

The background is a complex, abstract pattern. It features a dense grid of small, semi-transparent circles in various colors including green, blue, purple, brown, orange, and red. Overlaid on this grid are faint, light-gray geometric shapes, including squares and circles, some of which contain internal patterns like concentric circles or radial lines. The overall effect is a textured, digital aesthetic.

Algorithms

Complexity analysis

How to analyze an algorithm?

- The execution time of an algorithm with a particular input is the number of primitive operations or "**steps**" executed. These define a notion that is as machine-independent as possible.
- Thus, a constant amount of time is required to execute each line of pseudocode. One line may take a different amount of time than another, but it must be assumed that each execution of the i -th line takes C_i , where C_i is a constant value.

How to analyze an algorithm?

- When an algorithm contains an iterative control structure such as a **while** or **For loop**, the execution times can be expressed as a sum of the time spent in each execution of the '**body**' of the loop.



- Given a sequence $\mathbf{a_1, a_2, ..., a_n}$ of numbers, where \mathbf{n} is a non-negative integer, we can write the finite sum:

$$\sum_{k=1}^n a_k = a_1 + a_2 + \cdots + a_n$$

Note. if $\mathbf{n=0}$, the value of the sum is defined as zero.

The value of a series is always "well defined", and you can add the terms in any order.

- We also can write an infinite sum like:

$$\sum_{k=1}^{\infty} a_k = a_1 + a_2 + \cdots$$

Which can be interpreted as:

$$\lim_{n \rightarrow \infty} \left(\sum_{k=1}^n a_k \right)$$

If the **limit does not exist**, the series **diverges**; otherwise, it **converges**.

- For any real number **c** and any finite sequence **a₁, a₂, ..., a_n** and **b₁, b₂, ..., b_n**

$$\mathbf{1)} \quad \sum_{k=1}^n (c a_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$$

- There are additional properties that allow us to write the summations in different ways, as we see fit:

$$\mathbf{2)} \quad \sum_{k=1}^n k \rightarrow 1 + 2 + \cdots + n \quad \neq \quad \sum_{k=1}^n 1 \rightarrow n$$

$$\mathbf{3)} \quad \sum_{k=0}^n k = 0 + 1 + 2 + \left(\sum_{k=3}^n k \right) \quad \mathbf{and} \quad \sum_{k=1}^8 k = \left(\sum_{k=1}^{10} k \right) - 9 - 10$$

$$\mathbf{4)} \quad \sum_{k=l}^n k = \sum_{k=0}^n k - \sum_{k=1}^l k \quad (0 \leq l < n)$$

$$\mathbf{5)} \quad \sum_{i=l}^u a_i = \sum_{i=l}^m a_i + \sum_{i=m+1}^u a_i \quad (l \leq m < u)$$

Series & Summation

- Arithmetic series

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

- Sum of squares & cubic

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^n k^3 = \frac{n^2(n+1)^2}{4}$$

- Geometric series

For real values $x \neq 1$

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}$$

- Harmonic series

For positive integer values n

$$\sum_{k=1}^n \frac{1}{k} = \ln n + O(1)$$

Products

We can write a finite product like

$$\prod_{k=1}^n a_k = a_1 \times a_2 \times a_3 \times \cdots \times a_n$$

If **n=0**, the value of the product is defined as **1**. We can convert a formula with a product to a summation

$$\log \left(\prod_{k=1}^n a_k \right) = \sum_{k=1}^n \log(a_k) = \log(a_1) + \log(a_2) + \cdots + \log(a_n)$$

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the **running time of an algorithm** when the input tends towards a particular value or a limiting value.

There are mainly three asymptotic notations:

- **Big-O notation**
- **Omega notation**
- **Theta notation**



Big-O notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the **worst-case** complexity of an algorithm.

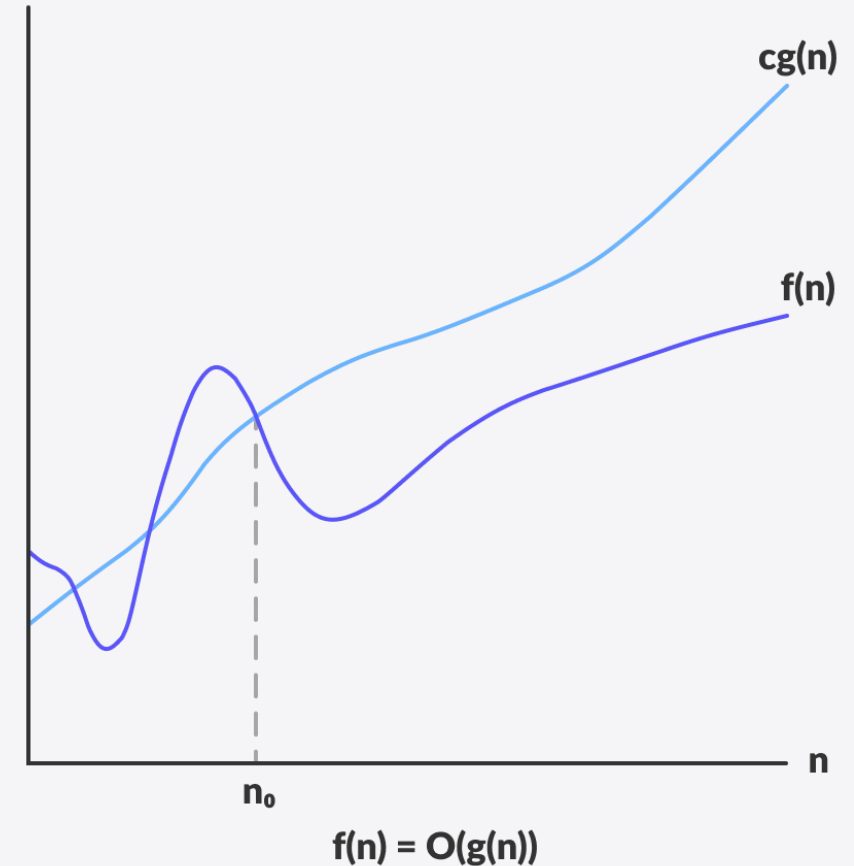
$$f(n) = O(g(n))$$

The above expression can be described as a function **f(n)**, function that describe the execution time of the algorithm, has the upper bound **O(g(n))** if there exists a positive constant **c** such that

$$f(n) \leq c g(n)$$

for sufficiently large **n** ($n \geq n_0$, where **n₀** is constant).

It means that for any value of $n \geq n_0$, the running time of an algorithm does not cross the time provided by **c g(n)**.





Omega Notation (Ω -notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the **best-case** complexity of an algorithm.

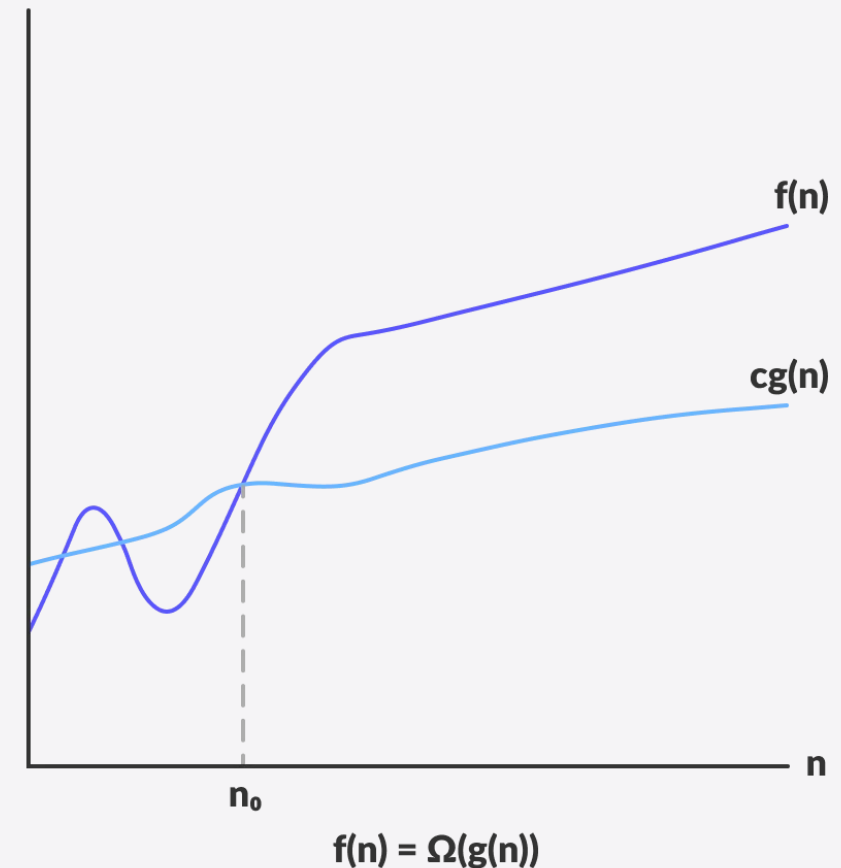
$$f(n) = \Omega(g(n))$$

The above expression can be described as a function **$f(n)$** , function that describe the execution time of the algorithm, has the lower bound **$\Omega(g(n))$** if there exists a positive constant **c** such that

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

for sufficiently large **n** ($n \geq n_0$, where **n_0** is constant).

It means that for any value of **$n \geq n_0$** , the minimum time required by the algorithm is given by **$c g(n)$** .





Theta Notation (Θ -notation)

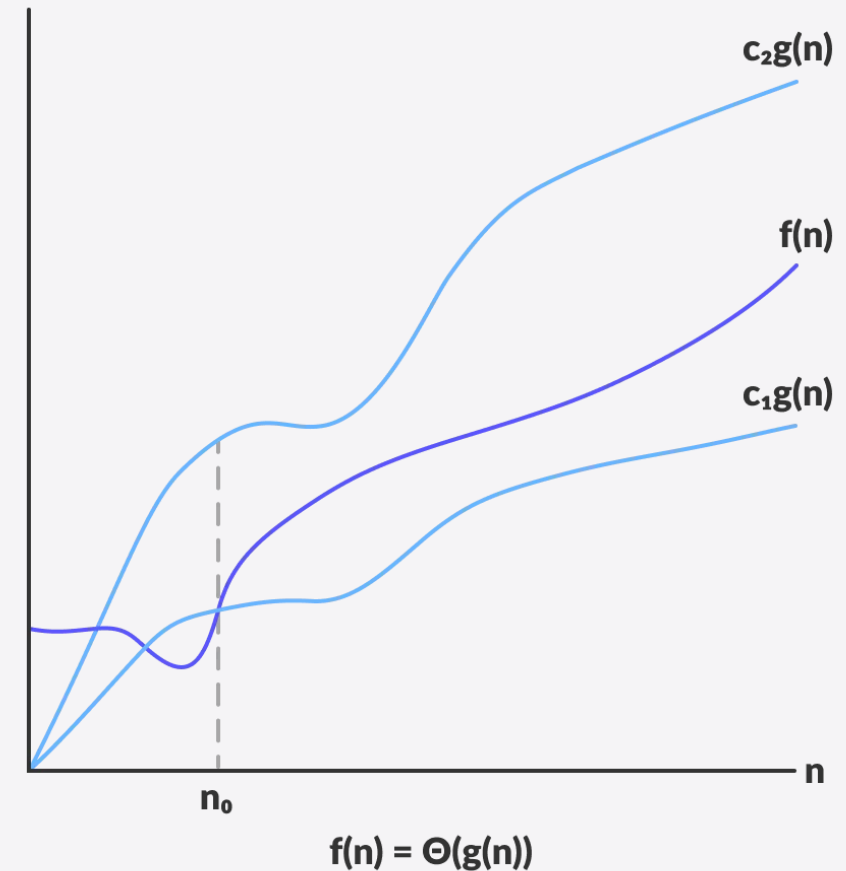
Theta notation encloses the function from above and below. Since it represents the **upper** and the **lower bound** of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.

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

The above expression can be described as a function **$f(n)$** has the upper / lower bound **$\Theta(g(n))$** if there exist positive constants **c_1** and **c_2** such that

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

If a function **$f(n)$** lies anywhere in between **$c_1 g(n)$** and **$c_2 g(n)$** for all **$n \geq n_0$** , then **$f(n)$** is said to be **asymptotically tight bound**.



Growth functions

The purpose of obtaining the order of growth of an algorithm is to give a simple characterization of the efficiency of the algorithm, being able to compare the relative performance of alternative algorithms.

Super-precision is usually not worth it. For sufficiently large inputs, the multiplicative constants and low-order terms at exact runtimes are dominated by the higher-order terms.

- $5n^2 + 3n \leq c \cdot n^2$
- $100n^{10} + 2^n + 6n \leq c \cdot 2^n$
- $1 + \log(n) + n \cdot \log(n) - n \leq c \cdot n \cdot \log(n)$
- $5n^2 + 3n < 5n^2 + 3n^2 \leq c \cdot n^2$
- $5n^2 + 3n < 8n^2 \leq c \cdot n^2$

Time complexity classifications

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

The slower growing functions are listed first.

Clase	Nombre
1	constante
$\log n$	logarítmico
n	lineal
$n \log n$	n-log-n o linealítmico
n^2	cuadrático
n^3	cúbico
2^n	exponencial
$n!$	factorial



Time complexity - Some Examples

Constant Time Complexity: $O(1)$

(These algorithms shouldn't contain loops, recursions or calls to any other non-constant time function)

- Single basic arithmetic operations (sums, subtract, etc), solve conditional statements (if, >, <, ==, etc).
- Print out once a phrase (like the classic "**Hello World**")
- **Exception.** A loop or recursion that runs a constant number of times is also considered as $O(1)$.

Linear Time Complexity: $O(n)$

(When time complexity grows in direct proportion to the size of the input)

- You must look at every item in a list to accomplish a task (e.g., find the maximum or minimum value)

Quadratic Time Complexity: $O(n^2)$

(The time it takes to run grows directly proportional to the square of the size of the input)

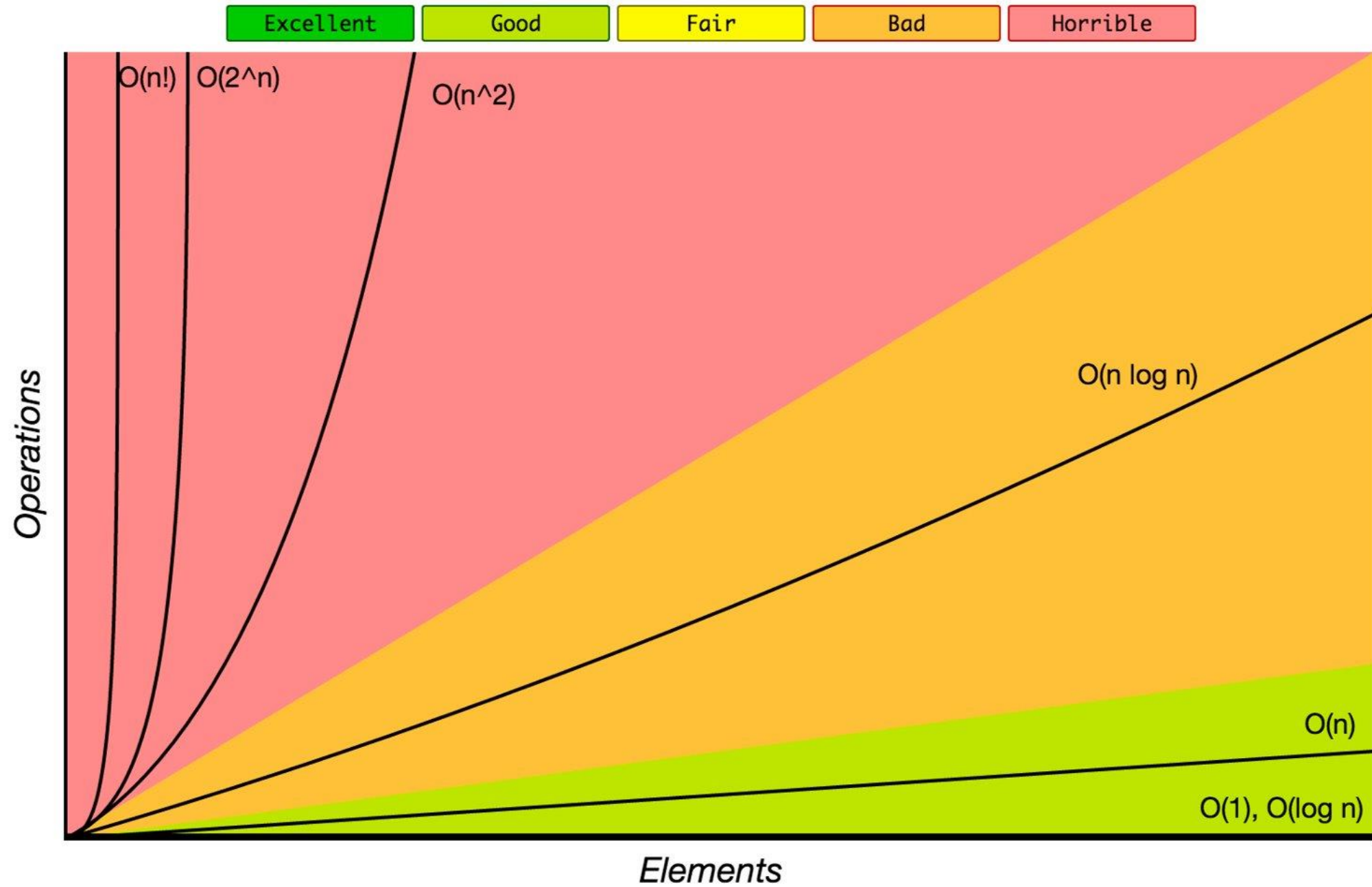
- Nested For Loops, running a linear operation within another linear operation, $n * n = n^2$.

Exponential Time Complexity: $O(2^n)$

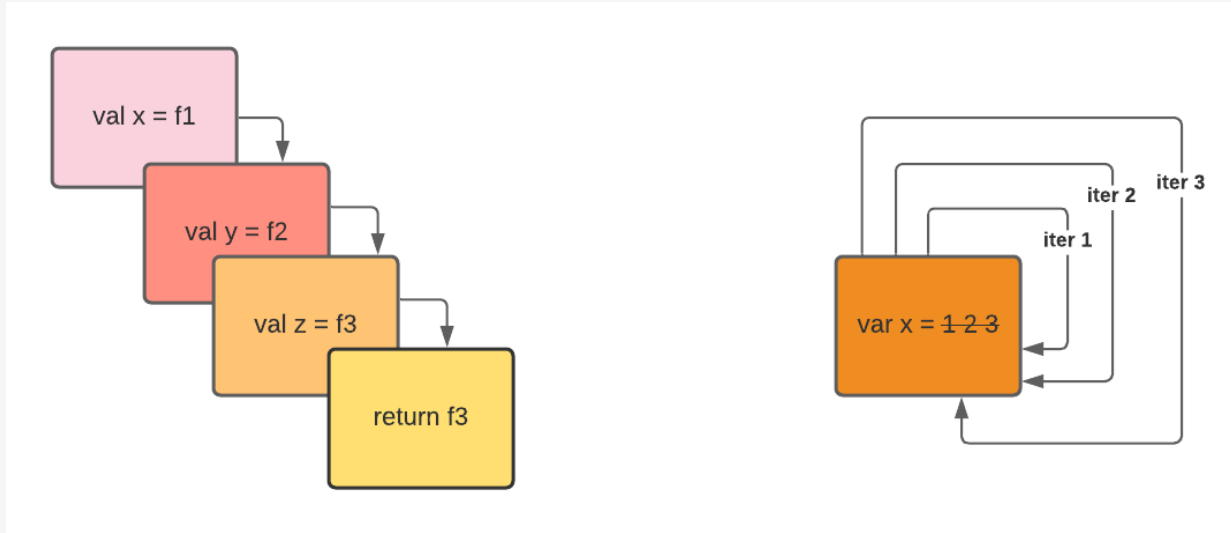
(The growth rate doubles with each addition to the input)

- Situations where you must try every possible combination or permutation on the data.

Big-O Complexity Chart



Types of algorithms



- **Iterative algorithms** contain a function that repeats until a condition is no longer met.
- In **recursive algorithms** the function calls itself until a condition is met.

Any program that can be written using iterations can be written using recursion and vice versa.

In case the algorithm does not contain iterations or recursions it means that the execution time does not depend on the size of the input (the execution time will always be constant).



Iterative Algorithms

Iterative problems (1 of 3)

1)


```
f(n)
{
    for(i=1, (i^2)<=n, i++)
        printf("Hello")
}
```

Answer:
 $O(n^{1/2})$

2)

```
f(n)
{
    for(i=1, i<n, i=i*2)
        printf("Hello")
}
```

Answer:
 $O(\log(n))$



Iterative problems (2 of 3)

3)


```
f(n)
{
    while(n>1)
        n = n/2
}
```

Answer:
 $O(\log(n))$

4)

```
f(n)
{
    i=1, s=1
    while(s<=n)
        i++
        s = s + i
        printf("Hello")
}
```

Answer:
 $O(n^{1/2})$



Iterative problems (3 of 3)

5)


```
f(n)
{
    for(i=1, i<=n, i++)
        for(j=1, j<=n, j=j+i)
            printf("Hello")
}
```

Answer:
 $O(n \cdot \log(n))$

6)

```
f(n)
{
    for(i=1, i<=n, i++)
        for(j=1, j<=i, j++)
            for(k=1, k<=100, k++)
                printf("Hello")
}
```

Answer:
 $O(n^2)$



EXTRA - Iterative problems

a) `f(n)`

```
{
    for(i=1, i<=n, i++)
        for(j=1, j<=(i^2), j++)
            for(k=1, k<=(n/2), k++)
                printf("Hello")
}
```

Answer:
 $O(n^4)$

b) `f(k)`

```
{
    n = 2^(2^k)
    for(i=1, i<=n, i++)
        j = 2
        while(j<=n)
            j = j^2
            printf("Hello")
}
```

Answer:
 $O(n \cdot \log(\log(n)))$



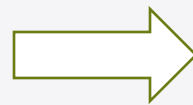
Recursive Algorithms

Analyze recursive algorithm

When an algorithm contains a recursive call to itself, we can describe its runtime through a **recurrence function**.

We define **$T(n)$** as the execution time of a problem of size **n** . If the problem size is small enough, **$n \leq c$** , the direct solution takes a constant time, **$\Theta(1)$** .

```
Factorial(n)
{
    if(n==0)
        return 0;
    else
        return n * Factorial(n-1);
}
```



$$T(n) = \begin{cases} T(n-1) + \theta(1), & \text{if } n > 0 \\ \theta(1), & \text{Otherwise} \end{cases}$$

- **Substitution Method**

```
F(n)
{
    if(n>1)
        return F(n-1);
}
```



$$T(n) = \begin{cases} T(n-1) + 1, & \text{if } n > 1 \\ 1, & \text{Otherwise} \end{cases}$$

$$T(n) \leq T(n-1) + 1$$

$$\leq (T(n-2) + 1) + 1 = T(n-2) + 2$$

$$\leq (T(n-3) + 1) + 2 = T(n-3) + 3$$

$$\leq (T(n-4) + 1) + 3 = T(n-4) + 4$$

....

It looks like **$T(n-j) + j$**

If **$j = (n-1)$** then

$$\mathbf{T(1) + (n-1)}$$

By definition **$T(1) = 1$** , then

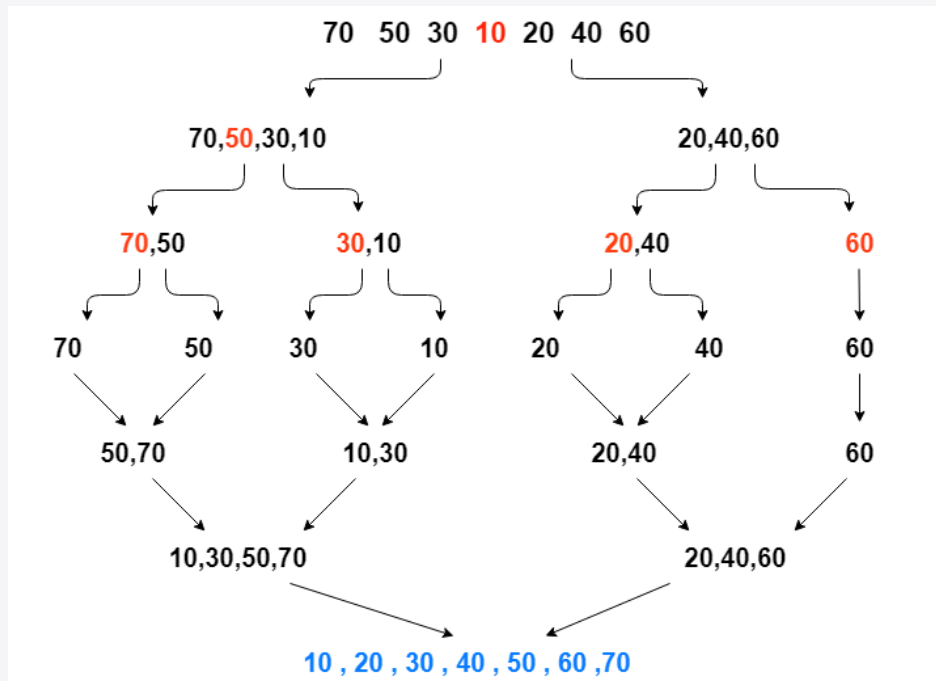
$$\mathbf{T(n) \leq 1 + (n-1)}$$

This is **$O(n)$**

Analyze recursive algorithm - Divide & Conquer

If we assume that we divide our problem into **a** subproblems, each **1/b** the size of the original problem. It would take a time of **T(n/b)** to solve a subproblem of size **n/b**, and therefore it takes a time of **a*T(n/b)** to solve the a subproblems.

If not explicitly stated, we assume that if the problem size is small enough, **n ≤ c**, the direct solution takes a constant time, **Θ(1)**.



Merge sort recurrence function

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n/2) + \Theta(n), & \text{if } n > 1 \end{cases}$$

Master Method

The **Master Method** depends on the following Theorem:

Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence:

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$, and if $a \cdot f\left(\frac{n}{b}\right) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$.

regularity condition

Master Theorem Limitations

$$T(n) = a T\left(\frac{n}{b}\right) + f(n)$$

The master theorem cannot be used if:

- **T(n)** is not monotone. eg. $T(n) = \sin(n)$
- **f(n)** is not a polynomial. eg. $f(n) = 2^n$
- **a** is not a constant. eg. $a = 2n$
- **a < 1**

Recursive problems (1 of 1)

1) $T(n) = 2 T\left(\frac{n}{2}\right) + \theta(n)$

Answer:

$T(n) = O(n \log(n))$

2) $T(n) = T\left(\frac{2n}{3}\right) + \theta(1)$

Answer:

$T(n) = O(\log(n))$

3) $T(n) = 8 T\left(\frac{n}{2}\right) + \theta(n^2)$

Answer:

$T(n) = O(n^3)$

4) $T(n) = 7 T\left(\frac{n}{3}\right) + \theta(n^2)$

Answer:

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

EXTRA - Recursive problems

a) $T(n) = 7 T\left(\frac{n}{2}\right) + \theta(n^2)$

Answer:
 $T(n) = O(n^{2.8})$

b) $T(n) = 9 T\left(\frac{n}{3}\right) + \theta(n)$

Answer:
 $T(n) = O(n^2)$

c) $T(n) = 3 T\left(\frac{n}{4}\right) + \theta(n \log(n))$

Answer:
 $T(n) = O(n \log(n))$

d) $T(n) = 2 T\left(\frac{n}{2}\right) + \theta(n \log(n))$

Answer:
 $T(n) = ? O(n \log(n)) ?$

Common Data Structure Operations

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