

The background is a light gray field covered with a dense, irregular pattern of small, semi-transparent circles in various colors including green, blue, purple, brown, orange, and red. Faint, thin white lines form concentric arcs and radial patterns across the background, suggesting a grid or coordinate system.

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** a_1, a_2, \dots, a_n of numbers, where n is a **non-negative integer**, we can write the **finite sum**:

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

Note. if $n=0$, 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**.

- Given a **sequence** a_1, a_2, \dots, a_n of numbers, where n is a **non-negative integer**, we can write the **finite sum**:

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

Note. if $n=0$, 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:

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

or

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

Summation properties

- For any real number **c** and any **finite sequence** **a**₁, **a**₂, ..., **a**_n and **b**₁, **b**₂, ..., **b**_n. There are **properties** that allow us to write summations in different ways:

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

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

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

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

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

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

$$\sum_{k=1}^n \frac{1}{k} = \ln n$$

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**

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 **product to a summation**:

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



Big-O notation (O-notation)

It represents the upper bound of the running time of an algorithm. Thus, it gives the **worst-case** scenario.

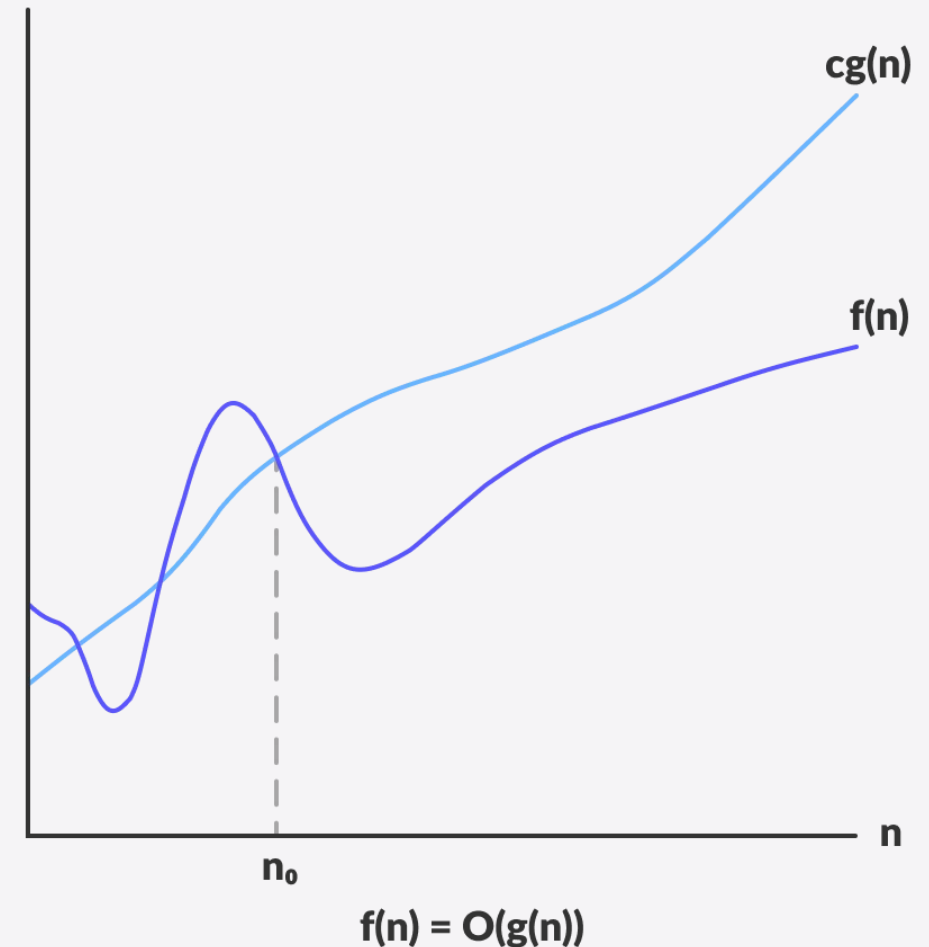
$$f(n) = O(g(n))$$

The function **f(n)** describes 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 n_0 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)

It represents the lower bound of the running time of an algorithm. Thus, it provides the **best-case** scenario.

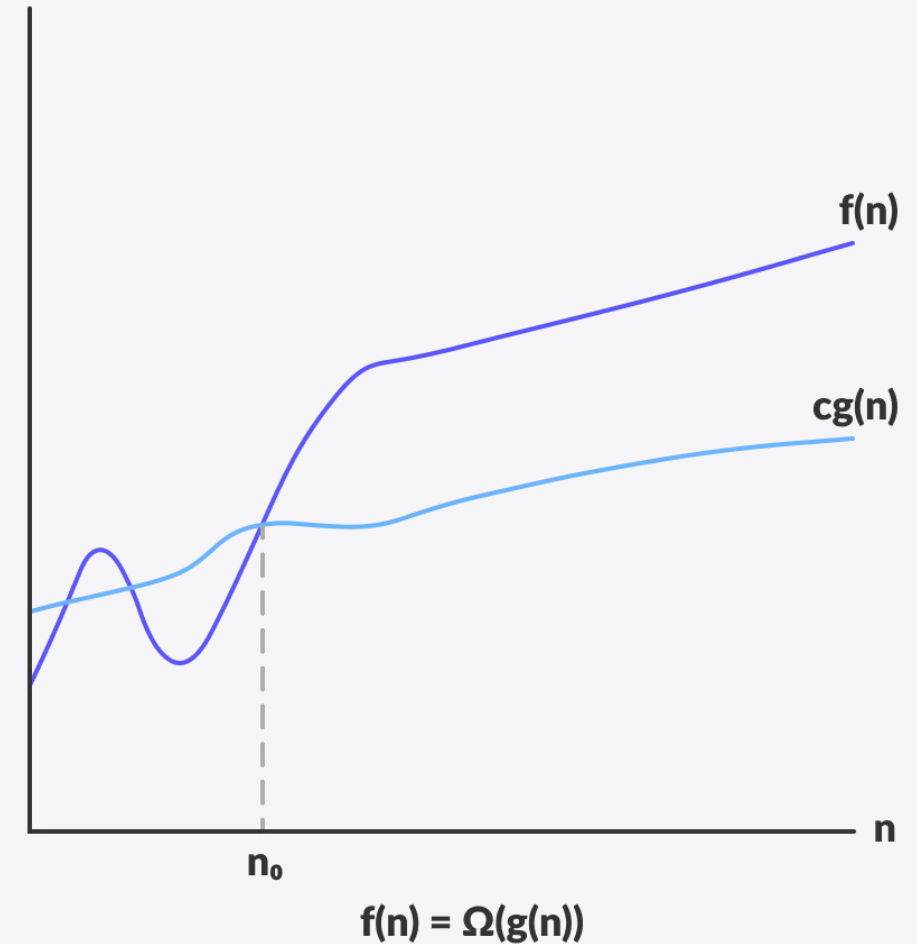
$$f(n) = \Omega(g(n))$$

The function **$f(n)$** describes 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)

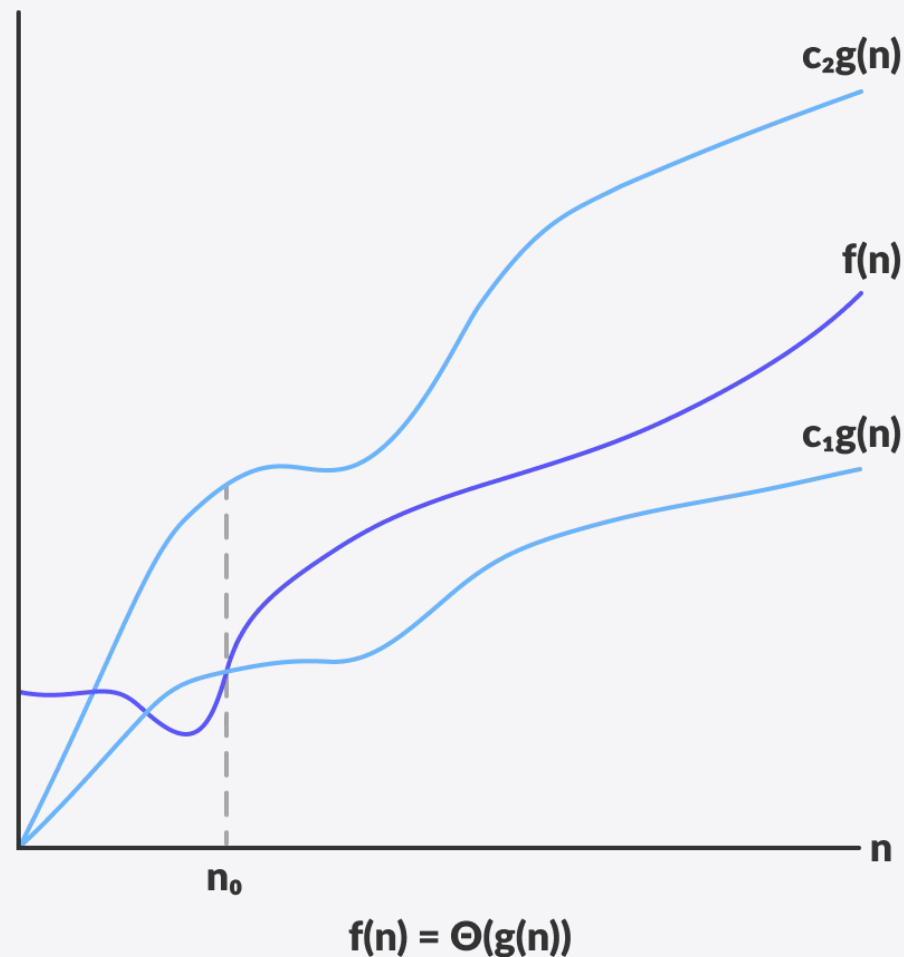
This notation represents the **upper** and the **lower bound** of the running time of an algorithm. Thus, it provides the **average-case** scenario.

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

The 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**).

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

The slower growing functions are listed first.



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)



Time complexity - Some Examples

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

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

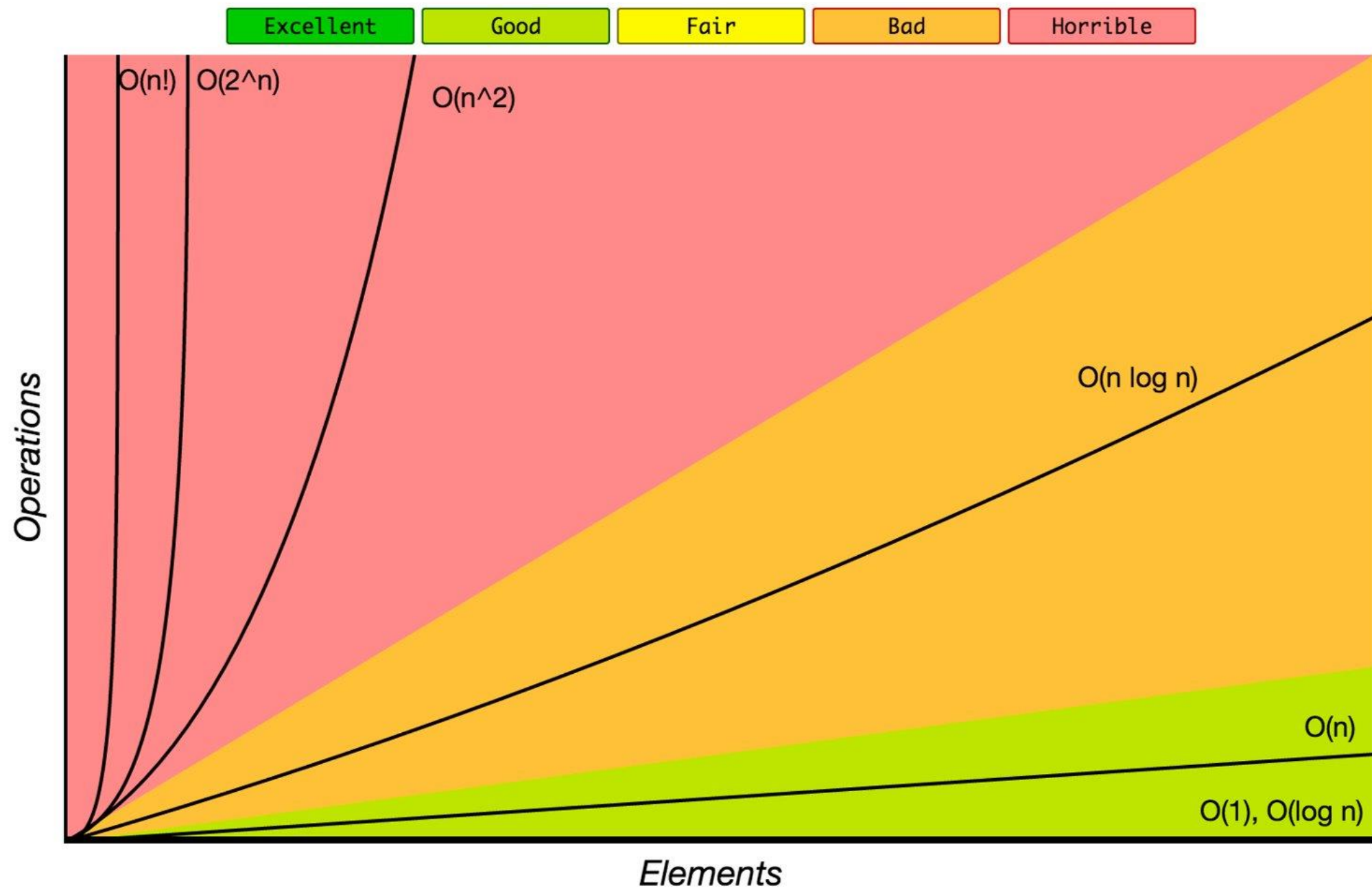
- **Nested For Loops**, running a linear operation within another linear operation, $n * n = n^2$.

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

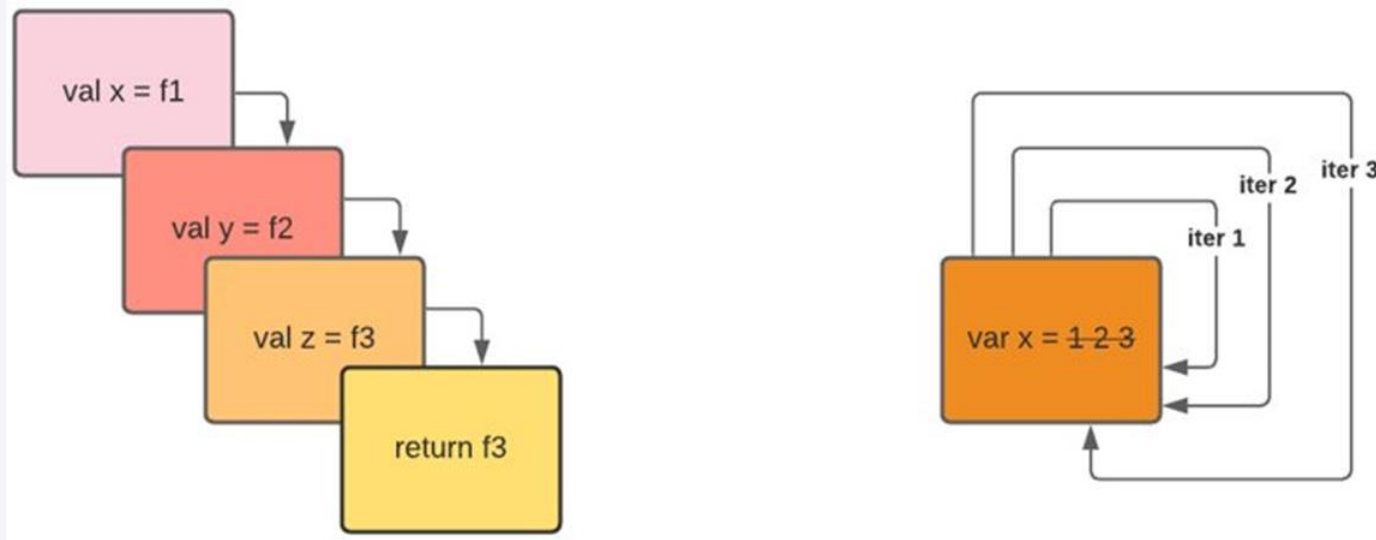
(The growth rate doubles with each addition to the input)

- 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.
- **Recursive algorithms** the function calls itself until a condition is met.

In case the algorithm does not contain **iterations** or **recursions**, the execution time does not depend on the size of the input (**constant**).

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



Logarithms - Definitions & Properties

Definitions

- $\text{Log } n = \log_2 n$ (binary logarithm)
- $\text{Ln } n = \log_e n$ (natural logarithm)
- $\text{Log}_a(1) = 0$, for any value of a
- $\text{Log}_a(0)$ is undefined
- $\text{Log}_a(-x)$ undefined for negative values
- $\text{Log}_a(a) = 1$

Properties

For $a, b, c > 0$ y n

- $\text{Log}_c(a * b) = \log_c(a) + \log_c(b)$
- $\text{Log}_a(b / c) = \log_a(b) - \log_a(c)$
- $\text{Log}_b(a^n) = n * \log_b(a)$
- $\text{Log}_b(a) = \log_c(a) / \log_c(b)$, for any c
- $\text{Log}_b(a) = 1 / \log_a(b)$
- $a^{\text{Log}_b(c)} = c^{\text{Log}_b(a)}$



Iterative Algorithms

Iterative problems (1 of 4)

1)

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

Answer:
 $O(n)$

2)

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

Answer:
 $O(n^2)$

Iterative problems (2 of 4)

3)


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

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

4)

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

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



Iterative problems (3 of 4)

5)

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

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

6)

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

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

Iterative problems (4 of 4)

7)

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

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

8)

```
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 (1 of 2)

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

EXTRA - Iterative problems (2 of 2)

c)

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

Answer:
 $O(n^2)$

d)

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

Answer:
 $O(n^2 * \log(n))$



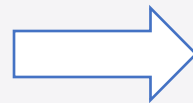
Recursive Algorithms

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);
}
```



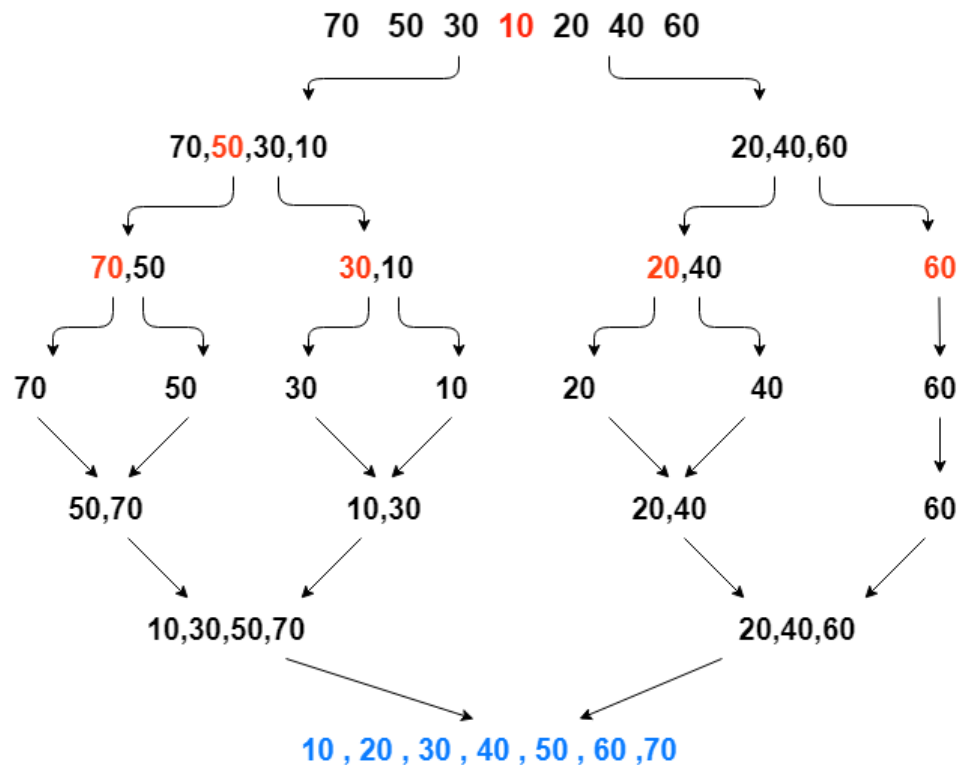
$$T(n) = \begin{cases} T(n-1) + \theta(1), & \text{if } n > 0 \\ \theta(1), & \text{Otherwise} \end{cases}$$

Analyze recursive algorithm - Divide & Conquer

If 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, **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

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>	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$




Problem classification



Tractability

What constitutes reasonable time?

Standard working definition: **polynomial time**

- **Polynomial time:** $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
 - **Not in polynomial time:** $O(2^n)$, $O(n^n)$, $O(n!)$
- 



Tractability

What constitutes reasonable time?

Standard working definition: **polynomial time**

- **Polynomial time:** $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- **Not in polynomial time:** $O(2^n)$, $O(n^n)$, $O(n!)$

Are ALL problems solvable in polynomial time?






Tractability

What constitutes reasonable time?

Standard working definition: **polynomial time**

- **Polynomial time:** $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- **Not in polynomial time:** $O(2^n)$, $O(n^n)$, $O(n!)$

Are ALL problems solvable in polynomial time?

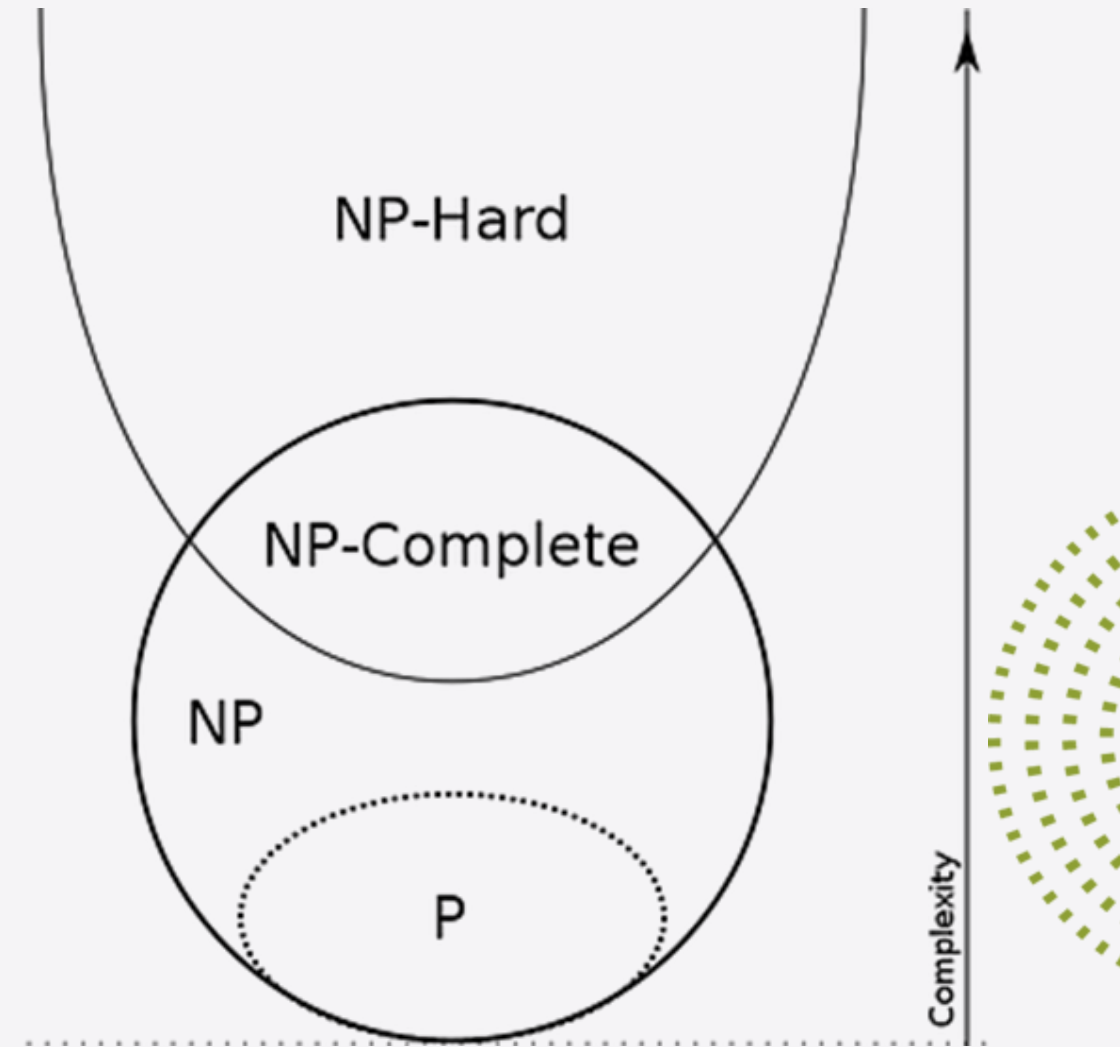
- No, there are problems not solvable by any computer, no matter how much time is given.
- 

Introduction

In theoretical computer science, the classification and complexity of common problem definitions have two major sets; **Polynomial time (P)** and **Non-deterministic Polynomial time (NP)**.

In the case of rating from easy to hard:

- Easy : **P**
- Medium : **NP**
- Hard : **NP Complete**
- Hardest : **NP Hard**






P problem

The class **P** consists of those problems that are solvable in polynomial time, e.g., $O(n^k)$ where **k** is a **constant** and **n** is **size of input**.

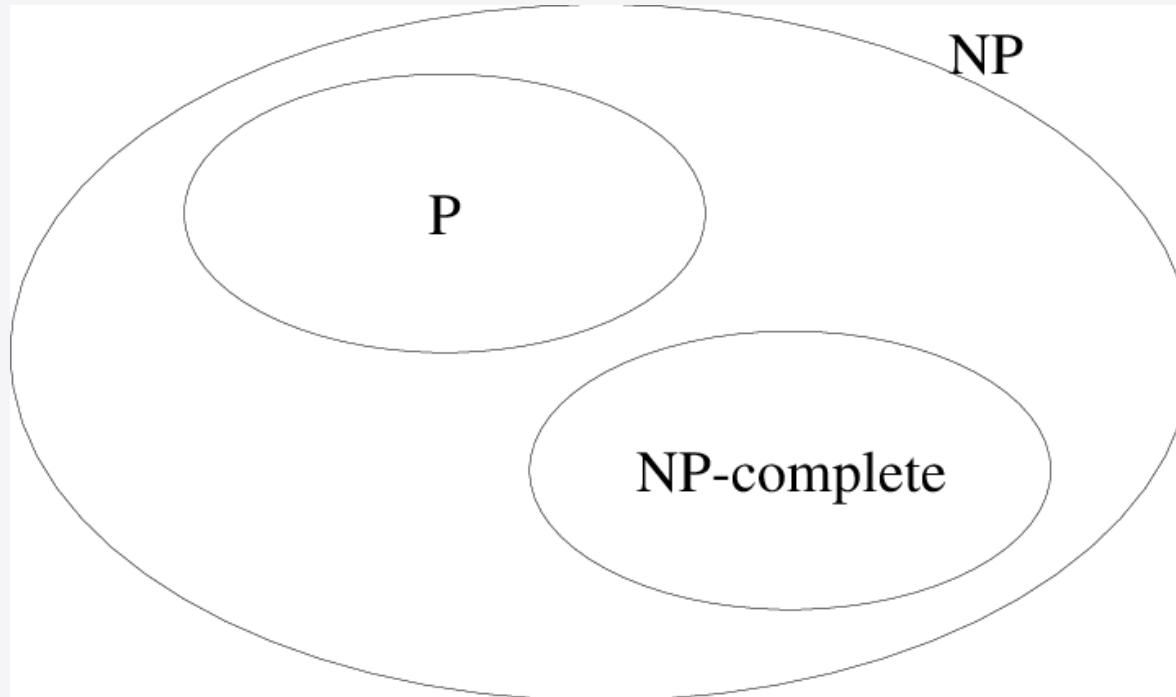
these are basically tractable; they are solved by deterministic algorithms.

Examples

- A set of decision problem with **yes/no** answer.
 - Calculating the **greatest common divisor (GCD)**.
 - sorting **n** numbers in ascending or descending order.
 - search the element in the list
- 

NP problem

NP stands for **non-deterministic polynomial** time if its solution can be guessed and **verified** in polynomial time; **non-deterministic** means that no particular rule is followed to make the guess.





A practical Example #1

Perhaps the most famous **exponential-time** problem in **NP**, for example, is **finding prime factors of a large number**.

Verifying a solution just requires multiplication, but solving the problem seems to require systematically trying out lots of candidates.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100





A practical Example #2

Another example of this is **Sudoku**. To phrase it as a decision problem:

"Given a sudoku puzzle, does it have a solution?"

It may take a long time to answer that question (because you must solve it), but if someone gives you a solution you can very quickly **verify** that the solution is correct.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9



A practical Example #2

Another example of this is **Sudoku**. To phrase it as a decision problem:

"Given a sudoku puzzle, does it have a solution?"

It may take a long time to answer that question (because you must solve it), but if someone gives you a solution you can very quickly **verify** that the solution is correct.

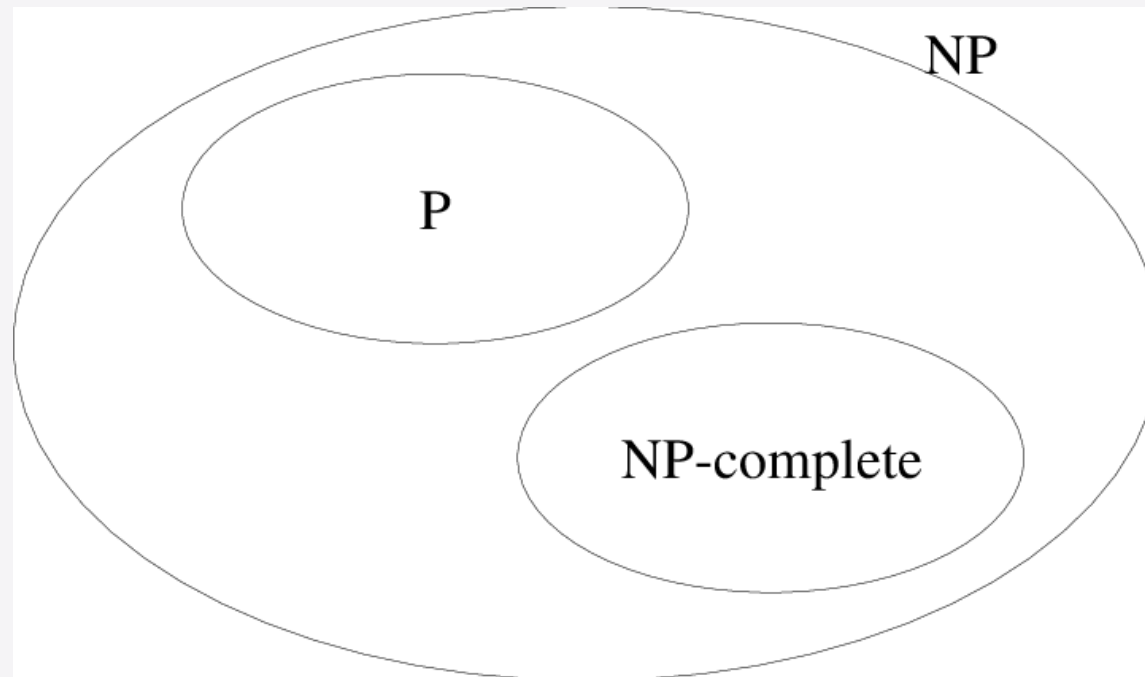
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

That's what NP is -- the set of all decision problems where you can efficiently verify the answer.

NP-Completed problem

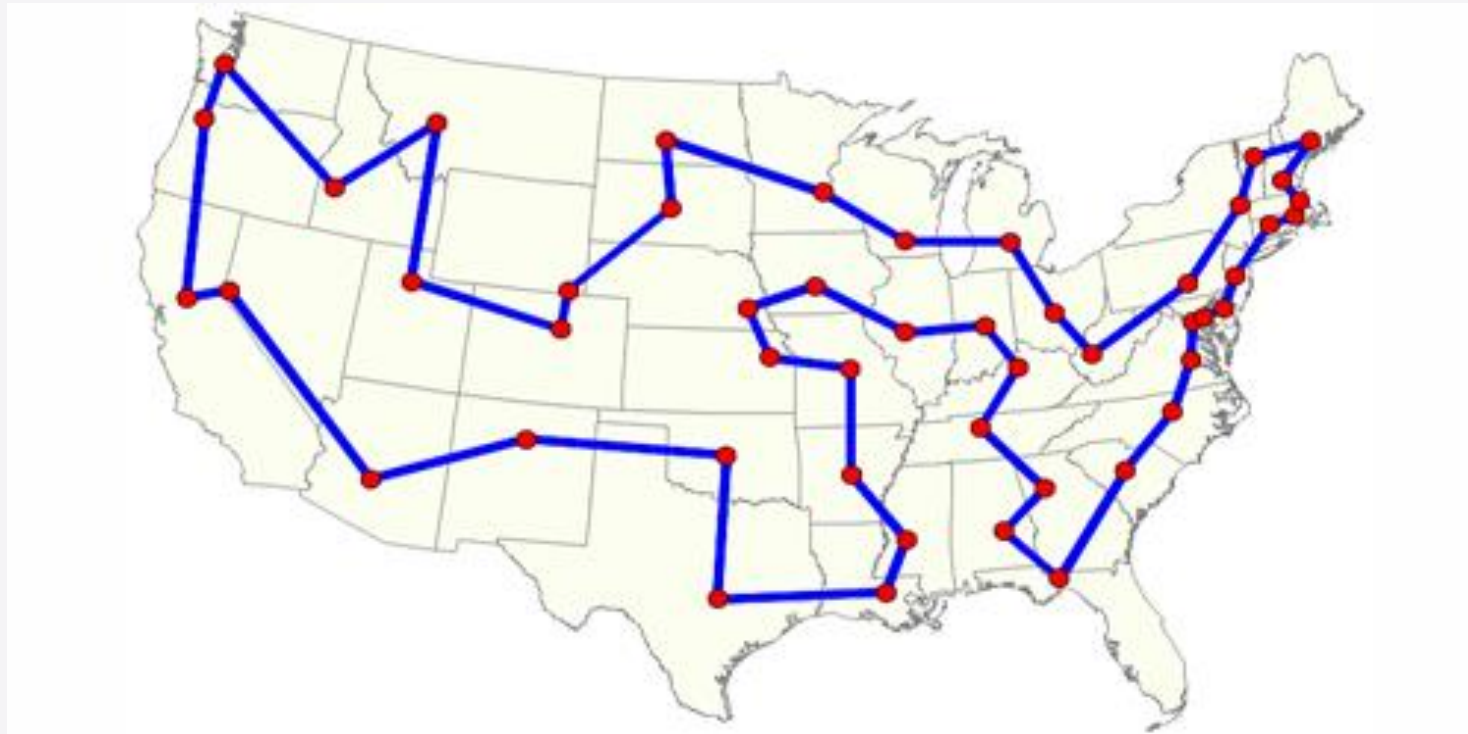
Any of a class of computational problems for which **no efficient solution algorithm has been found**.

The vast majority of **NP problems** whose solutions seem to require **exponential time**.



In real life, NP-complete problems are fairly common, especially in large **scheduling tasks**.

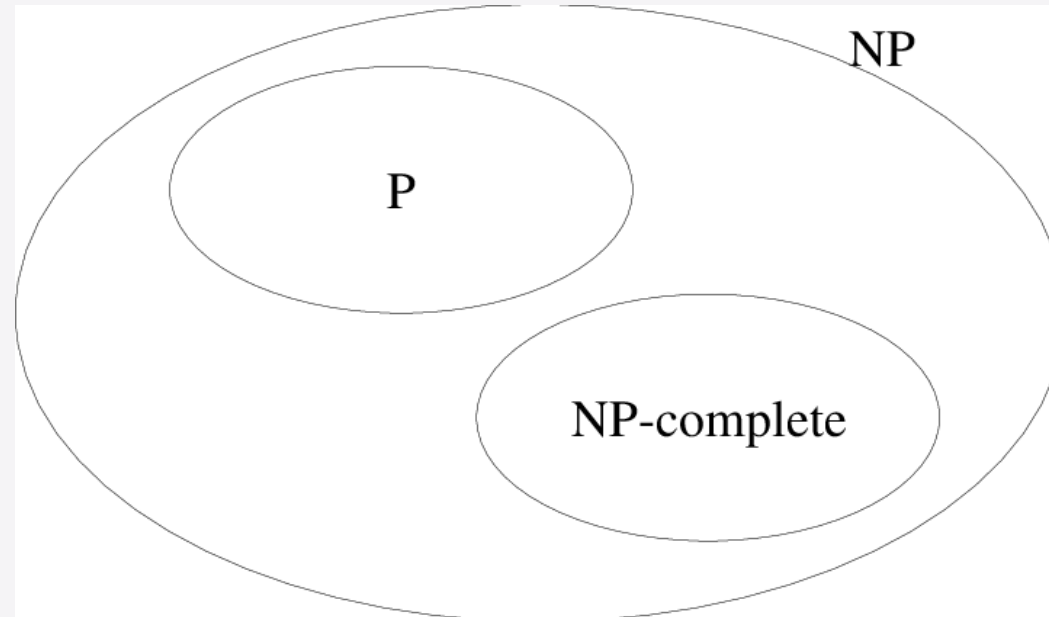
The most famous **NP-complete** problem, for instance, is the so-called **traveling-salesman problem**: Given **N** cities and the **distances** between them, can you find a route that hits all of them but is shorter than... **whatever limit you choose to set?**




Does $P = NP$?

The question means **"If the solution to a problem can be verified in polynomial time, can it be found in polynomial time?"**.

Part of the question's allure is that most **NP problems** whose solutions seem to require exponential time are what's called **NP-complete**, meaning that **a polynomial-time solution to one can be adapted to solve all the others**.





Given that **P** probably doesn't equal **NP**, however – that efficient solutions to **NP** problems will probably never be found – **what's all the fuss about?**

the **P-versus-NP** problem is important for deepening our understanding of computational complexity.

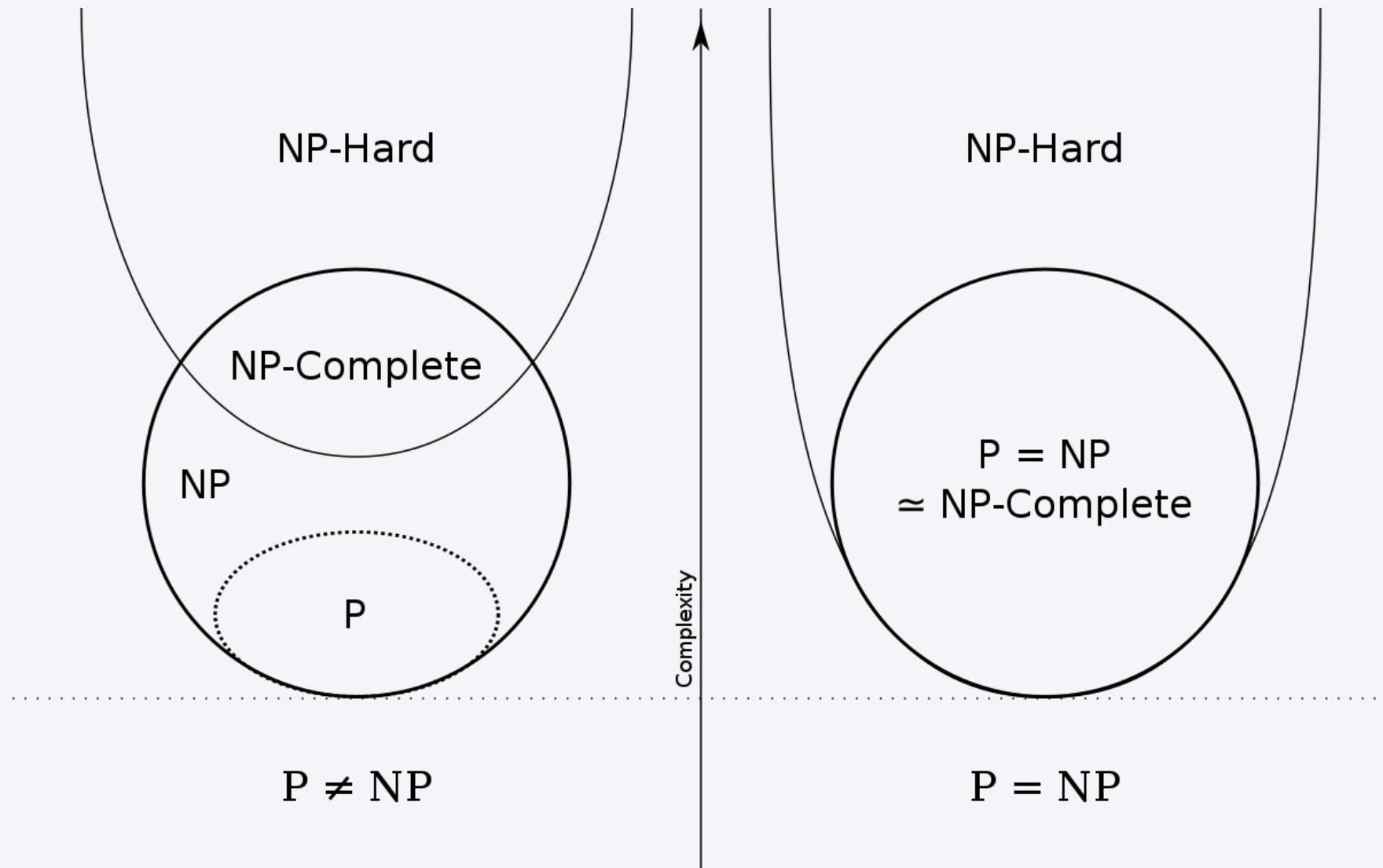
A major application is in the **cryptography area**, where the security of cryptographic codes is **often ensured by the complexity of a computational task**.



Additionally, the excitement around **quantum computation** really boiled over.

If **P=NP** then the algorithm in **NP** can be solved in polynomial time.

The problem in **NP-Hard** cannot be solved in polynomial time, until **P = NP**.





NP-Completed problem

A lot of times you can solve a problem by reducing it to a different problem. We can reduce **Problem B** to **Problem A** if, given a solution to **Problem A**, we can easily construct a solution to **Problem B**. (In this case, "easily" means "in polynomial time.").

- A Problem **X** is **NP-Hard** if there is an **NP-Complete** problem **Y**, such that **Y** is reducible to **X** in polynomial time.
- A problem is **NP-hard** if all **NP** problems are reducible to it in polynomial time.

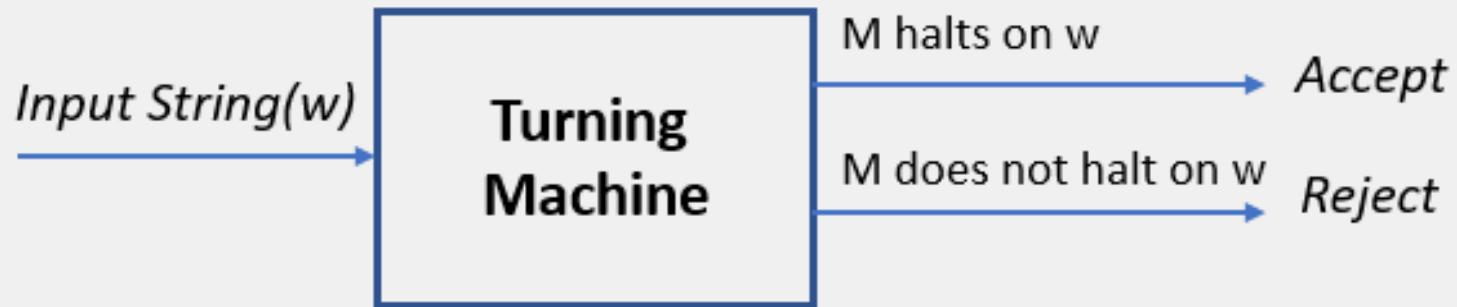




NP-Hard problems are as hard as **NP-Complete problems**, although it might, in fact, be harder. **NP-Hard problems** are not only **hard to solve** but are **hard to verify** as well. In fact, some of these **problems aren't even decidable**.

The Halting problem – Given a program/algorithm will ever halt or not?

Halting means that the program on certain input will accept it and halt or reject it and halt and it would never go into an infinite loop. Basically, halting means terminating.



About Comprehensive Activity (Competency Evidence)

- Asegurarse que funcione en otras computadoras (uso de librerías estándar).
- Recordar el propósito de este curso, utilizar las implementaciones de las estructuras y/o algoritmos que se realizaron previamente (no usar librerías).
- Poner atención y leer detenidamente los requerimientos y criterios de evaluación de cada actividad (se seguirán al pie).
- **Precaución:** Varias actividades dependen de otras

Mentalidad de vendedor, no de ingeniero.

- De ser necesario, el documento debe contener las instrucciones de uso (como ingresar los datos, formato, etc). Evitar pedirle mucho al usuario.
- Se debe comprobar el correcto funcionamiento del programa a manera de demostración: llamar a las funciones implementadas y "testearlas" en diferentes situaciones (casos atípicos o límite incluidos) para probar su correcto funcionamiento y robustez de su programa.