; k <= n; k = k * 2)     // log2 n
print ("Hello World");
}
```

$$\frac{n}{2} \cdot \log_2 n \cdot \log_2 n = \frac{n\,(\log_2 n)^2}{2}$$

$$\boxed{O(n\,(\log_2 n)^2)}$$

Independent Loop

Algo11 ()

for ($i = 1$ to n){
 for (k = 1 to m)
print (“Hello World”);
}

$\boxed{O(nm)}$

Dependent Loop

Algo12 ()

for ($i = 1; i \leq n; i++$){
 for ($k = 1;\ k <= i;\ k = k + 1$)
print (“Hello World”);
}

i	$1\ time$	$2\ times$	$3\ times$	…	$n\ times$
k	$1\ time$	$1, 2\ times$	$1, 2{,}3\ times$	…	$1, 2, 3, \ldots, n\ times$

$$TC = 1 + 2 + 3 \ldots + n = \left(\frac{n(n+1)}{2}\right) = \left(\frac{n^2 + n}{2}\right)$$

$$\boxed{O(n^2)}$$

$$\sum_{k=1}^{n} \frac{1}{k} \approx \int_{1}^{n} \frac{dx}{x} = logn$$

Algo13 ()

for $(i = 1; i \leq n; i++)\{$
  for $(j = 1;\ j <= n; j = j + i)$
$x = x + 1;$
$\}$

i	1	2	3	...	n
j	n	$n/2$	$n/3$	...	n/n

$$\boxed{O(nlogn)}$$

$$TC = n\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) = nlogn$$

$$1 + (1+2) + (1+2+3) + \cdots + (1+2+3+\cdots+n) = \frac{n(n+1)(n+2)}{6}$$

Algo14 ()

$x = 0;$
for $(i = 1; i \leq n; i++)\{$
for $(j = 1; j \leq i; j = j++)$
for $(\mathrm{k} = 1; k \leq j; k = k++)$
$x = x + 1;$
$\}$

i	1	2	3	...	n
j	1	1; 2	1; 2; 3	...	$1, 2, 3, \ldots, n$
k	1	1; 1, 2	1; 1, 2; 1, 2, 3	...	$1; 1, 2; \ldots 1,2,3 \ldots n$

$O(n^3)$

Now it's time

to practice!

Practice

```
int fun(int n)
{
    int count = 0;
    for (int i = n; i > 0; i /= 2)
        for (int j = 0; j < i; j++)
            count += 1;
    return count;
}
```

$$\boxed{T(n) = n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = O(n)}$$

```
int a = 0, b = 0;
for (i = 0; i < N; i++) {
    a = a + rand();
}
for (j = 0; j < M; j++) {
    b = b + rand();
}
```

$$\boxed{O(N + M)}$$

Practice

```
for (int i = n; i > 0; i = i / 2)
{
   for (int j = 1; j < n; j = j * 2)
{
      for (int k = 0; k < n; k = k + 2)
{
//some logic with complexity X
}
}
}
```

$\boxed{O(n\,(\log_2 n)^2)}$

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
   for (j = 2; j <= n; j = j * 2)
{
      k = k + n / 2;
   }
}
```

$\boxed{O(nlogn)}$

Practice

```
for(int i=0;i<n;i++){
i*=k;
}
```

$\boxed{O(\log_k n)}$

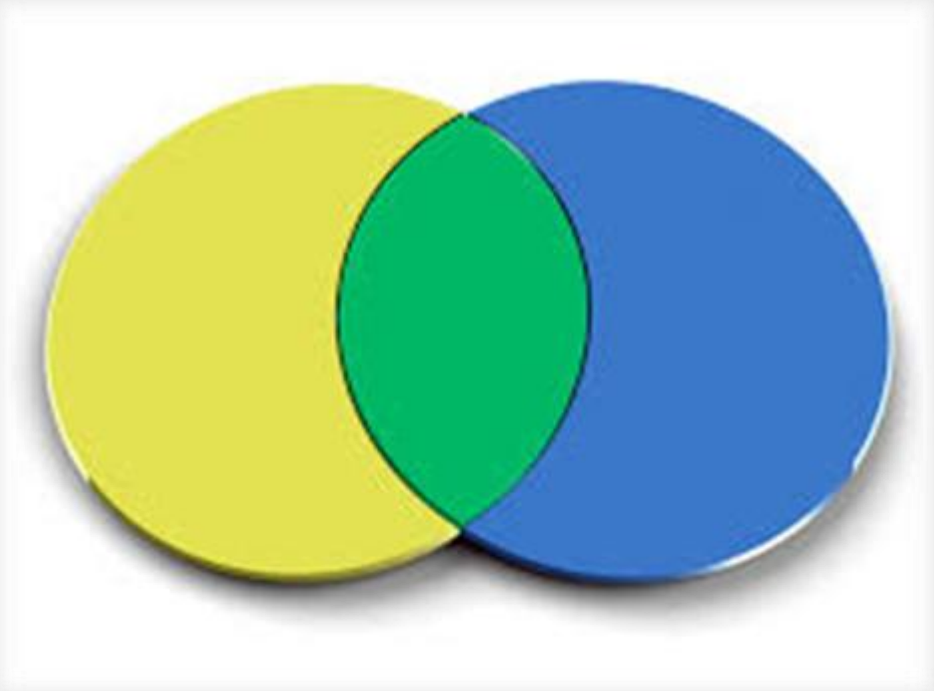
```
int a = 0, i = N;
while (i > 0) {
    a += i;
    i /= 2;
}
```

$\boxed{O(\log n)}$

```
int value = 0;
for(int i=0;i<n;i++)
    for(int j=0;j<i;j++)
      value += 1;
```

$\boxed{O(n^2)}$

Calculating Space complexity of Algorithms



Space Complexity = Auxiliary Space + Input Space

y axis: size in bytes
x axis: N value

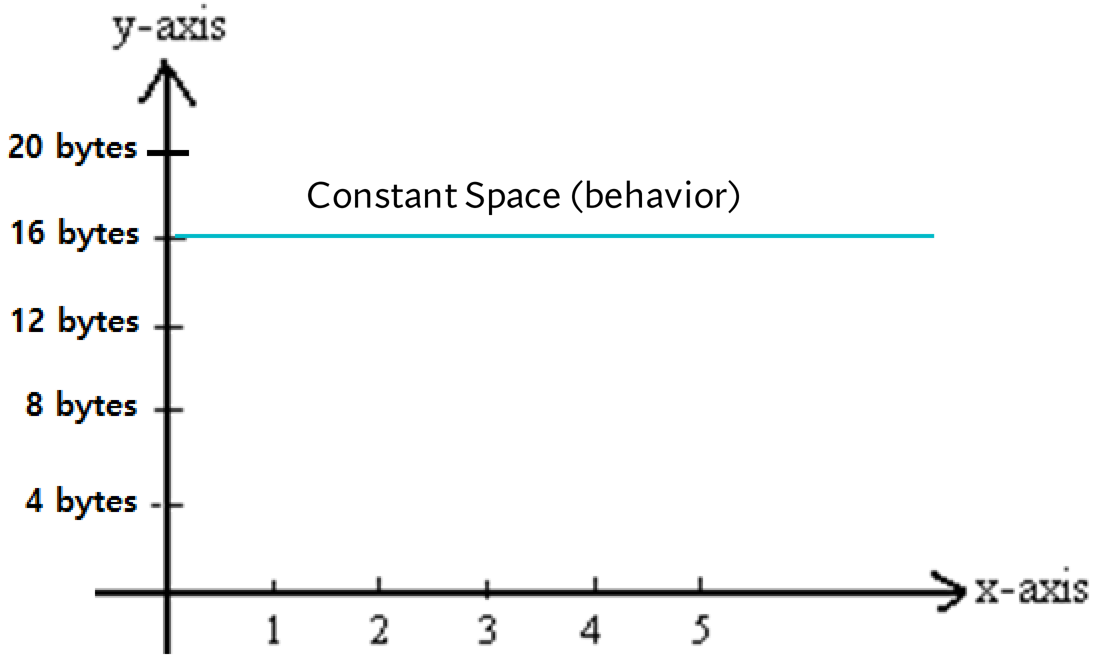
Algo1 () – Addition of two numbers

```
function add(n1, n2)
{
sum = n1 + n2
return sum
}
```

n1 – 4 bytes
n2 – 4 bytes
sum – 4 bytes
Aux (function call, return) – 4 bytes

$O(1)$

Total (estimated): 16 bytes = C



Space Complexity = Auxiliary Space + Input Space

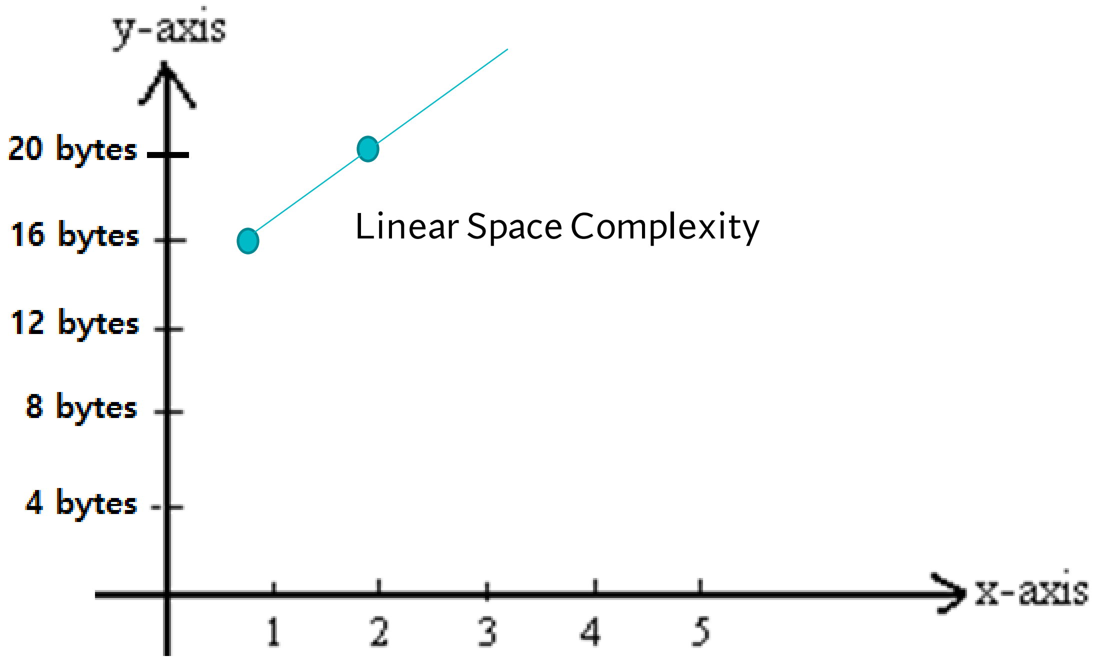
Algo2 () – Sum of all elements in array

```
function sumOfNumbers(arr[], N)
{
sum = 0
for (i = 0 to N)
{
sum = sum + arr[i]
}
print (sum)
}
```

$O(n)$

arr – N * 4 bytes
sum – 4 bytes
i – 4 bytes
Aux (initializing for loop, function call, return) – 4 bytes

Total (estimated): 4N + 12 bytes



Space Complexity = Auxiliary Space + Input Space

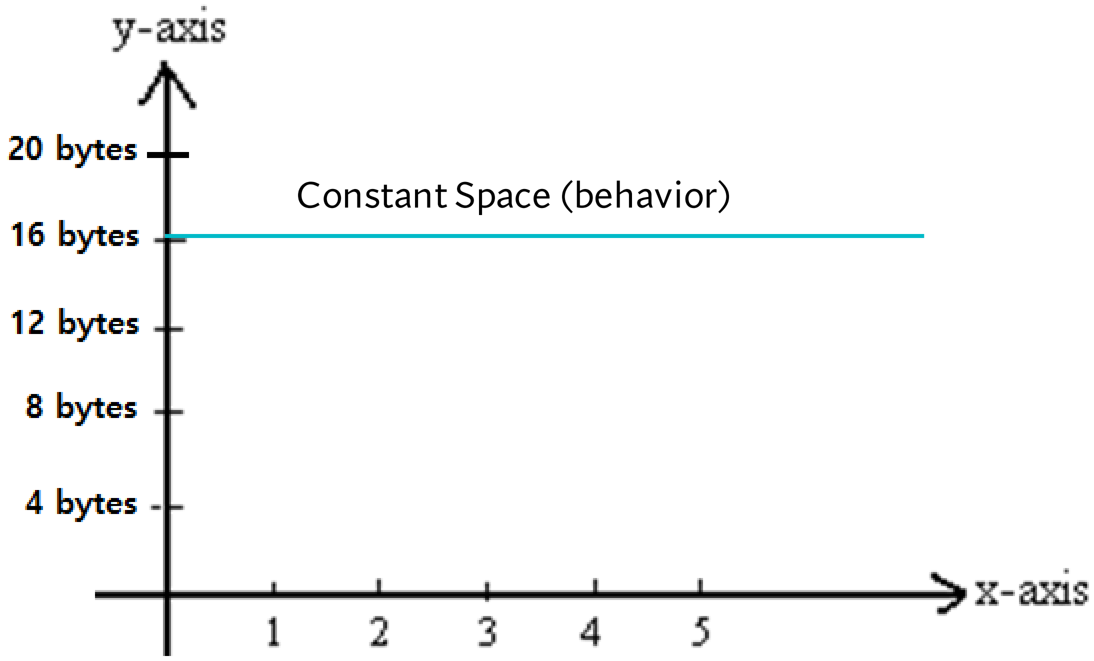
Algo3 () – Factorial of a number (iterative)

```
int fact = 1;
for (int i = 1; i <= n; i++)
{
fact *= i;
}
return fact;
```

fact – 4 bytes
n – 4 bytes
i – 4 bytes
Aux (initializing for loop, function call, return) – 4 bytes

Total (estimated): 16 bytes

$O(1)$



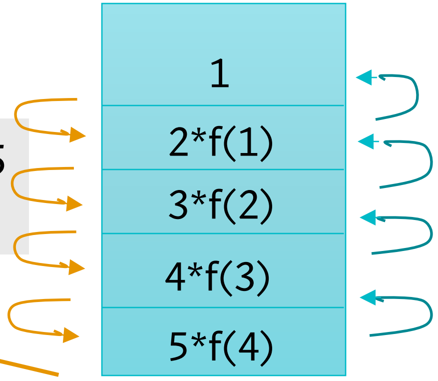
Space Complexity = Auxiliary Space + Input Space

Algo4 () – Factorial of a number (recursive)

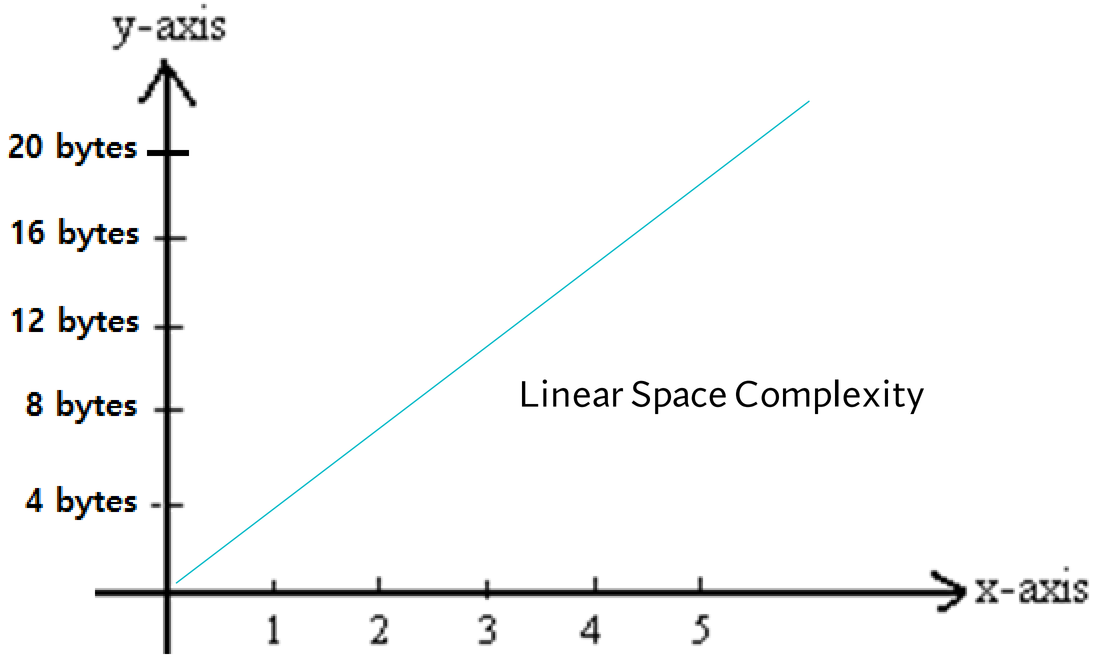
```
factorial(n){
if (n <= 1) {
return 1;
} else {
return (n*factorial(n-1));
}
}
```

$O(n)$

assume n = 5
$fn(5)$



Call Stack



n – 4 bytes
Aux (function call) – 5 * 4 bytes

Total (estimated): 4 bytes + 4*n bytes

Space – Time Tradeoff and Efficiency

All efforts made by analyzing time and space complexity lead to the algorithm's efficiency.

But, when we can say that an algorithm is efficient? The answer seems to be obvious: it should be fast, and it should take the least amount of memory possible.

Unfortunately, in algorithmics, space and time are like two separate poles. **Increasing speed will most often lead to increased memory consumption and vice-versa**.

On the one side, we have merge sort, which is extremely fast but requires a lot of memory. On the other side, we have bubble sort, a slow algorithm but one that occupies minimal space. There are also some balanced ones like in-place heap sort. Its speed and space usage are not the best, but they're acceptable.

Maximizing both the algorithm's space and time complexity is impossible. We should adjust those parameters according to our requirements and environment.

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	Θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)

5 Basic Sequences and Their Sums

Adding positive integers

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

Summing up the squares

$$S = \frac{n(n+1)(2n+1)}{6}$$

Finding the sum of the cubes

$$S = \frac{n^2(n+1)^2}{4}$$

Summing odd numbers

$$S = n^2$$

Adding up even numbers

$$S = n(n+1)$$

PRACTICE on your OWN

What is the time and space complexity of the following codes:

a)

```
let a = 0, b = 0;

for (let i = 0; i < n; ++i) {
    a = a + i;
}
for (let j = 0; j < m; ++j) {
    b = b + j;
}
```

b)

```
let a = 0, b = 0;
for (let i = 0; i < n; ++i) {
    for (let j = 0; j < n; ++j) {
        a = a + j;
    }
}
for (let k = 0; k < n; ++k) {
    b = b + k;
}
```

c)

```
let a = 0;
for (let i = 0; i < n; ++i) {
    for (let j = n; j > i; --j) {
        a = a + i + j;
    }
}
```

What is the time and space complexity of the following codes:

a)

```
let a = 0, b = 0;

for (let i = 0; i < n; ++i) {
    a = a + i;
}
for (let j = 0; j < m; ++j) {
    b = b + j;
}
```

Time Complexity: O(n + m)
Space Complexity: O(1)

b)

```
let a = 0, b = 0;
for (let i = 0; i < n; ++i) {
    for (let j = 0; j < n; ++j) {
        a = a + j;
    }
}
for (let k = 0; k < n; ++k) {
    b = b + k;
}
```

Time Complexity: $O(n^2)$
Space Complexity: O(1)

c)

```
let a = 0;
for (let i = 0; i < n; ++i) {
    for (let j = n; j > i; --j) {
        a = a + i + j;
    }
}
```

Time Complexity: $O(n^2)$
Space Complexity: O(1)

What is the time complexity of the following codes:

a)

```
for (let i = n; i > 0; i = parseInt(i / 2)) {
    console.log(i);
}
```

b)

```
for (let i = 1; i < n; i = i * 2) {
    console.log(i);
}
```

c)

```
for (let i = 0; i < n; ++i) {
    for (let j = 1; j < n; j = j * 2) {
        console.log(j);
    }
}
```

What is the time complexity of the following codes:

a)

```
for (let i = n; i > 0; i = parseInt(i / 2)) {
    console.log(i);
}
```

Time Complexity: O(log n)

b)

```
for (let i = 1; i < n; i = i * 2) {
    console.log(i);
}
```

Time Complexity: O(log n)

c)

```
for (let i = 0; i < n; ++i) {
    for (let j = 1; j < n; j = j * 2) {
        console.log(j);
    }
}
```

Time Complexity: O(nlog n)

What is the time complexity of the following code:

```
// Fibonacci of nth element
function fibonacci (n) {
    if (n <= 1) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

What is the time complexity of the following code:

```
// Fibonacci of nth element
function fibonacci (n) {
    if (n <= 1) {
        return 1;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Time Complexity: O(2^n)

What is the time complexity of the following code:

```
// search an element in an array
// list is already sorted
function search (list, item, start, end) {
    if (start > end) {
        return false;
    }
    const mid = Math.floor((start + end) / 2);
    if (list[mid] < item) {
        return search(list, item, mid + 1, end);
    }
    if (list[mid] > item) {
        return search(list, item, start, mid - 1);
    }
    return true;
}
```

What is the time complexity of the following code:

Recursive Method:

```
// search an element in an array
// list is already sorted
function search (list, item, start, end) {
    if (start > end) {
        return false;
    }
    const mid = Math.floor((start + end) / 2);
    if (list[mid] < item) {
        return search(list, item, mid + 1, end);
    }
    if (list[mid] > item) {
        return search(list, item, start, mid - 1);
    }
    return true;
}
```

Time Complexity: O(log n)

Auxiliary Space: O(log n)

The **iterative** implementation of Binary Search:

```
int binarySearch(int[] A, int x)
{
    int low = 0, high = A.length - 1;
    while (low <= high)
    {
        int mid = (low + high) / 2;
        if (x == A[mid]) {
            return mid;
        }
        else if (x < A[mid]) {
            high = mid - 1;
        }
        else {
            low = mid + 1;
        }
    }
    return -1;
}
```

Time Complexity: O (log n)
Auxiliary Space: O (1)

Analysis of input size at each iteration of Binary Search:

At Iteration 1:

Length of array = n

At Iteration 2:

Length of array = $\frac{n}{2}$

At Iteration 3:

Length of array = $\frac{\frac{n}{2}}{2} = \frac{n}{2^2}$

Therefore, after Iteration k:

Length of array = $\frac{n}{2^k}$

Also, we know that after k iterations, the length of the array becomes 1 Therefore, the Length of the array:

$$\frac{n}{2^k} = 1$$

$$=> n = 2^k$$

Applying *log* function on both sides:

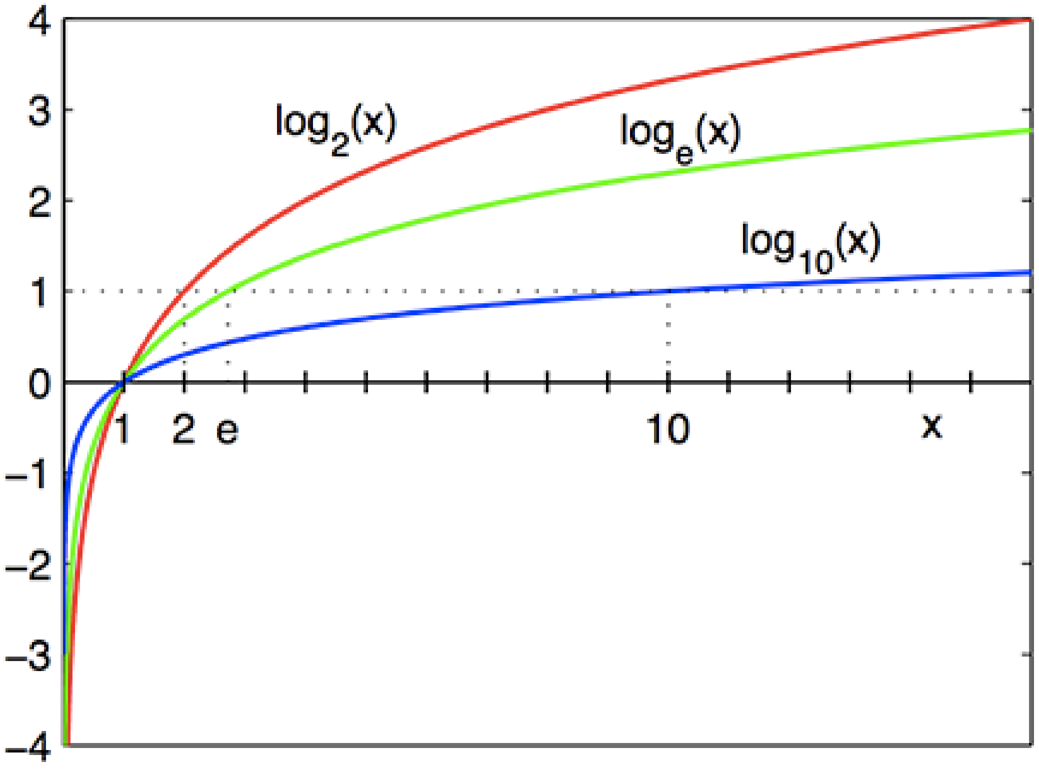
$=> \log_2 n = \log_2 2^k$

$=> \log_2 n = \mathrm{k} * \log_2 2$

As $(\log_a a = 1)$ Therefore, $k = \log_2 n$

Logarithmic Form **Exponential Form**

$$\log_{\underbrace{b}_{\text{Base}}} M = \overbrace{N}^{\text{Exponent}} \quad \Leftrightarrow \quad \underbrace{b}_{\text{Base}}^{\overbrace{N}^{\text{Exponent}}} = M$$



$$\log_b\left(M^k\right) = k \cdot \log_b M$$

$$\log_b\left(b^k\right) = k$$

Self-study links

- https://medium.com/@manishsakariya/time-complexity-examples-6a4877a1b923

- https://www.youtube.com/watch?v=AWHi1-Xmd-Y

- https://www.youtube.com/watch?v=yOb0BL-84h8&t=1107s

- https://www.enjoyalgorithms.com/blog/time-complexity-analysis-of-loop-in-programming