Introduction to Algorithms & Data Structures

-

PALISSON Antoine

Table of **Contents**

01

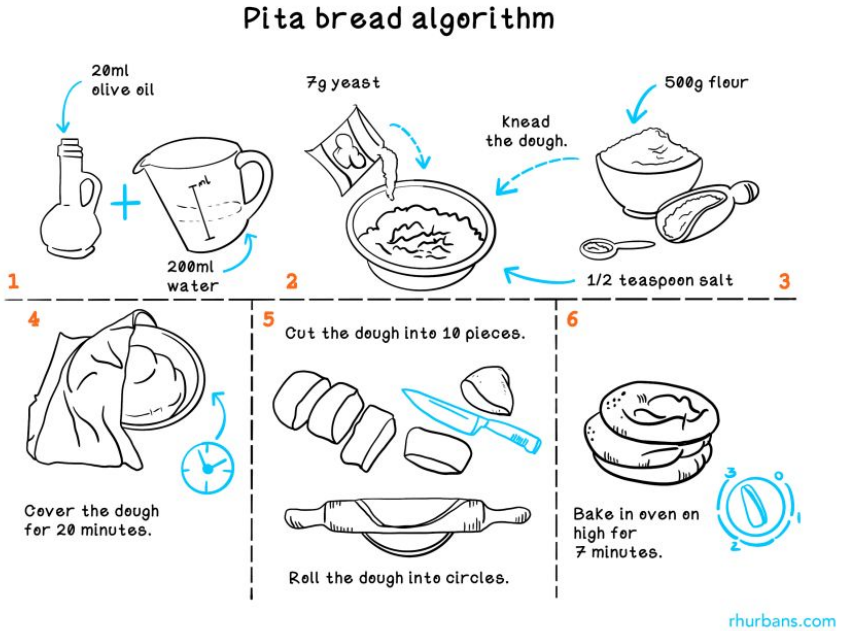
Introduction

Introduction to Algorithms & Data Structures

What is an algorithm?

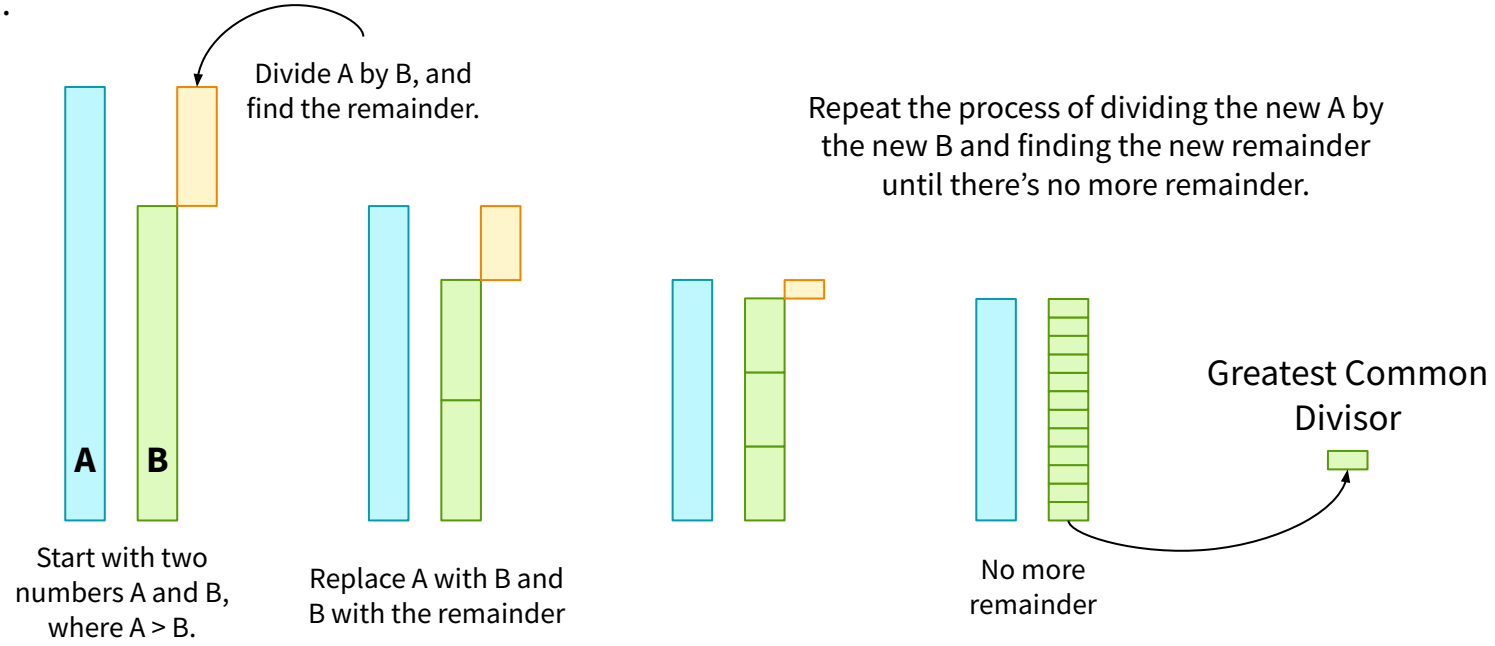
An **algorithm** is a set of instructions designed to perform a specific task. This set of instructions is written in a step-by-step manner, and it is executed by a computer.

Think of an algorithm as a recipe in a cookbook. Just as a recipe provides step-by-step instructions on how to prepare a dish, including what ingredients to use, in what order, and how to combine them, an algorithm gives clear, step-by-step instructions to a computer on how to perform a task or solve a problem. The ingredients in cooking can be seen as the data, and the pots, pans, and utensils can be thought of as data structures that hold and organize these ingredients as you prepare the dish.



The First “True” Algorithm (?)

The first algorithm recorded in history is often attributed to the mathematician Euclid around 300 BC. This algorithm, known as Euclid's algorithm, is a method for finding the greatest common divisor of two integers. The greatest common divisor of two numbers is the largest number that divides both of them without leaving a remainder.

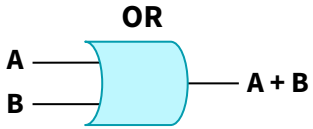


Boolean Algebra

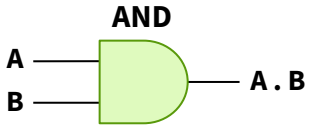
George Boole, an English mathematician and logician in the 19th century, developed **Boolean algebra** showing that logical reasoning could be represented mathematically which later laid the foundational groundwork for the digital logic used in modern computers and computer algorithms.

On

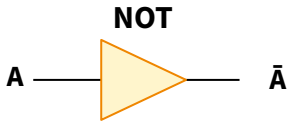
Off



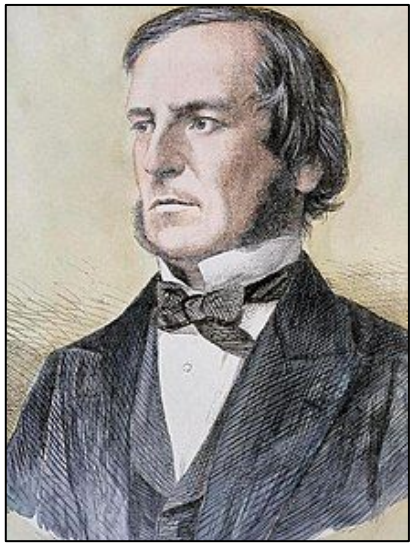
A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1



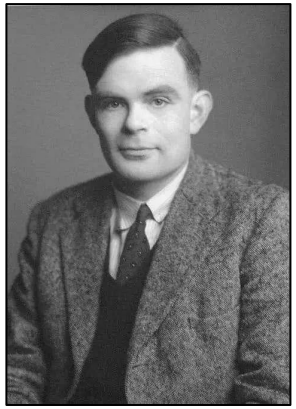
A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1



A	Output
0	1
1	0

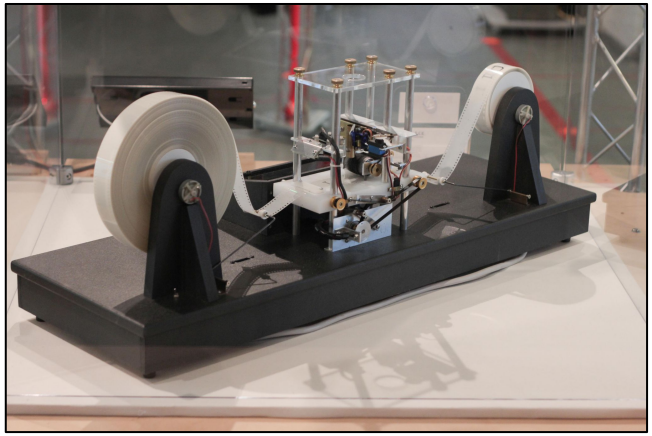


Alan Turing and the Concept of Algorithm

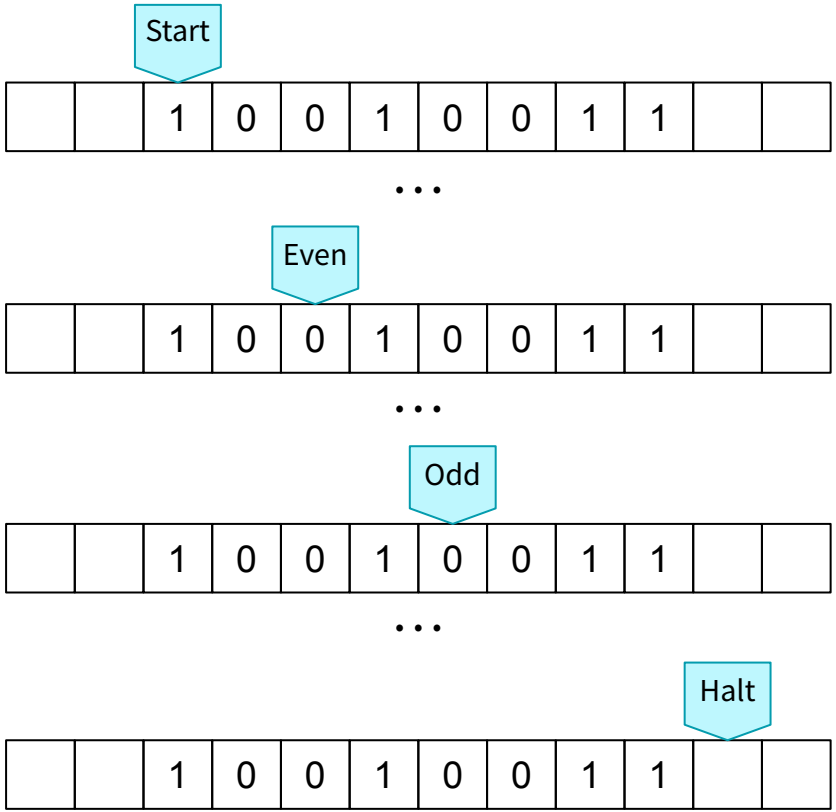


Alan Turing's work in the mid-1930s, particularly his formulation of what is now known as the **Turing machine** provided a formal and concrete definition of an algorithm as a set of instructions run on a computational machine. It showed that a very simple set of rules and operations could be used to represent any computable function.

The machine operates on an infinite tape divided into cells. Each cell can contain a symbol, which might be a digit or a blank space. A tape head reads and writes symbols to the tape. The machine has a state register that stores the state of the Turing machine, one of a finite number of conditions. The behavior of the machine is dictated by a set of rules that instructs the machine what to do, depending on its current state and the symbol it is reading.



Alan Turing and the Concept of Algorithm



This Turing Machine has four states: Start, Odd, Even and Halt.

It can perform the following actions:

- Each time the head reads a 0 in the "Start" or "Even" state, it transitions to the "Odd" state, and vice versa. Then the head moves to the right.
- If the head reads a 1, it stays in the current state and moves to the right.
- When it reads a blank, it stops.

Introduction of Public Key Cryptography

The development of computers during the mid-20th century revolutionized algorithms, making it possible to perform complex calculations rapidly and reliably. This laid the groundwork for **encryption algorithms**, such as RSA, that transformed communication security and are fundamental to modern internet security.

RSA Algorithm

Key Generation

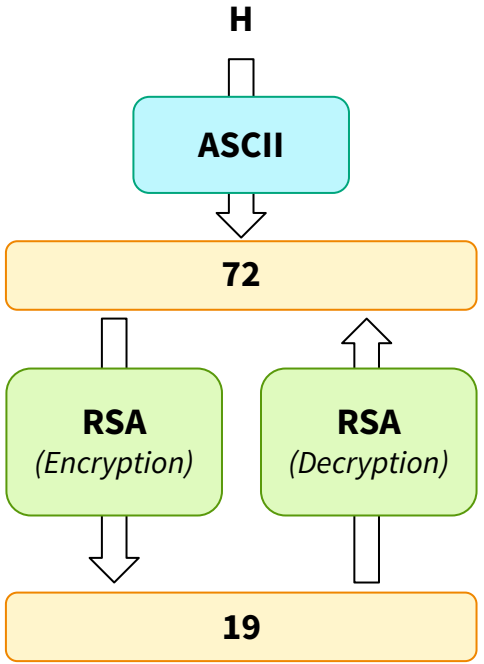
Select p,q	p and q both prime; $p \neq q$
Calculate $n = p \times q$	
Calculate $\phi(n) = (p-1)(q-1)$	
Select integer e	$\gcd(\phi(n),e) = 1;\ 1 < e < \phi(n)$
Calculate d	$de \bmod \phi(n) = 1$
Public key	$KU = \{e,n\}$
Private key	$KR = \{d,n\}$

Encryption

Plaintext:	$M < n$
Ciphertext:	$C = M^e \pmod n$

Decryption

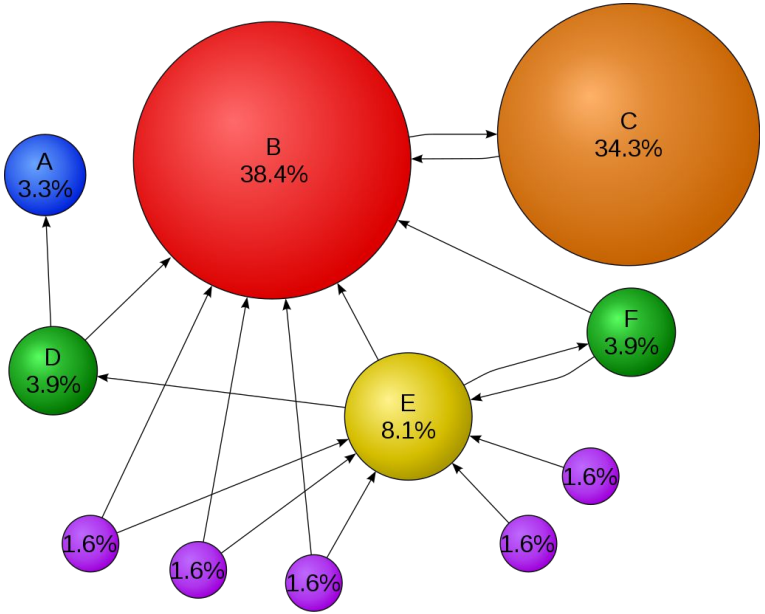
Plaintext:	C
Ciphertext:	$M = C^d \pmod n$



$p = 13$
$q = 11$
$\phi(n) = 60$
$e = 7$
$d = 43$

Development of the Internet and Search Algorithms

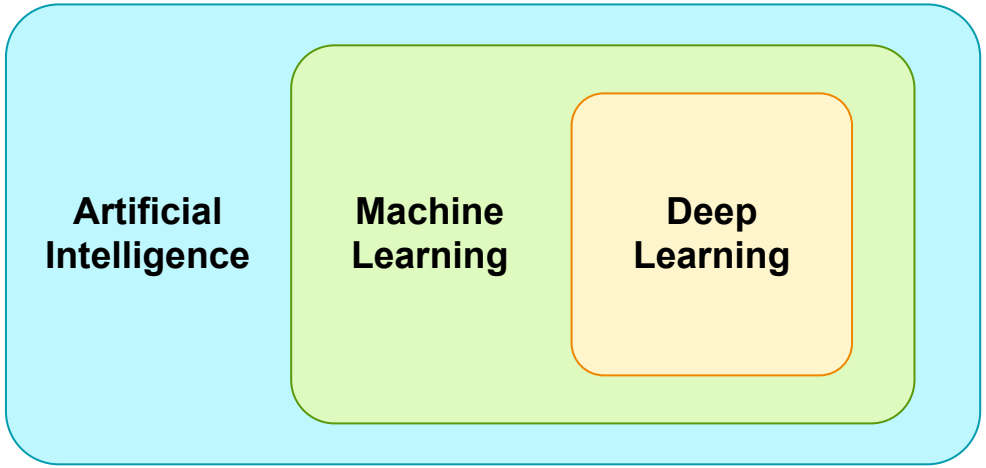
The explosion of information on the internet towards the late 20th century necessitated the development of sophisticated algorithms for searching that information. These algorithms were built to solve the problem of finding relevant information in the vast amount of data on the World Wide Web.

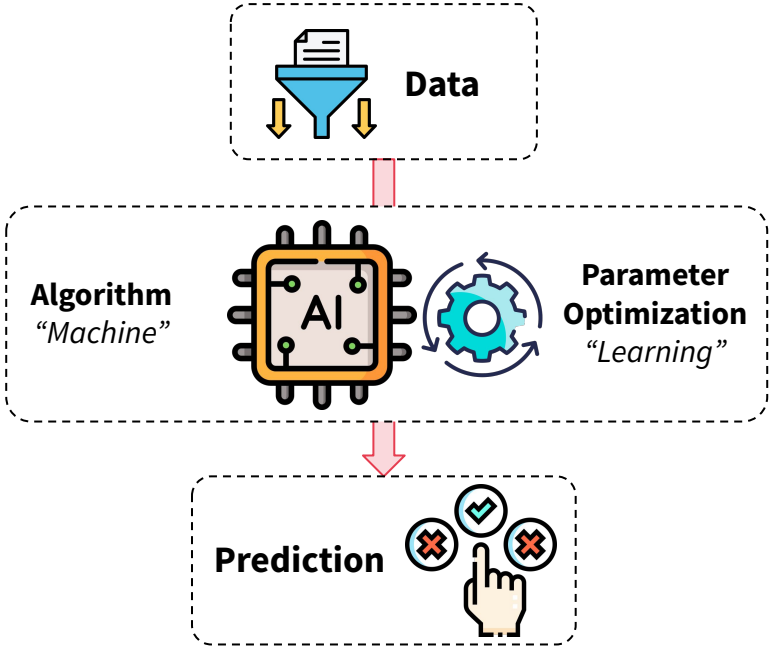


Before **PageRank** (introduced by Google), search engines primarily ranked pages based on how many times a search term appeared on the page. PageRank introduced a new method that counted the number and quality of links to a page to estimate how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites. It shifted the focus from just the content of the web pages themselves to the relationships between them.

Machine Learning Algorithms

The 21st century has seen remarkable strides in the field of artificial intelligence, fueled by the advancement of algorithms that can learn from data. Unlike traditional algorithms, which follow a clear set of instructions to perform a task, Machine Learning algorithms improve automatically through experience.

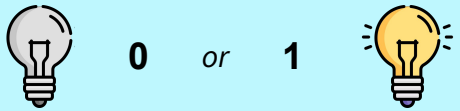




How is Data represented?

Data is represented using the **binary system**, a method of storing and processing information using two symbols. It is the fundamental language of computers, like an alphabet. Just as we use different letters to form words, computers use the binary system, which consists of just two 'letters', also called digits: 0 and 1. Every data that a computer handles is broken down into a combination of these two digits.

A **bit** is the smallest unit of data for computers, and it's one of those binary digits - either a 0 or a 1.



A **byte** is a group of **8 bits**. Since 8 bits can create many different patterns (from 00000000 to 11111111), a byte can represent **256 different values**.

There are many bit-width representations in computing. The most widely used are **8-bit** (256 memory locations), **16-bit** (65 536 memory locations), **32-bit** (4+ billion memory locations), and **64-bit** (more than eighteen quintillion memory locations which is 18 followed by 18 zeros).

How is Data stored?

This data is kept in **storage devices** like a hard drive, solid-state drive, or a memory card, where the computer can read and write the data in the form of binary code as needed.

```
01000001011011000110011101101111
01110010011010010111010001101000
01101101001000000110000101101110
01100100001000000110100001100001
01110100011000010010000001110011
01110100011100100111010101100011
01110100011101010111001001100101
…
```

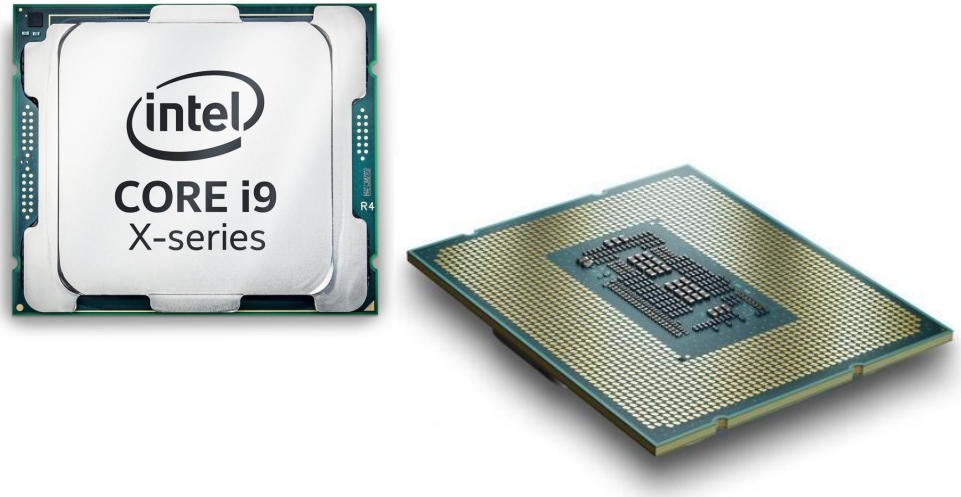
One these storage devices, a TB - a Terabyte - represents 8,000,000,000,000 bits.

How is Data used?

Data is processed by the computer **central processing unit** (CPU). The CPU interprets the binary data according to the instructions of the software running on the computer.

*Every CPU can perform operations, such as executing an instruction or accessing memory, in what's called a **cycle**. The more cycles a CPU can perform in a second, the more operations it can theoretically execute.*

*The **Hz** (Hertz) of a CPU is a measure of how many cycles per second the processor can execute. It's a key metric for understanding the performance capabilities of a CPU. Modern CPUs can **perform billions of cycles per second** (**GHz**).*



Power & Memory Evolution

The evolution of power and memory in computers is a story of exponential growth and miniaturization, largely described by Moore's Law, which predicted that the number of transistors on a microchip would double approximately every two years. With more power and memory, more complex algorithms can be run in reasonable timeframes. Algorithms that were once theoretical due to their computational requirements have become practical.

1969

Apollo mission

***CPU**: 2 MHz*
***RAM**: 8 KB*

1984

Olivetti PCS

***CPU**: 100 MHz (x50)*
***RAM**: 8 MB (x1000)*

2002

Toshiba Satellite

***CPU**: 2.4 Ghz (x24)*
***RAM**: 512MB (x64)*

2023

Iphone 15 (2023)

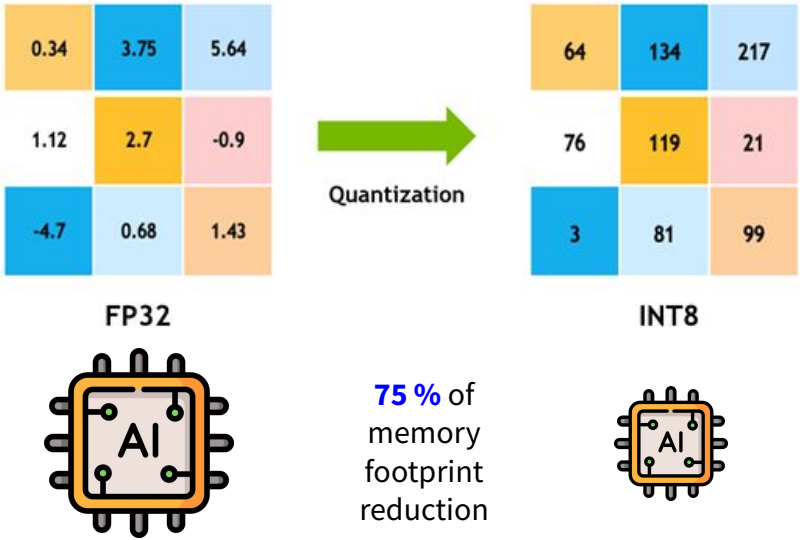
***CPU**: 2 x 3,7 Ghz + 4 x 2.2 Ghz (x8)*
***RAM**: 8GB (x16)*

Algorithm Optimization

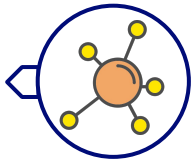
The Binary Number System is the core of computer arithmetic, data storage, and manipulation, directly influencing the design and **optimization of algorithms**. It enables a clearer grasp of a computer's architecture and limitations, such as memory usage, which in turn inform **complexity** analysis and performance tuning. Moreover, manipulating the bit representation is essential in areas like cryptography, compression, and error correction.

Quantization is the process of lowering the bit representation in AI algorithms to run models on smaller devices. This reduction in precision can significantly decrease the model's memory footprint and computational requirements, which is essential for deployment on devices with limited resources, such as mobile phones, embedded systems, or IoT devices.

Quantization typically involves converting a model from using 32-bit decimals numbers to 16-bit (or even 8-bit) integers.



02



Core Data Structures

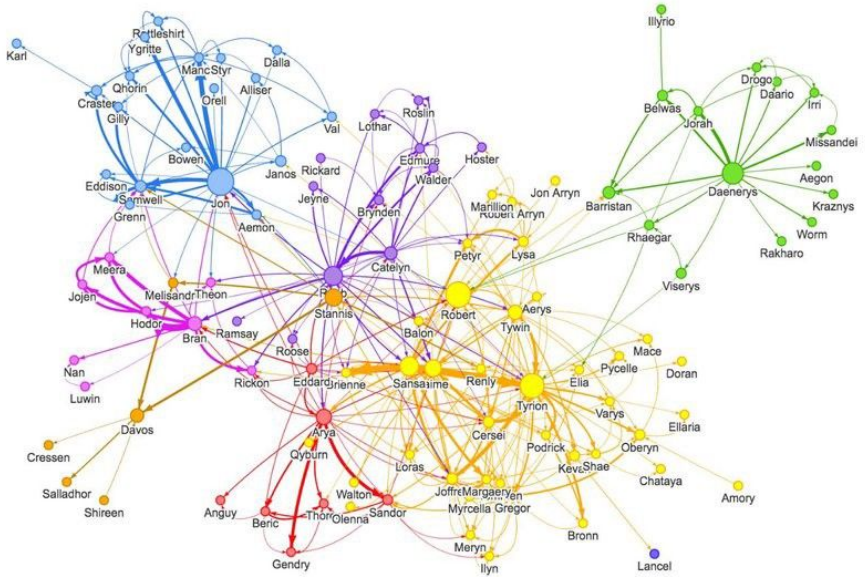
Introduction to Algorithms & Data Structures

Definition

Data structures are ways of organizing and storing data so that it can be used efficiently by algorithms. Different types of data structures are suited to different kinds of applications, and the choice of a particular data structure can affect the efficiency of an algorithm. In simple terms, data structures are like tools for handling, accessing, and storing the data that algorithms work with. An algorithm uses these data structures to manipulate the data, make decisions, and produce results.

Remember about the cooking recipe analogy ?

Just as different cooking tools are better suited for certain tasks (like using a frying pan for sautéing or a pot for boiling), different data structures are more effective for different types of data processing and manipulation in algorithms.



Primitive Data Structures

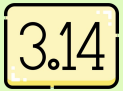
Primitive data types are the simplest forms of representing data that are directly operated on by the machine instructions. They are the building blocks for all other data structures.

Integers

Represents whole numbers without any decimal points.

Floating-Point Numbers

Numerical values that include decimals.



Characters

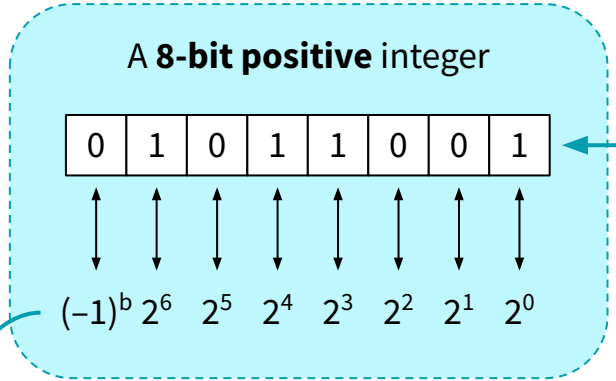
A sequence of characters, essentially text.

Booleans

They can hold only two values: True or False.

Integers

At the core, computers operate using binary, a **base-2 numeral system**. Each binary bit is a power of two. When you look at a series of bits, such as 1101, you're seeing a binary number that can be converted to a decimal (base-10) number that we use in everyday life.



The first bit is the sign bit:

- 0 indicates a positive integer
- 1 indicates a negative integer

Each bit value (0 or 1) can be converted in a power of two, leading to a base-10 value.

$(-1)^0 \times (\mathbf{1}\times\mathbf{2^6} + 0\times2^5 + \mathbf{1}\times\mathbf{2^4} + \mathbf{1}\times\mathbf{2^3} + 0\times2^2 + 0\times2^1 + \mathbf{1}\times\mathbf{2^0})$
$= 2^6 + 2^4 + 2^3 + 2^0$
$= 89$ *$(8\times10^1 + 9\times10^0)$*

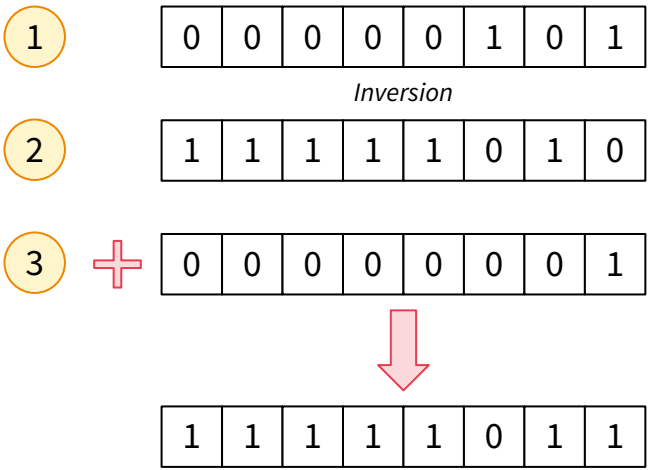
However, this method has two representations for zero (positive zero and negative zero), which is inefficient for calculations.

Integers – Two's Complement Method

The two's complement method is the most common method used in computers for binary arithmetic. This method has only one zero and is well-suited for arithmetic operations.

Two's Complement method for **8-bit negative** integers

example: **-5**



First, write the integer in a "regular" binary positive format.

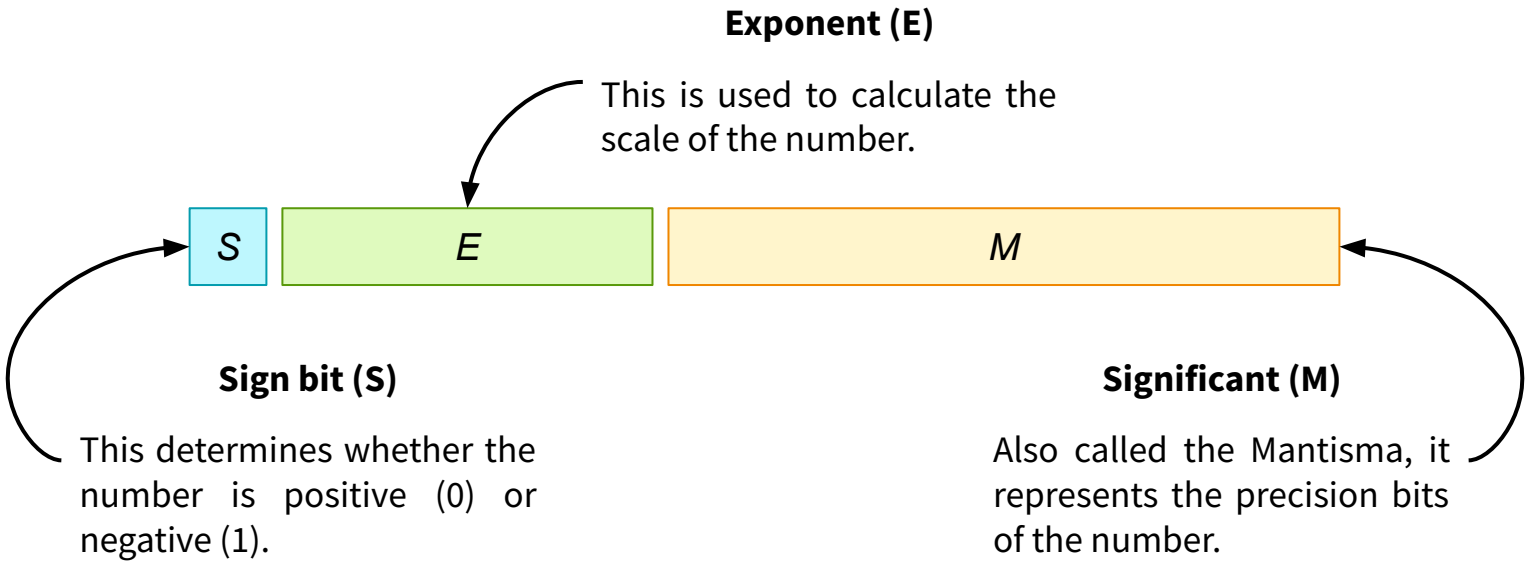
Then, invert all the bits of the integer (it's called one's complement).

Finally, add one to the resulting number.

This is the 8-bit representation of -5 using the two's complement.

Floating-Point Numbers

The **IEEE Standard for Floating-Point Arithmetic** (IEEE 754) is the most widely used standard for floating-point computation, and it defines the format for 16-bit, 32-bit, 64-bit, and other sizes. In a standard floating-point representation like IEEE 754, a number is broken down into three parts:

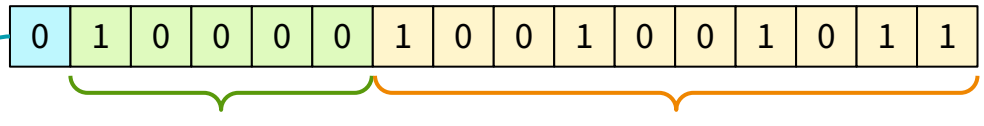


Floating-Point Numbers – 16-bit

The value represented by a floating-point number in the IEEE 754 format can be written as:

$$S \times 1.F \times r^{E - bias}$$

In the aforementioned formula, **S** is the sign, **1.F** is the significand (the leading 1 is referred to as the implicit bit and is set to 0 when the Exponent field is all zeros), **r** is the radix and depends on the base used (r = 2 in the case of the base-2 numeral system), **E** is the exponent and **bias** is a value that depends on the bit representation (15 for 16-bit, 127 for 32-bit, 1023 for 64-bit …).



Sign $(-1)^{0} = 1$

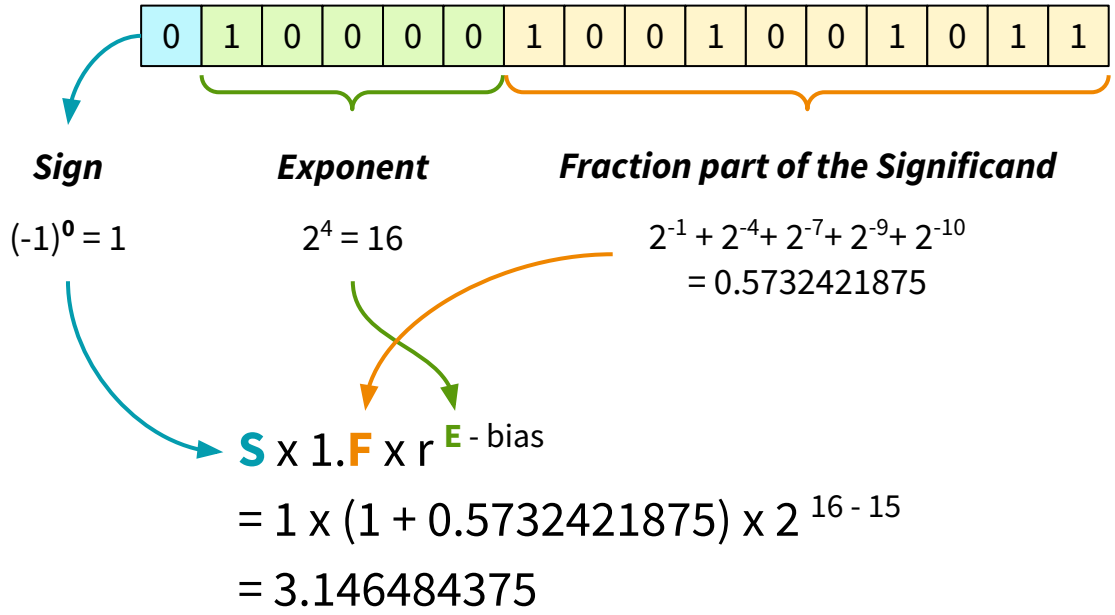
Exponent $2^{4} = 16$

Fraction part of the Significand $2^{-1} + 2^{-4} + 2^{-7} + 2^{-9} + 2^{-10} = 0.5732421875$

Each bit in the fraction part represents a negative power of 2. The first bit after the binary point represents 2^{-1}, the second represents 2^{-2} and so on.

$$\sum_{i=-23}^{-1} bit_i * 2^i$$

Floating-Point Numbers – 16-bit



Strings

Each character in a string is represented by a specific code using a character encoding scheme. The most common encoding schemes are **ASCII** (American Standard Code for Information Interchange) and **Unicode** which encompasses various encodings like UTF-8.

The **ASCII** encoding uses 7 bits to represent each character, which allows for 128 unique symbols. In ASCII, the letter 'A' is represented by the binary value 100 0001 and the letter ‘a’ by 110 0001.

UTF-8 use up to 4 bytes but is backward compatible with ASCII for the first 128 characters (the first byte).

ASCII TABLE

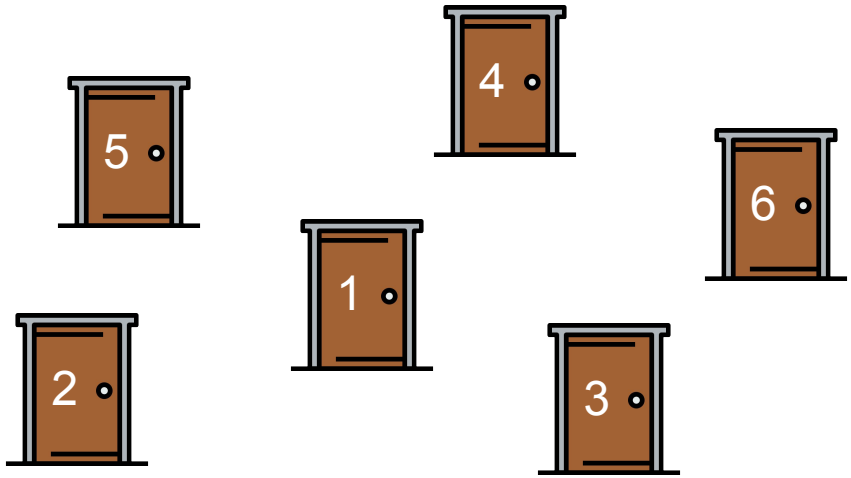
Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[END OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

Storing Primitive Data Structures

In computer programming, primitive data structures such as integers, floats, and strings are typically stored in memory without any specific organization.

Unlike complex data structures that have defined ways of organizing their elements, these primitives are allocated memory spaces based on availability and efficiency. As a result, their storage locations in memory can appear 'random' or unstructured, without following any discernible pattern.

This lack of organization is a characteristic feature of how these basic data types are handled at a low level in computer systems.



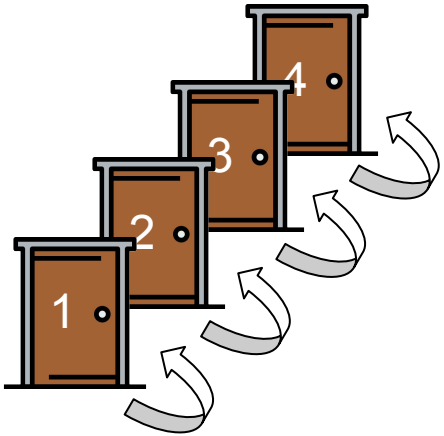
*When the computer needs to store or access data, it opens a door based on where there's space and efficiency considerations, much like choosing any available room in a hotel. There is **no inherent link** between the doors that the computer chooses. Each door (memory location) is opened individually, without reference to the others.*

Linear Data Structures

Linear data structures are a way to store data where elements are arranged in a straight line, one after another. Each element is connected to the next and possibly the previous one, in a specific sequence. In these structures, you add, remove, or access elements in a predictable, orderly manner.

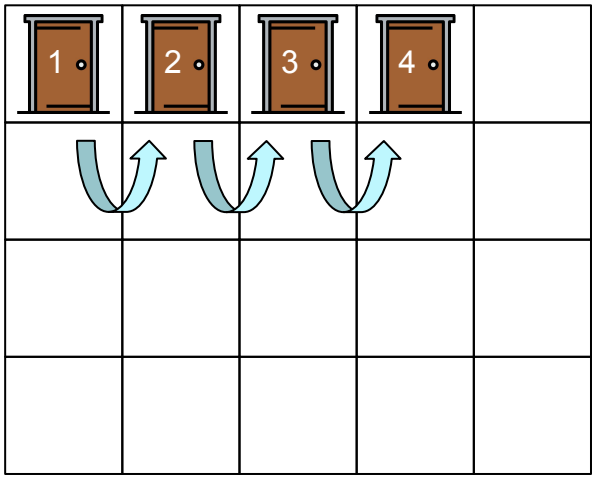
There are two main linear data structures: **Arrays** and **Linked Lists**.

Linear data structures can be thought of as a series of connected rooms, each with a door leading directly to the next. In this scenario, when the computer stores elements in a linear data structure, it's akin to placing them in a sequence of adjoining rooms. Each door in this sequence is directly linked to the next, creating a clear and orderly path through the rooms.



Linear Data Structures – Array

An **array** is a structured collection of elements that are of the **same data type** and **stored in contiguous memory locations**. The size of the array is determined at the time of creation and cannot be altered.



“Memory Locations”

In Python, there isn't a built-in array as you might find in languages like C or Java.

Lists and tuples can be used as replacement but they don’t have all the characteristics of a real array.

However, Python does provide an array-like data structure through its **array module** or the **NumPy library**. These arrays are **more efficient than lists** for storing large amounts of data of the same type.

Array – The array Module (Documentation)

Type Code	Type Python	Range/Size
'b'	int	[−127 ; +127]
'B'	int	[0 ; +255]
'u'	Unicode	2 octets
'h', 'i'	int	[−32 767 ; +32 767]
'H', 'I'	int	[0 ; +65 535]
'l'	int	[−2 147 483 647 ; +2 147 483 647]
'L'	int	[0 ; +4 294 967 295]
'q'	int	8 octets (signed)
'Q'	int	8 octets (unsigned)
'f'	float	4 octets (32-bit float)
'd'	float	8 octets (64-bit float)

The type code table

```python
import array

arr_int = array.array('i', [1, 2, 3, 4, 5])
arr_float = array.array('f', [1.1, 2.2, 3.3, 4.4])

arr_int.append(6)
arr_int.extend([7, 8, 9])

last = arr_int.pop()
arr_int.remove(1)

print(arr_int)
```

Type code

```
array('i', [2, 3, 4, 5, 6, 7, 8])
```

Python array module doesn't strictly adhere to the traditional fixed-size nature of arrays. Thus, you can add or remove elements.

Array – The NumPy Module (Documentation)

In NumPy, you can also add or remove elements from an array, but **NumPy arrays are still in line with the traditional definition of arrays in programming**.

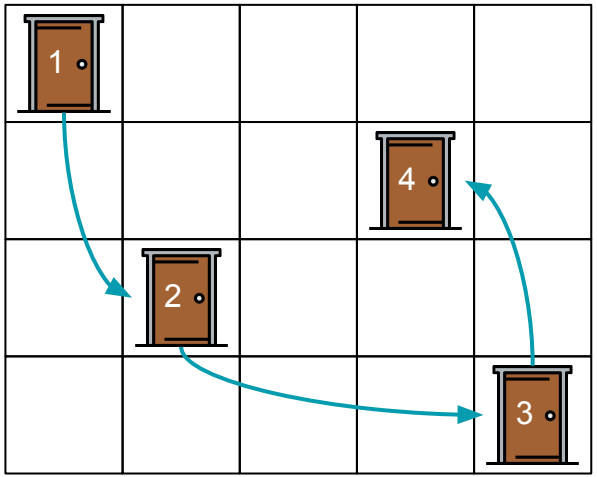
Indeed, any modification on a NumPy array do not modify the original array. Instead, a new array is created with the modification and replace the previous one.

```python
import numpy as np

arr_1D = np.array([1, 2, 3])
arr_2D = np.array([[1, 2, 3], [4, 5, 6]])

print(arr_2D)
print(f"Shape: {arr_2D.shape} / Size: {arr_2D.size}")
print(f"Type: {arr_2D.dtype}")
```

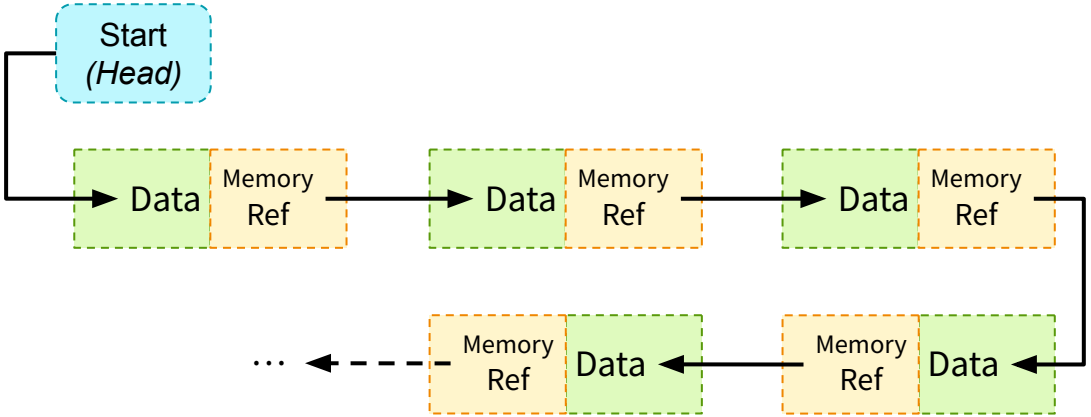
```
[[1 2 3]
 [4 5 6]]
Shape: (2, 3) / Size: 6
Type: int32
```

Linear Data Structures – Linked Lists

A **linked list** is a dynamic collection of elements, known as nodes, where each node contains two parts: the data and a reference to the next node in the sequence. This structure allows for flexible sizes without the need for a continuous block of memory. Instead, its elements are stored in arbitrary locations throughout the memory. Below is a Singly Linked List that allows traversal in one direction only – from the head to the end.



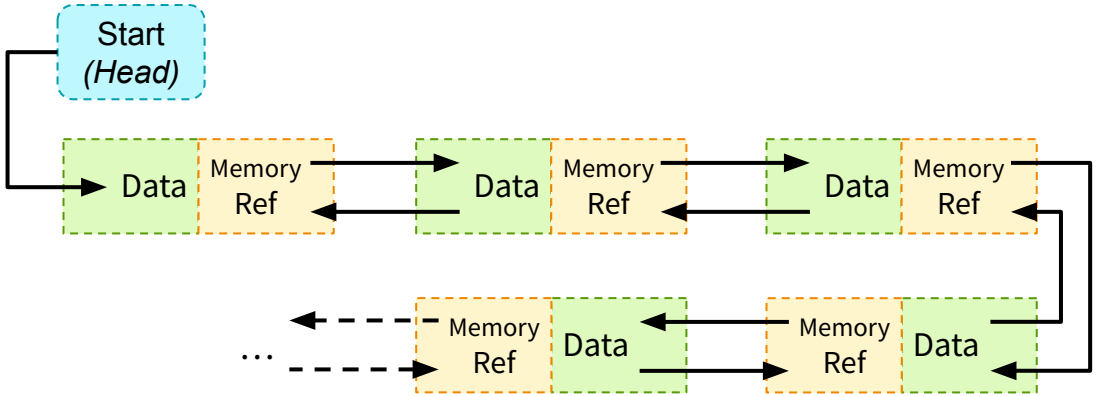
"Memory Locations"



Linked Lists can be created using Python OOP. An example can be found on this website.

Doubly Linked Lists

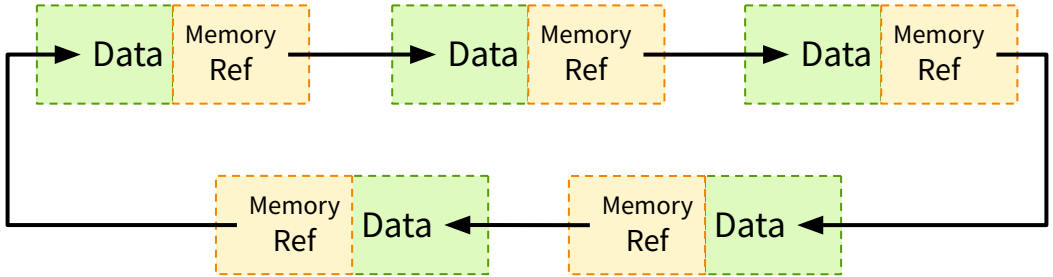
Doubly Linked List nodes contain data, a pointer to the next node, and a pointer to the previous node. It can be traversed in both directions – forward and backward.



Doubly Linked Lists can be created using Python OOP. An example can be found on this website.

Circular Linked Lists

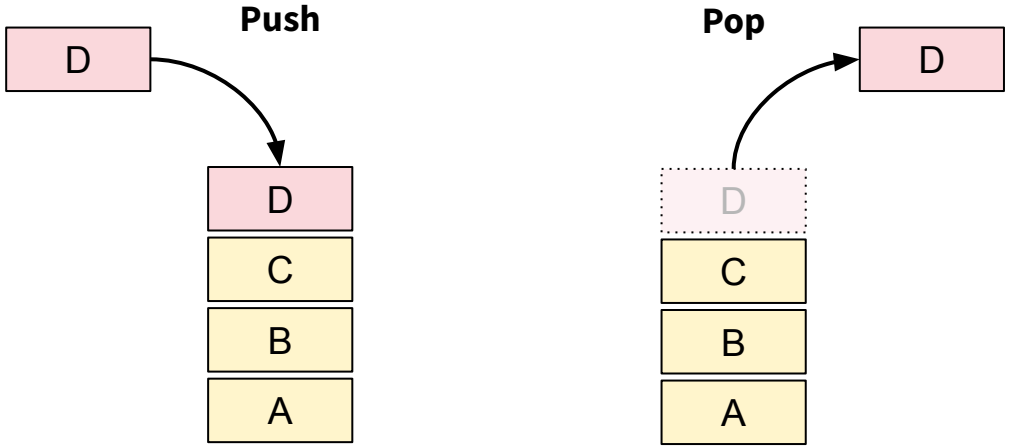
Circular Linked List are similar to singly linked list, but the last node points back to the first node, forming a circle. They can be looped through continuously without needing to find the end. Circular Linked List can also be doubly which which combines the features of both doubly linked lists and circular linked lists. The last node points to the first node and vice versa, and it allows continuous traversal in both directions.



Circular Linked Lists can be created using Python OOP. An example can be found on this website.

Linear Data Structures – Stacks

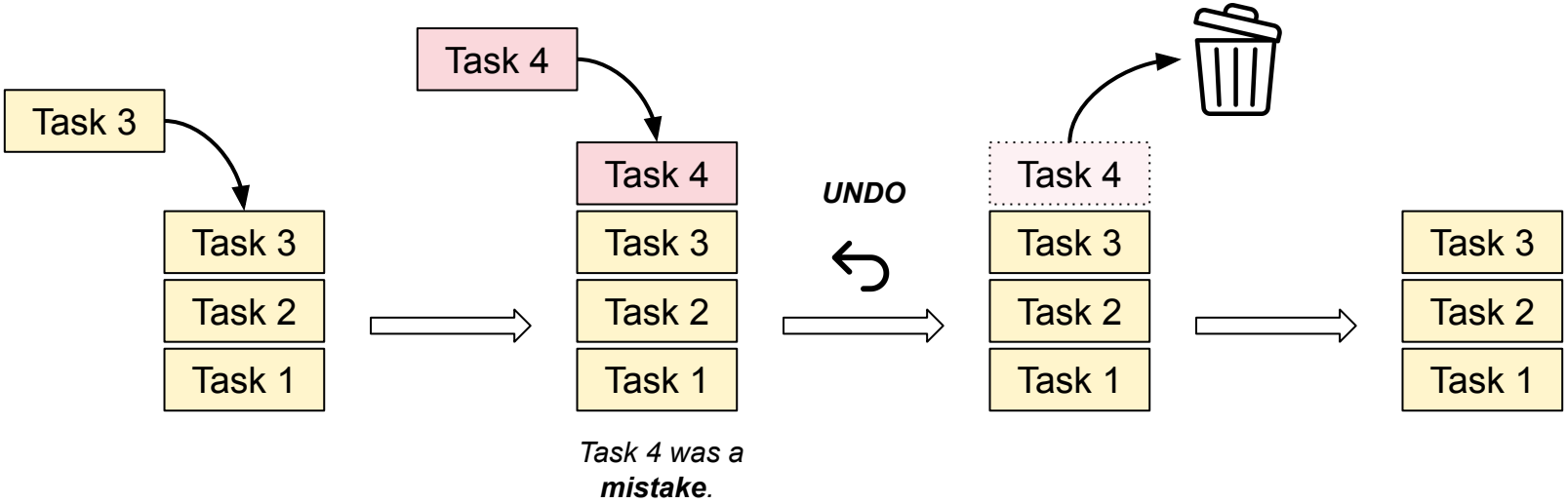
A **stack** is a collection that follows the **Last In, First Out (LIFO)** principle: The last element added to the stack is the first one to be removed. The three primary operations are **push** (add an item), **pop** (remove and return the top item), and **peek** (return the top item without removing it).



Imagine a stack of plates. When you add a new plate, you place it on top of the stack. When you need a plate, you take the top one off. This is the Last In, First Out principle. The last plate you put on the stack is the first one you take off.

Stacks – Use Case

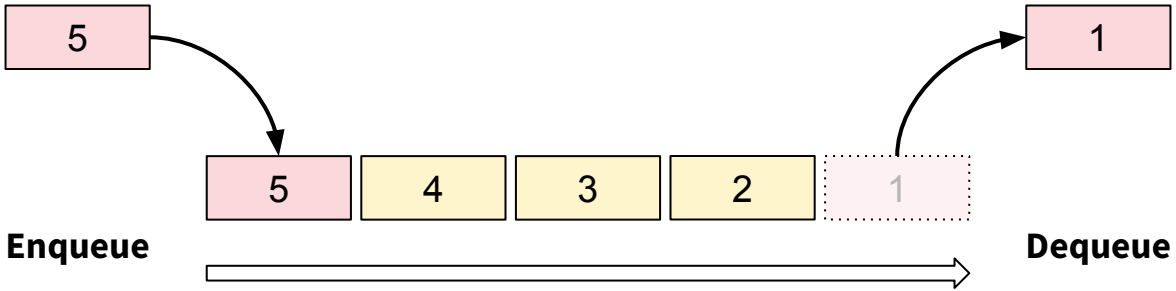
Many software applications use a stack to implement the **undo feature**. Each action is pushed onto the stack, and when the user undoes an action, the application pops it off the stack.



Other main applications of Stacks exist such as **Web Browser History** to keep track of visited pages, **Parsing and Syntax Checking** of programming languages to ensure that syntax elements like brackets and parentheses are properly balanced.

Linear Data Structures – Queues

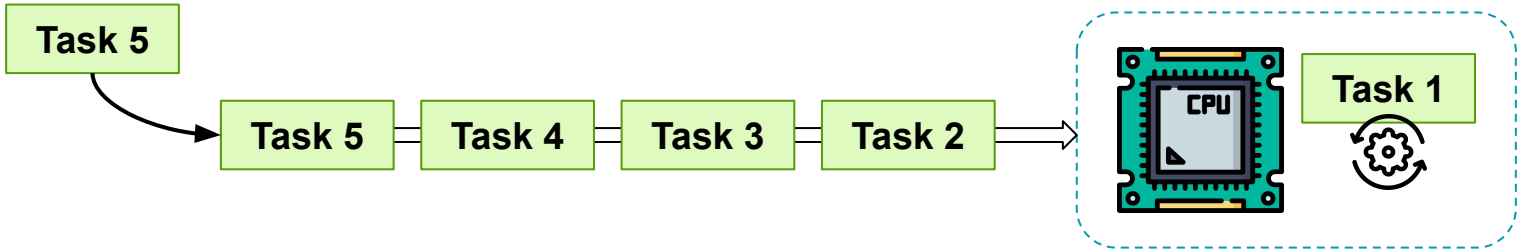
A **queue** is a collection that follows the **First In, First Out (FIFO)** principle: The first element added is the first one to be removed. The three main operations are **enqueue** (add an item to the end), **dequeue** (remove and return the first item), and **peek** (return the first item without removing it).



Think of a queue as a line of people at a grocery store checkout. The first person to get in line is the first person to be served and leave the line. As new people arrive, they join the end of the line. This is the First In, First Out principle.

Queues – Use Case

Queues are used in operating systems to manage tasks. Processes are lined up in a queue, and the CPU schedules them based on their order in the queue, ensuring a fair and efficient distribution of CPU time.



Other main applications of Stacks exist such as the **Handling of Real-Time Systems** like traffic lights control or call center systems, **Network Buffering** to hold the packets temporarily until the network device is ready to process them …

Queues & Stacks – Python's Deque (Documentation)

Queues and Stacks are generally implemented using a Linked List. However, the **deque** (double-ended queue) in Python's **collections** module is also an excellent choice for implementing both stacks and queues because of its efficient operations at both ends of the structure.

Stack

```python
from collections import deque

stack = deque()

# Pushing elements onto the stack
stack.append('A')
stack.append('B')
stack.append('C')

# Popping elements from the stack
print(stack.pop())
print(stack.pop())

print(stack)
```

deque(['A'])

Queue

```python
from collections import deque

queue = deque()

# Pushing elements onto the stack
queue.appendleft('A')
queue.appendleft('B')
queue.appendleft('C')

# Popping elements from the stack
print(queue.pop())
print(queue.pop())

print(queue)
```

deque(['C'])

Non-Linear Data Structures

Non-linear data structures are types of data organization where data elements are not arranged in a sequential order. In these structures, elements are connected in a more complex way than a simple linear sequence.

Linear Data Structures	Non-Linear Data Structures
Elements are arranged in a sequential order. Every element except the first and last has a unique predecessor and successor.	Elements are arranged in a hierarchical or interconnected manner.
Simple and easy to understand and implement. Efficient in accessing elements in a sequential manner.	Better organization of data for complex relationships. Efficient in representing hierarchical or networked data.
Limited flexibility in terms of insertion and deletion (especially in arrays). Poor utilization of memory and processing in some cases (like linked lists)	More complex to understand and implement. Operations like searching and traversing can be more time-consuming.

Trees

Trees are a way of organizing data that involves a collection of nodes arranged in a hierarchical structure.

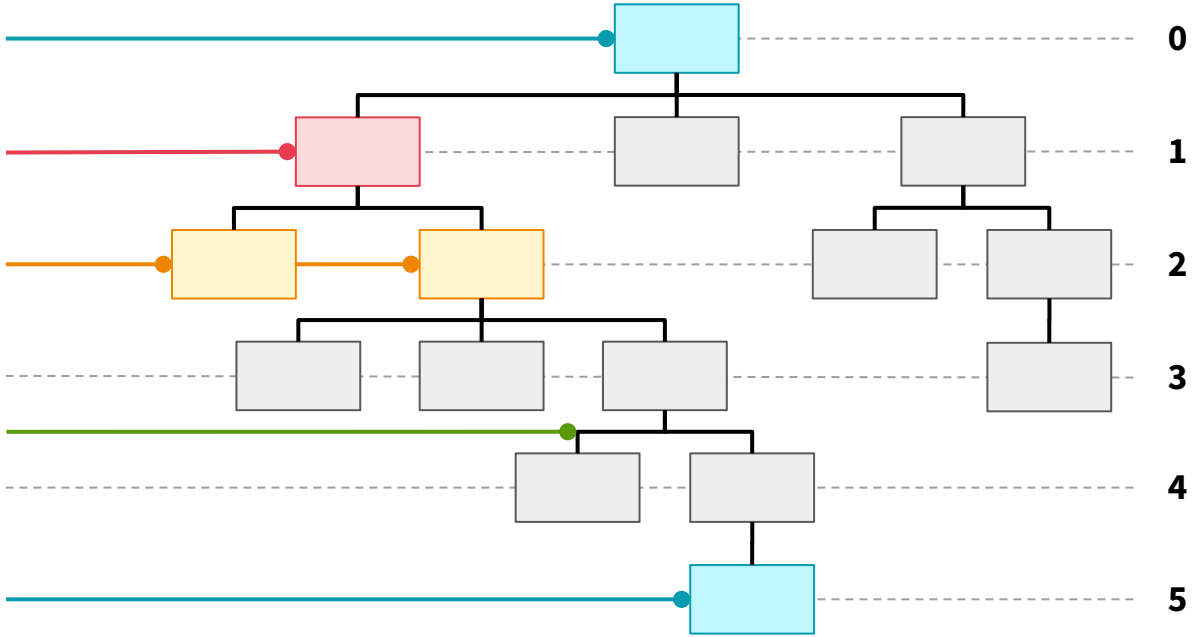
Root Node: This is the topmost node and does not have a parent.

Parent Node: Every node has one and only one parent node.

Child Nodes: These nodes are the children of the parent node (siblings).

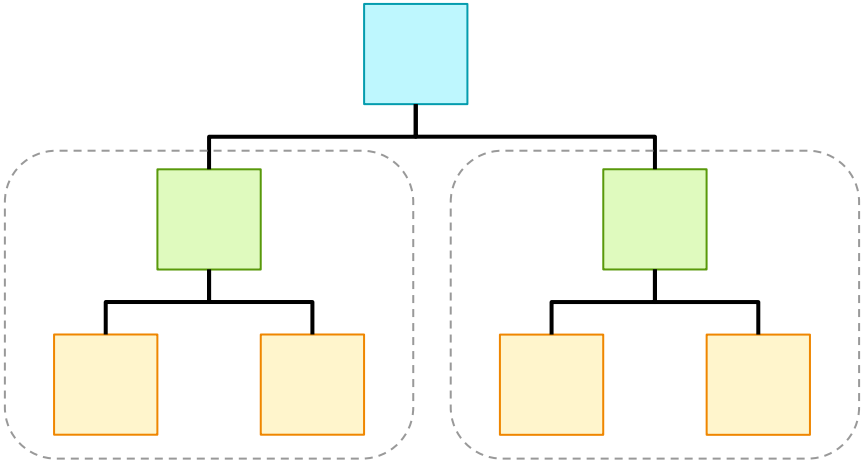
Edges: Connections between nodes are called edges.

Leaf Node: Nodes without children are called leaves or leaf nodes.



Basic Tree Types

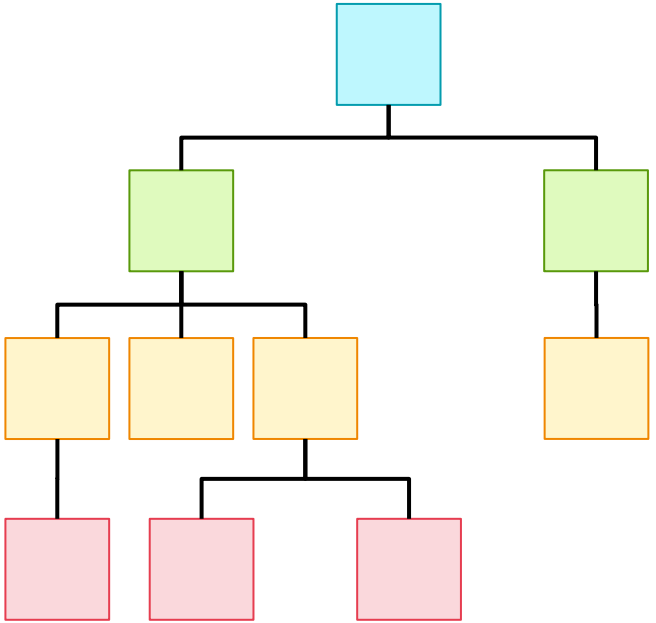
Binary Tree



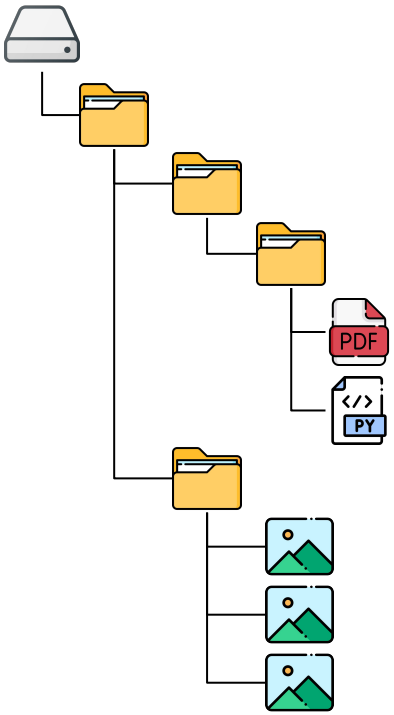
Left Subtree
(of the root node)

Right Subtree
(of the root node)

Generic Tree



Trees - Use Cases



Trees are widely used to organize file systems on computers. In file systems, trees are used to represent and manage the hierarchy of files and directories:

- The **root** of the tree represents the base level of the file system, like the drive or main folder.
- Each **node** in the tree represents a file or a directory.
- The **edges** denote the relationship between directories (**parent nodes**) and their contents (**child nodes**), which can be files or subdirectories.

The hierarchical nature of trees makes them ideal for this purpose, as they efficiently mirror the way directories and subdirectories are organized and nested within each other.

Other main applications of Tree Data Structures exist such as **Web Page Rendering** to represent and manipulate the structure of web pages (HTML), **Database Indexing** to make data retrieval faster, **Machine Learning** to predict based on data inputs, and so on.

Trees – Python

Trees are generally implemented using **classes** (see Implementation) but they can also be implemented using a dictionary or the **anytree library** (Documentation).

Basic Dictionary Tree Implementation

```
tree = {
    'root': {
        'child1': {
            'grandchild1': {},
            'grandchild2': {},
        },
        'child2': {
            'grandchild3': {}
        }
    }
}
```

Tree Implementation using anytree

```
from anytree import Node

root = Node("root")

child1 = Node("child1", parent=root)
gchild1_1 = Node("grandchild1", parent=child1)
gchild1_2 = Node("grandchild2", parent=child1)

child2 = Node("child2", parent=root)
gchild2_1 = Node("grandchild3", parent=child2)
```

Exercises

Draw the following trees.

Tree n°1

1. The root node have a value of 'A'.
2. The second level should consist of two nodes: 'B' (left child of 'A') and 'C' (right child of 'A').
3. The third level should have four nodes: 'D' and 'E' as the children of 'B', and 'F' and 'G' as the children of 'C'.

Tree n°2

In a **binary search tree**, all values in the left subtree of a node must be less than the node's value, and all values in the right subtree must be greater.

1. Draw a binary search tree with the following values: 40, 20, 60, 10, 30, 50, 70.
2. Start with 40 as the root.

Tree n°3

In a **preorder traversal**, the root node is visited first, followed by the left subtree and then the right subtree. In an **inorder traversal**, the left subtree is visited first, then the root, and finally the right subtree.

Draw the tree, given the following traversal sequences:

a. Preorder traversal sequence:
   T, R, A, V, E, R, S, A, L
b. Inorder traversal sequence:
   A, R, V, T, R, E, A, S, L

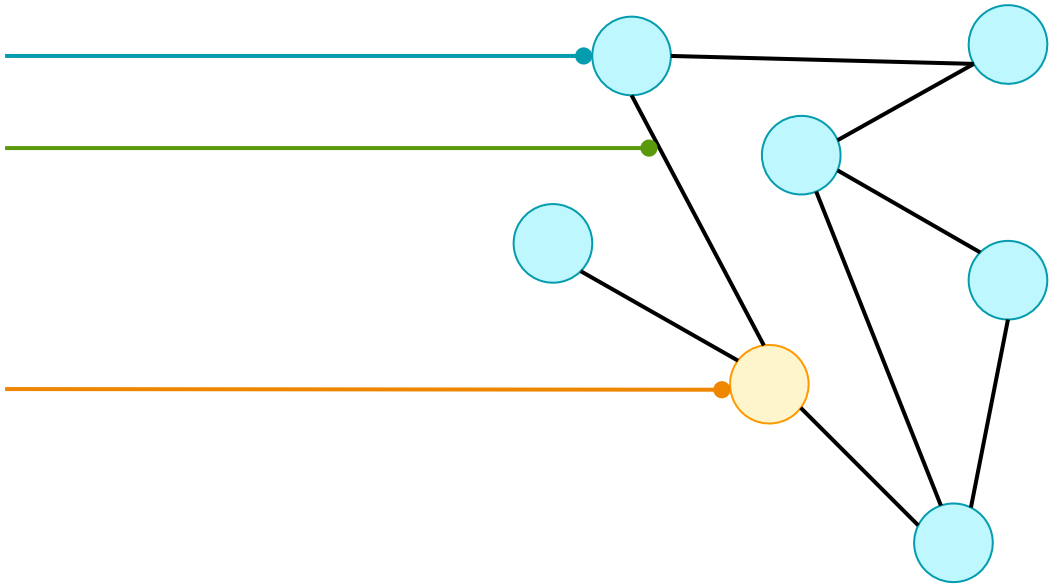
Graphs

Graphs are a collection of points called vertices or nodes connected by lines known as edges.

Vertex: Represents an entity or a point in the graph.

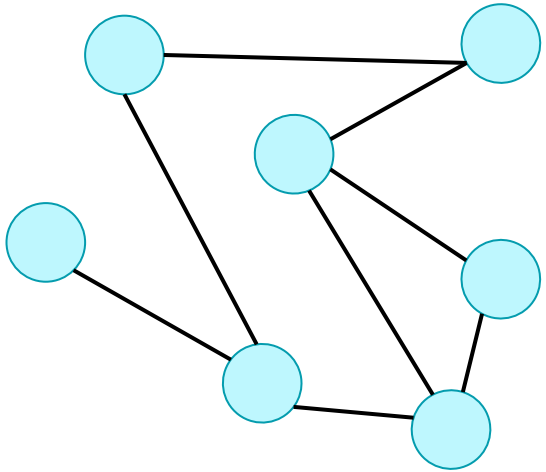
Edge: Connects two vertices.

The **degree** of a vertex is the number of edges connected to it. Here, the degree is 3.



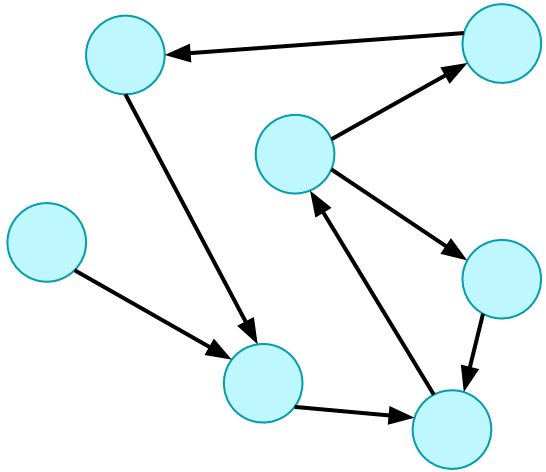
Graph Directionality

Undirected



A graph in which edges have no direction. This means the edges do not have arrows, and the connection between any two nodes is bidirectional.

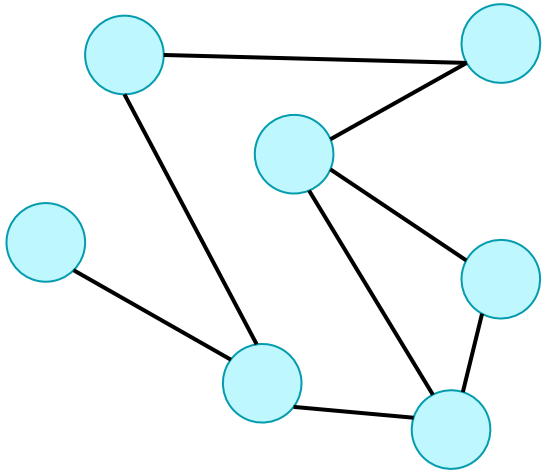
Directed *(Digraph)*



A graph in which all the edges have a specific direction. This is indicated by arrows on the edges, showing the direction of relationship from one node to another.

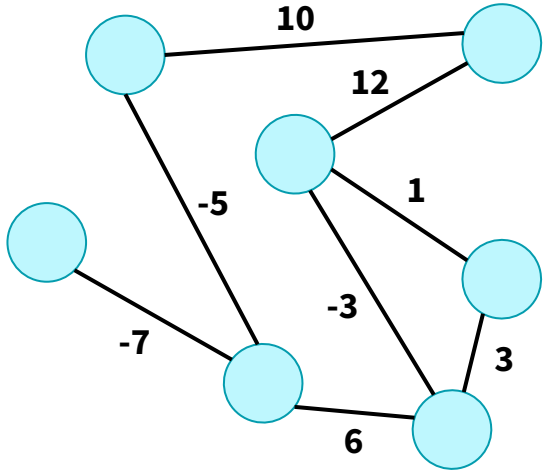
Edge Characteristics

Unweighted



A graph where the edges do not have any weights associated with them. Each edge is equal and has no specific value or label that distinguishes it from others in terms of weight.

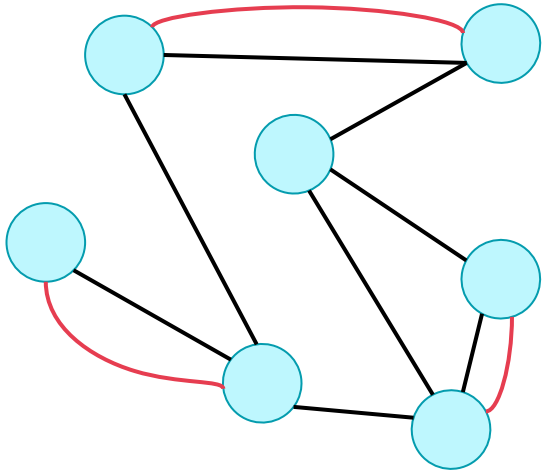
Weighted



A graph in which edges have assigned weights or costs. These weights can represent distances, costs, capacities ...

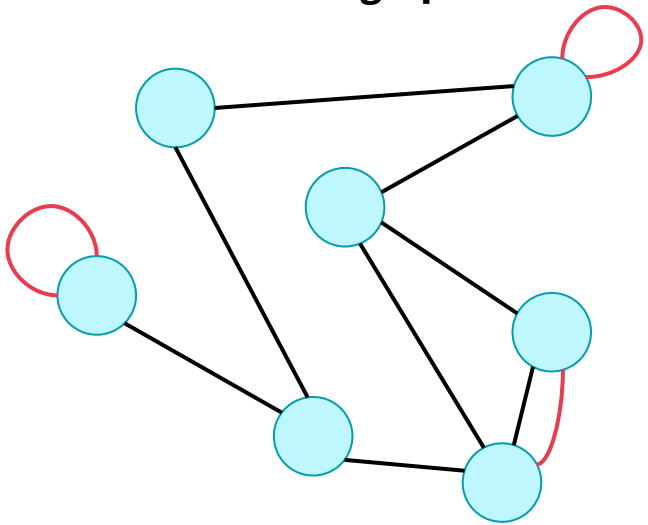
Multiple Edges

Multigraph



A graph that allows for multiple edges between the same pair of nodes. However, it does not permit self-loops.

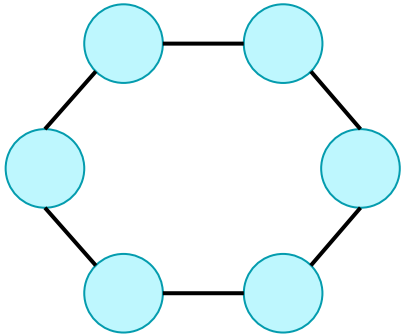
Pseudograph



A graph that allows both multiple edges between the same pair of nodes and self-loops.

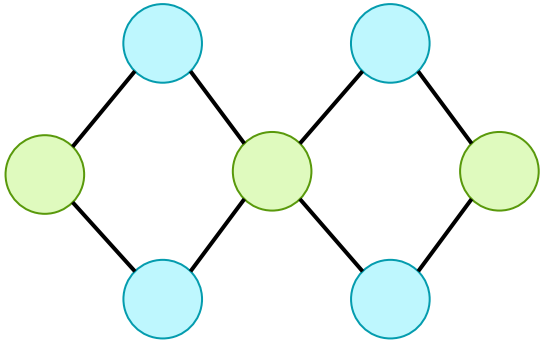
Some Special Graphs

Cyclic Graph



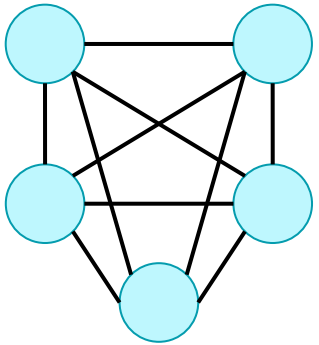
Contains at least one graph cycle.

Bipartite Graph



The set of nodes can be divided into two disjoint sets, such that every edge in the graph connects a vertex from one set to a vertex from the other set.

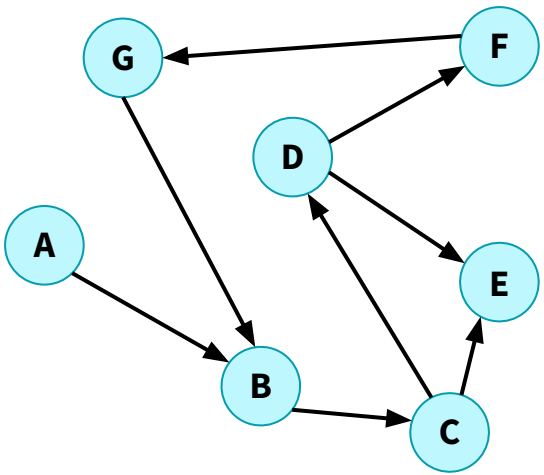
Complete Graph



Every pair of distinct nodes is connected by a unique edge.

Adjacency Matrix

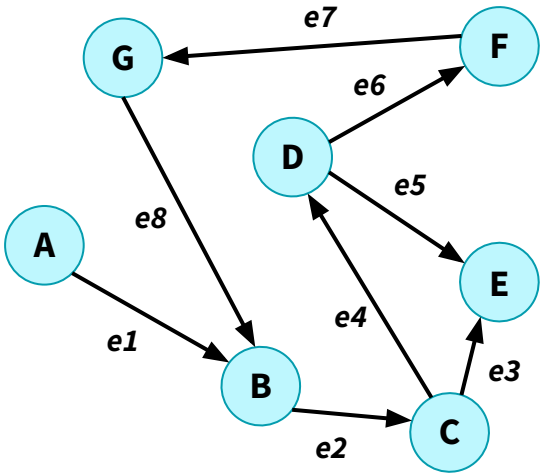
An **adjacency matrix** is a square matrix used to represent a graph. The elements of the matrix indicate whether pairs of nodes are adjacent or not in the graph. In this matrix, each row and column represent a node in the graph.



	A	B	C	D	E	F	G
A	0	**1**	0	0	0	0	0
B	0	0	**1**	0	0	0	0
C	0	0	0	**1**	**1**	0	0
D	0	0	0	0	**1**	**1**	0
E	0	0	0	0	0	0	0
F	0	0	0	0	0	0	**1**
G	0	**1**	0	0	0	0	0

Incidence & Path Matrices

An **incidence matrix** is a matrix used to represent the edges of a graph while a **path matrix** is a way of representing the connectivity and paths in a graph. Incidence matrices are particularly useful when dealing with edge-centric problems, while path matrices are beneficial for understanding the broader connectivity patterns in a graph.



Path Matrix

	A	B	C	D	E	F	G
A	0	1	2	3	3	4	5
B	inf	0	1	2	2	3	4
C	inf	4	0	1	1	2	3
D	inf	3	4	0	1	1	2
E	inf	inf	inf	inf	0	inf	inf
F	inf	2	3	4	4	0	1
G	inf	1	2	3	3	4	0

Inf indicates that there is no path from the nodes.

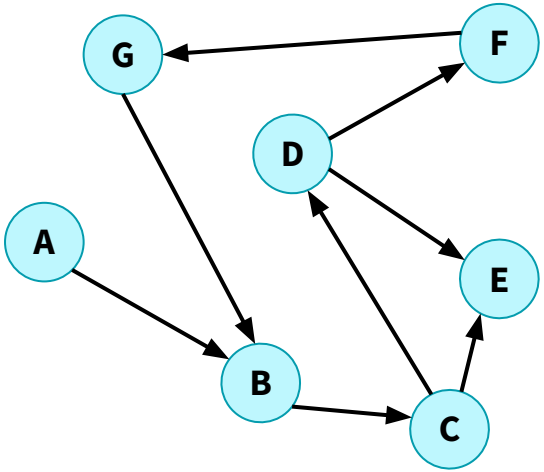
Incidence Matrix

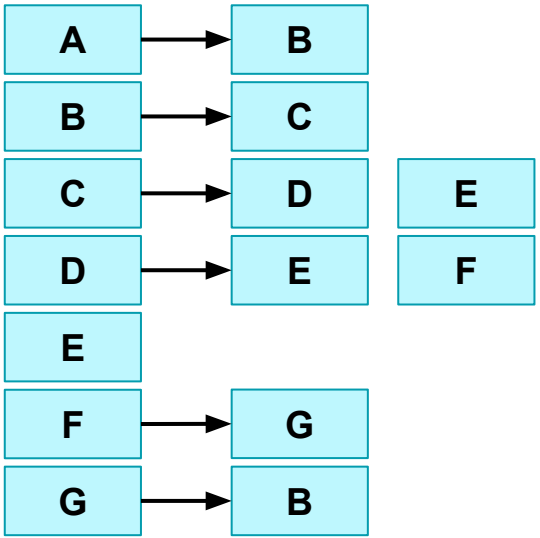
	e1	e2	e3	e4	e5	e6	e7	e8
A	**-1**	0	0	0	0	0	0	0
B	**1**	**-1**	0	0	0	0	0	**1**
C	0	**1**	**-1**	**-1**	0	0	0	0
D	0	0	0	**1**	**-1**	**-1**	0	0
E	0	0	**1**	0	**1**	0	0	0
F	0	0	0	0	0	**1**	**-1**	0
G	0	0	0	0	0	0	**1**	**-1**

-1 indicates the source node of an edge, ***1*** indicates the destination node.

Adjacency List

Similarly, an **adjacency list** is a collection where each node of a graph is stored as a key, and the list of nodes that are adjacent to it is stored as the value. This means for every node, you have a list of other nodes that it is connected to.





Adjacency lists are generally built using Linked Lists.

Graphs – Python

Graphs are generally implemented using three different methods in Python:

Graphs can be created using Python OOP. An example can be found on this website.

A Linked List or an Array

```python
import numpy as np
adj_matrix = np.array([[0, 1, 0, 1, 0],
                       [1, 0, 1, 0, 1],
                       [0, 0, 0, 0, 1],
                       [1, 0, 0, 1, 0],
                       [0, 1, 1, 0, 0]])

adj_list = {0: [1, 3],
            1: [0, 2, 4],
            2: [4],
            3: [0, 3],
            4: [1, 2]}
```

The networkx library
(Documentation)

```python
import networkx as nx

G = nx.Graph()   # undirected graph

G.add_node("A")
G.add_nodes_from(["B", "C", "D", "E"])
G.add_edge("A", 2)
G.add_edges_from([("B", 3), ("C", 4), ("D", 4)])
```

Adjacency List or Adjacency Matrix ?

An **adjacency matrix** is a suitable representation for dense graphs, where the number of edges is close to the maximum possible. It allows for quick, constant-time checks to determine if an edge exists between two vertices.

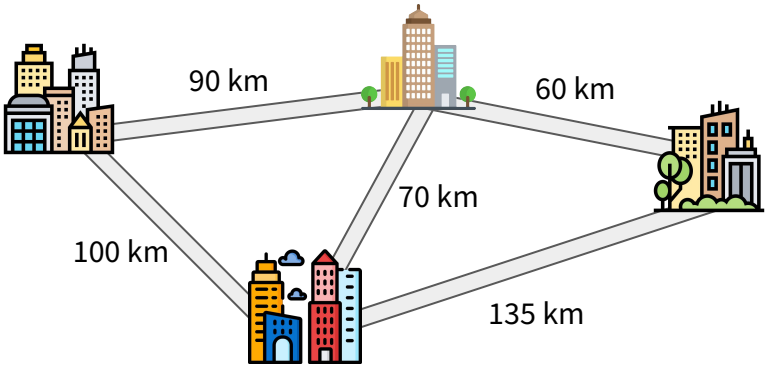
However, this method can be space-inefficient, particularly for sparse graphs with fewer edges, as it requires a space proportional to the square of the number of vertices. Additionally, for undirected graphs, it introduces redundancy due to its symmetrical nature, and iterating over edges can be less efficient.

An **adjacency list** is more space-efficient for sparse graphs, as the space used is proportional to the number of edges. The adjacency list is also more flexible, easily accommodating dynamic changes in the graph such as adding or removing vertices and edges.

However, it falls short in edge lookups, as finding whether an edge exists between two vertices can be slower, especially if a vertex has many neighbors.

Graphs – Use Case

The main application of graphs is arguably in routing and navigation systems, such as GPS and other map services. In these systems, locations are represented as nodes, and the roads or paths between them are the edges.



Other main applications of Graph Data Structures exist such as **Social Networks** to represent users their connections, **Internet Mapping** to represent websites and hyperlinks, **Supply Chains** to optimize routes …

Exercises

Construct the following Graph

1. Create a directed graph representing a small network of cities and their one-way flight routes.
2. Include the following cities: City A, B, C, D, and E.
3. City A has a flight route to City B with a travel time of 30 minutes.
4. City B has a flight route to City C with a travel time of 45 minutes.
5. City C has a flight route back to City A with a travel time of 50 minutes.
6. City B also has a flight to City D taking 40 minutes.
7. City D has two flight routes: one to City E (20 minutes) and a direct return flight to City B (35 minutes).
8. City E has a flight to City C that takes 25 minutes.
9. Lastly, City E also has a tourist flight of 15 minutes that goes around the city for the view.

Build the Adjacency matrix and list of this graph.

Build the Path matrix of the shortest path between Cities.

How did you proceed ?

How many steps did you do ?

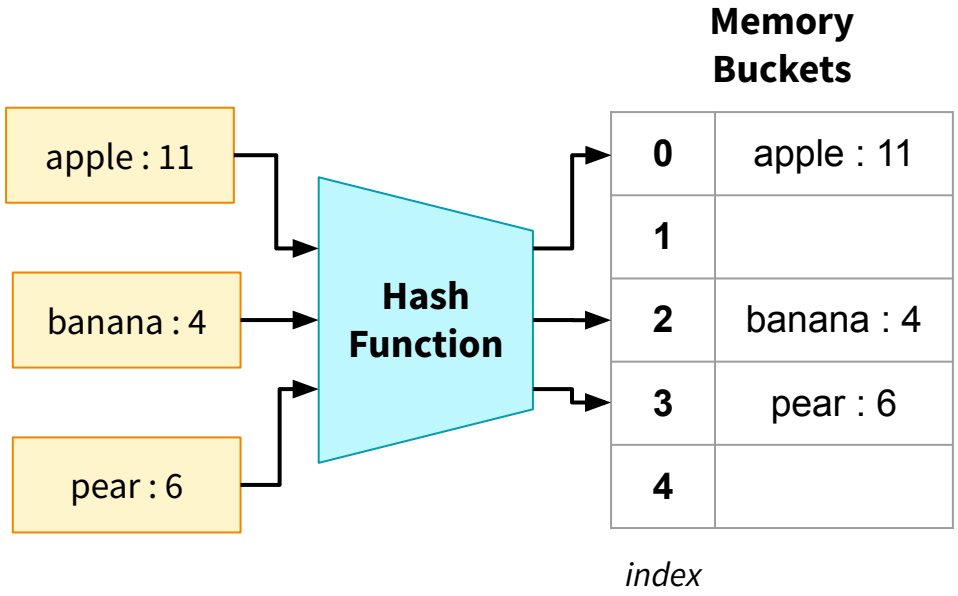
Hash-Tables

Hash tables are a data structure that store key-value pairs. They are designed to optimize the speed of data retrieval. The primary mechanism behind a hash table is the use of a **hash function**, which converts keys into array indices.

When a key-value pair is added to a hash table, the key is passed through a hash function.

This function computes an index based on the key, typically using methods to ensure that the computed indices are evenly distributed and that different keys are likely to produce different indices.

The **hash table consists of an array** where each element is a "bucket" that can hold **one or more key-value pairs**. The size of this array can affect the efficiency of the hash table.

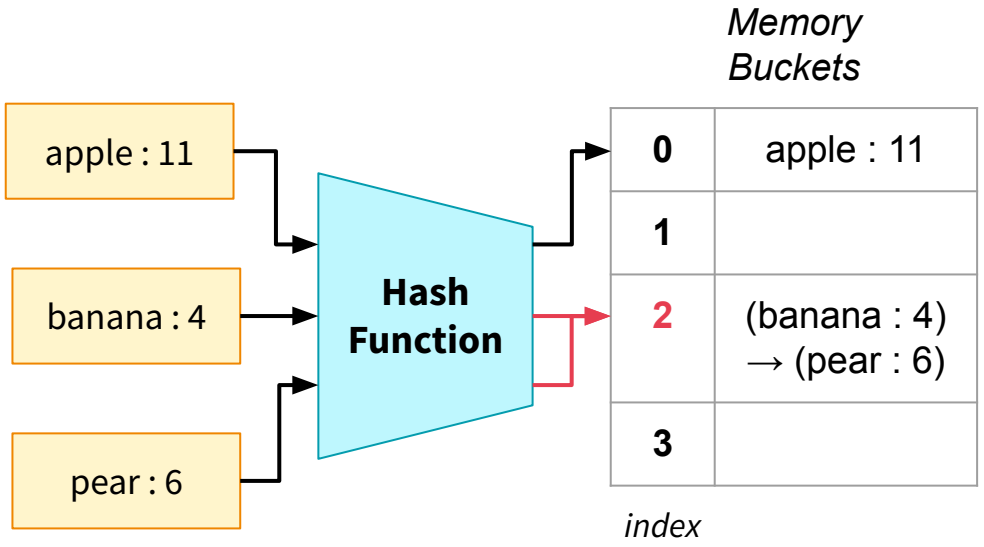


Hash-Tables – Collisions

Collisions in a hash table occur when two different keys are hashed to the same index. Indeed, a hash table has a finite number of buckets, determined by its size. However, the number of possible keys can be much larger than the number of buckets. This limitation means that at some point, different keys will inevitably hash to the same index.

A **good hash function aims to minimize collisions** by uniformly distributing keys across the available buckets. However, **collisions are a natural and expected part of using hash tables**. Thus, managing collisions efficiently is crucial. Two main methods are used to handle collisions:

- **Chaining** (each bucket is a linked list)
- **Open Addressing** (search for another free bucket) such as linear/quadratic probing or Double Hashing (a second hash function is used)



Handling a collision with a Chaining method

Hash-Tables – Python

Hash tables are generally implemented using dictionaries in Python. Indeed, Python's built-in dictionary is essentially a hash table. For specific applications, you can also create a custom hash table by defining a class (implementations [1], [2]). This approach allows you to implement custom hashing, collision resolution strategies, and other specific behaviors.

```
1 hash_table = {}
2 hash_table['key1'] = 'value1'
3 hash_table['key2'] = 'value2'
```

Prior to Python 3.7, dictionaries did not maintain insertion order. If maintaining order was important, OrderedDict from the collections module was used.

As of Python 3.7, the built-in dict type maintains insertion order by default, so OrderedDict is less commonly needed for this purpose.

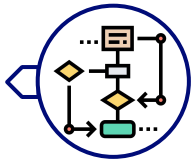
Dictionary keys must be hashable i.e. they must be immutable in order to be used in a hash function.

Exercises

For each of the following scenario, decide which Data Structure is the most appropriate and justify your choice.

- Storing Student Grades
- Keeping Track of Patients in a Doctor Waiting Room
- Implementing Undo Functionality in a Text Editor
- Organizing a Company's Organizational Structure
- Mapping Routes and Connections in a Public Transportation System
- Managing a Library Catalog System
- Implementing a Playlist for a Music Streaming Service
- Organizing Books in a Bookshelf Based on Categories
- Analyzing Social Networks
- Implementing a Username Lookup System in an Application
- Tracking the Order of Documents Printed by a Printer
- Building a Browser's History Feature

03



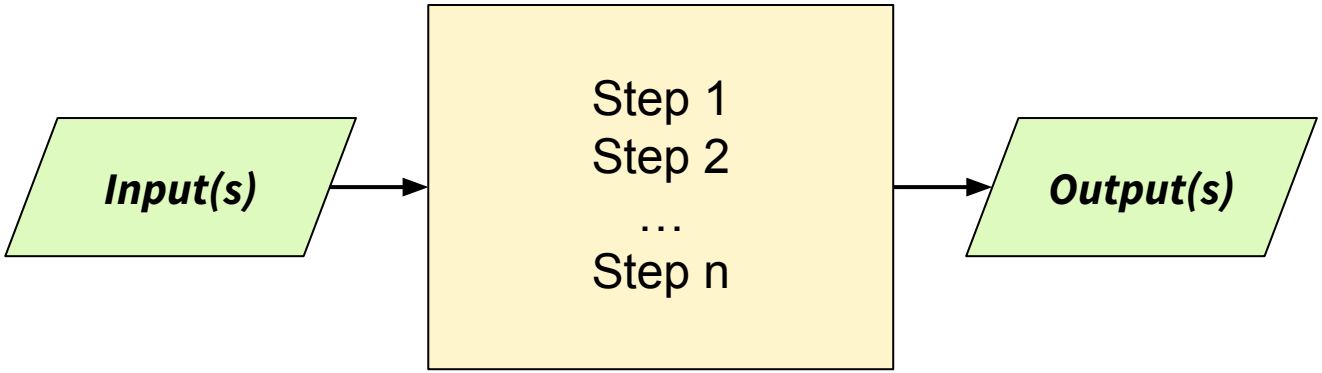
Algorithm basics

Introduction to Algorithms & Data Structures

Algorithm Definition

As a general definition, an **algorithm** can be considered as a recipe or a blueprint for solving a problem. It involves a series of steps that are carried out in a specified sequence to achieve a desired outcome or solution.

From a **computer science perspective**, an algorithm is a finite sequence of computational steps that transform the input into the output. It is a methodical and logical procedure that guarantees solving a particular problem when correctly applied.



Algorithm or Program ?

Algorithms and Programs are often used interchangeably in computer science, but they refer to different concepts.

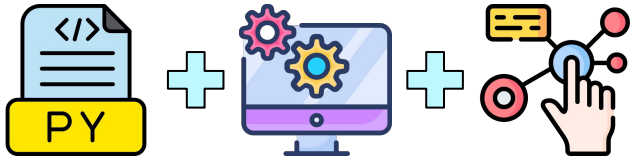
A **program** is a specific set of instructions written in a programming language that implements one or more algorithms to perform a specific task on a computer. It is a **concrete implementation of one or more algorithms** in a particular programming environment.

On the other hand, algorithms are typically language-agnostic and can be considered as “conceptual”. Thus, **algorithms** can be expressed in natural language, **pseudocode** or **flowcharts** and aren't bound to any specific programming language.

Algorithm

Idea *Recipe* *Adaptability*

Program



Code *Hardware Software* *Interactivity*

Building an Algorithm

Before starting to code an algorithm, the building process involves several **key steps**, each crucial for ensuring its effectiveness and efficiency:

Define the Problem

Clearly understand and define the problem you are trying to solve. This step is crucial as it guides the entire algorithm development process.

Research and Analysis

Conduct thorough research and analysis. This might include studying existing solutions, understanding the theoretical background, and analyzing similar problems and their solutions.

Design the Algorithm

This involves conceptualizing how the algorithm will work. It's often helpful to use **pseudocode** or **flowcharts** at this stage to outline the logic and structure of the algorithm.

Select the Right Data Structures

Based on the problem and the design, choose appropriate data structures. The efficiency of your algorithm heavily depends on this choice.

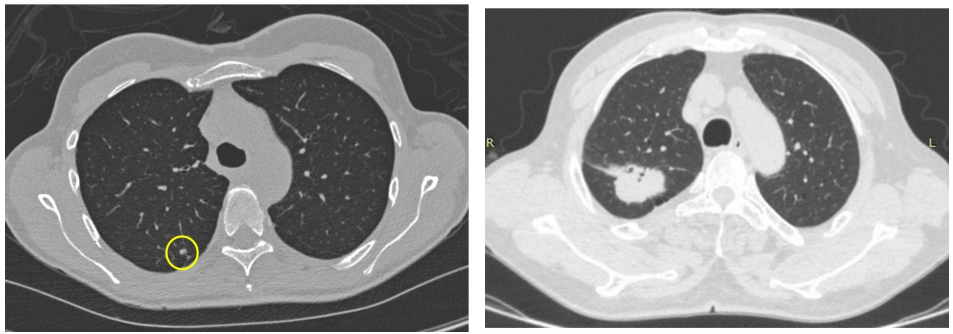
Define the Problem – Identify the Core Problem

Be as specific as possible about what the problem is and **avoid vague descriptions**. Moreover, if the problem is complex, **break it down into smaller**, more manageable parts.

Suppose you're working on an AI project related to healthcare, specifically for diagnosing diseases from medical images. You would start by studying medical imaging techniques, common diseases diagnosed through such images, and the role of AI in healthcare.

The **core problem might be identifying early signs of a specific disease**, like lung cancer, from CT scans. This requires precise identification of small, often easily missed anomalies in the images.

Source: itnonline.com



Source: journals.sagepub.com

Define the Problem – Clarify Objectives

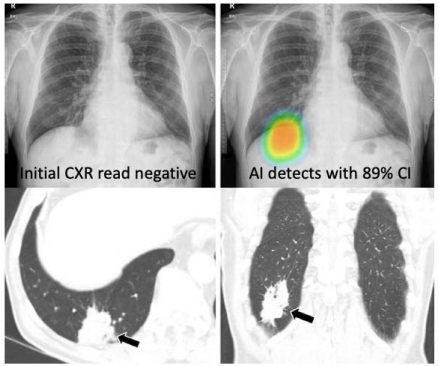
Define what you want to achieve with your algorithm and **be clear about the end goal** by establishing criteria for what will constitute a successful solution to the problem.

The objective may be to develop a machine learning model that can accurately and efficiently identify early-stage lung cancer from CT scans.

The model should achieve a certain metric percentage such as accuracy, and be able to process an image within a specific time frame.

It could be something like : “Develop a machine learning model that can identify early-stage lung cancer from CT scans with at least 80% AUC.”

Human-AI Interaction in Detecting Malignant Nodules on Chest Radiographs



- Retrospective study of 30 readers assessing 120 chest radiographs (CXRs with lung cancer, *n* = 60) with and without AI.
- Readers using a high accuracy AI had higher AUC (0.82 vs 0.77) and improvements (0.05 vs 0) than those using an inaccurate AI.
- High accuracy AI led to more changes in reader determinations (67%) between the two sessions than the inaccurate AI (59%).

Lee JH et al. Published Online: June 27, 2023
https://doi.org/10.1148/radiol.222976
Radiology

Source: pubs.rsna.org

Define the Problem – Define Constraints and Requirements

Identify any limitations such as **time**, **resources**, or **technological constraints** while listing the technical and functional requirements that your algorithm must meet.

For example, the model could have two major constraints:

- It must process each image within 30 seconds due to the high volume of scans in a typical hospital setting.
- It should integrate with existing medical imaging software and comply with healthcare data privacy regulations.

Knowing these constrains, the problem could be stated as the following:

“Develop a machine learning model that can identify early-stage lung cancer from CT scans with at least 80% AUC, *process each scan within 30 seconds*”

Research and Analysis

At that step, you should **conduct thorough research and analysis**. This might include studying existing solutions, understanding the theoretical background, and analyzing similar problems and their solutions.

Literature Review & Existing Solutions

Look for **academic papers**, **journal articles**, and **books** relevant to your problem. Websites like Google Scholar, IEEE Xplore, arXiv or PaperWithCode (AI-related papers) are valuable resources.

Identify and review existing algorithms. Open-source projects (GitHub), case studies, and documentation can be insightful.

If there's **no existing solutions**, **then look for problems that share similarities with yours**, even if they are in a different domain.

Data Analysis

If your problem involves data, **understanding the characteristics of your data is crucial**. You should conduct exploratory data analysis to understand your data's nature, quality, and peculiarities by identifying patterns, anomalies …

Design The Algorithm

In the design phase of algorithm development, two pivotal tools stand out for their utility to developers: **pseudocode** and **flowcharts**. Together, pseudocode and flowcharts form a powerful duo, enhancing a developer's ability to conceptualize, optimize, and communicate the intricacies of algorithm design efficiently and effectively.

Pseudocode

Pseudocode, a simplified form of programming language, serves as an **intermediary between plain language and code**, allowing developers to articulate their algorithm's logic in a clear, concise, and structured manner.

This tool is instrumental in breaking down complex processes into understandable steps, thereby **reducing the likelihood of errors when translating the algorithm into actual code**.

Flowcharts

Flowcharts, on the other hand, provide **a visual representation of the algorithm's flow**, offering a diagrammatic approach to understanding and communicating the sequence of actions and decision points.

Pseudocode

Pseudocode is a way to write down the steps of an algorithm using simple, easy-to-understand language. It is not actual computer code, but it looks somewhat like it. Pseudocode is used to plan how a computer program will work by outlining the logic and main actions without using the specific rules of a real programming language. This makes it easier for someone to understand the algorithm and later turn it into a real computer program.

English

Start by setting the total sum to zero. Then, you sequentially consider each integer, beginning from one and continuing up to n. As you count each number, you add it to the running total. Once you've counted up to n, you can display the final sum.

Pseudocode

```
SET sum TO 0
FOR each number i FROM 1 TO n
        ADD i TO sum
END FOR
DISPLAY sum
```

Code *(python)*

```python
sum = 0
for i in range(1, n+1):
    sum += i
print(sum)
```

Pseudocode Guidelines

Pseudocode, by its nature, doesn't have strict rules like programming languages, but it generally follows some common guidelines to maintain clarity and consistency. First, pseudocode should be **easy to read and understand**. It uses plain language and avoids overly complex terminology.

It often employs common **programming constructs** like **loops** (for, while), **conditionals** (if, else), etc.

```
SET sum TO 0
FOR each number i FROM 1 TO n
        ADD i TO sum
END FOR
DISPLAY sum
```

Pseudocode is written in the order in which the code should execute.

Similar to programming, pseudocode often uses **indentation** to represent the flow of an algorithm.

Variables in pseudocode are often given **descriptive names** to indicate their purpose.

Pseudocode – Initializing variables

```
SET number TO 10 AS an integer
SET number TO 3.14 AS a float
SET message TO “Hello World!” AS a string
```

Initializing primitive data structures with a value and a type.

```
SET myArray TO [1, 2, 3, 4, 5] AS an array of integers

SET myHashTable TO {“key1”: “val1”, “key2”: “val2”} AS string keys and string values

SET myLinkedList TO Node(1)
myLinkedList.next = Node(2)
myLinkedList.next.next = Node(3)
```

Initializing data structures with its elements and their associated types.

Pseudocode – Playing with Conditions

Pseudocode

```
SET age TO 20 AS an integer

IF age < 18 THEN
        DISPLAY “Underage”
ELIF age >= 18 AND age < 65 THEN
        DISPLAY “Adult”
ELSE
        DISPLAY “Senior”
END IF
```

Code *(python)*

```python
age = 20

if age < 18:
    print("Underage")
elif 18 <= age < 65:
    print("Adult")
else:
    print("Senior")
```

Pseudocode – Playing with Loops

Pseudocode

```
SET counter TO 0 AS an integer

WHILE counter < 5 DO
        DISPLAY “Count is: “, counter
        INCREMENT counter BY 1
END WHILE

FOR i FROM 1 TO 3 DO
        DISPLAY “Iteration: “, i
END FOR
```

Code *(python)*

```python
counter = 0

while counter < 5:
    print("Count is:", counter)
    counter += 1

for i in range(1, 4):
    print("Iteration:", i)
```

Pseudocode – Playing with Functions

Pseudocode

```
FUNCTION addNumbers(a, b)
    SET sum TO a + b
    RETURN sum
END FUNCTION

SET result TO addNumbers(5, 3)
DISPLAY result
```

Code *(python)*

```python
def addNumbers(a, b):
    sum = a + b
    return sum

result = addNumbers(5, 3)
print(result)
```

Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 4 - (EN) Exercises - PseudoCode

Flowcharts

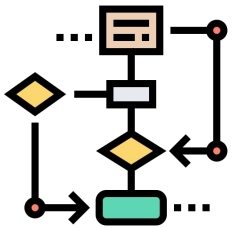
A **flowchart** is a type of diagram that represents a process or workflow. It uses various symbols to depict different steps or actions in a sequence.

Visualization

Flowcharts make complex processes or systems easier to understand by breaking them down into individual steps or stages.

Documentation

They serve as a visual documentation for processes, which can be helpful for training new personnel or for reference.



Planning

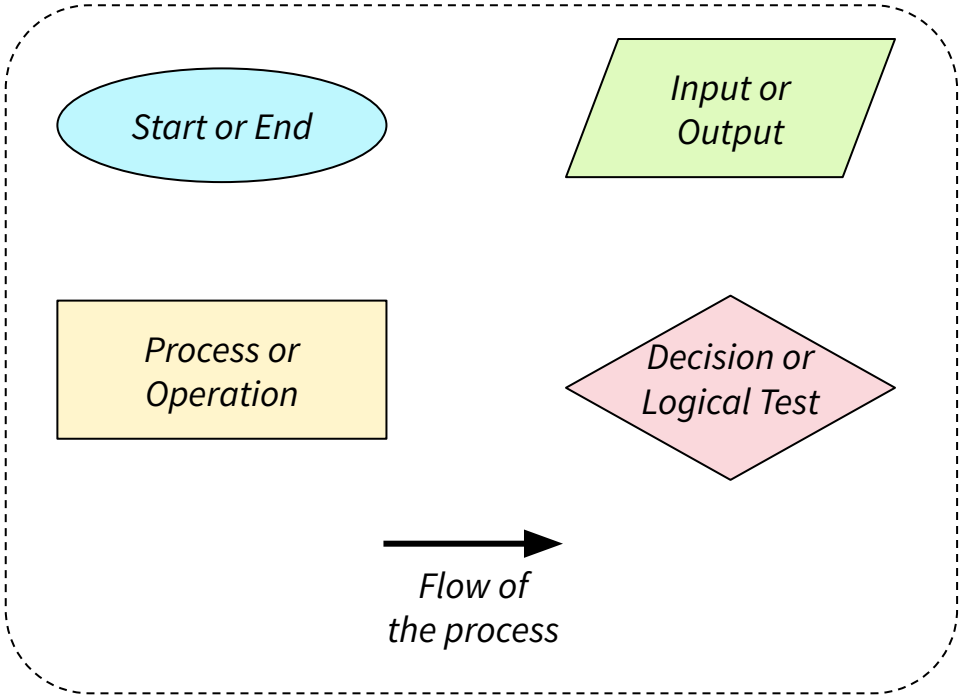
Before coding or implementing a new system, flowcharts can be used to plan the sequence of operations.

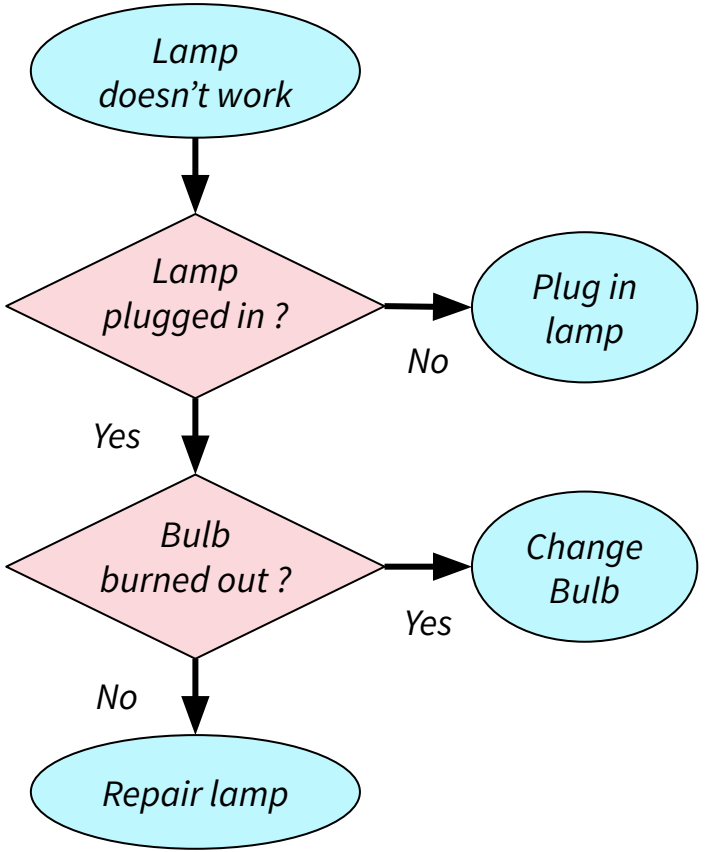
Analysis

By charting out a process, inefficiencies or redundancies can be identified and addressed.

Basic Elements of Flowcharts

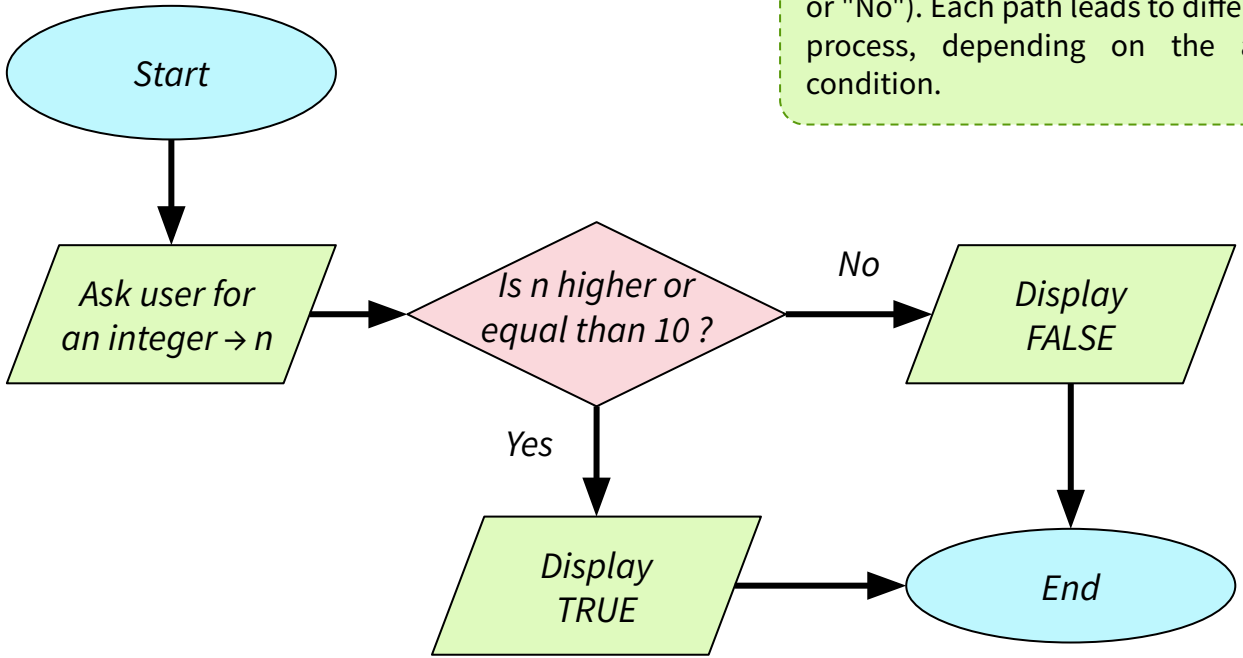
Each shape has a specific meaning. Some common symbols include:



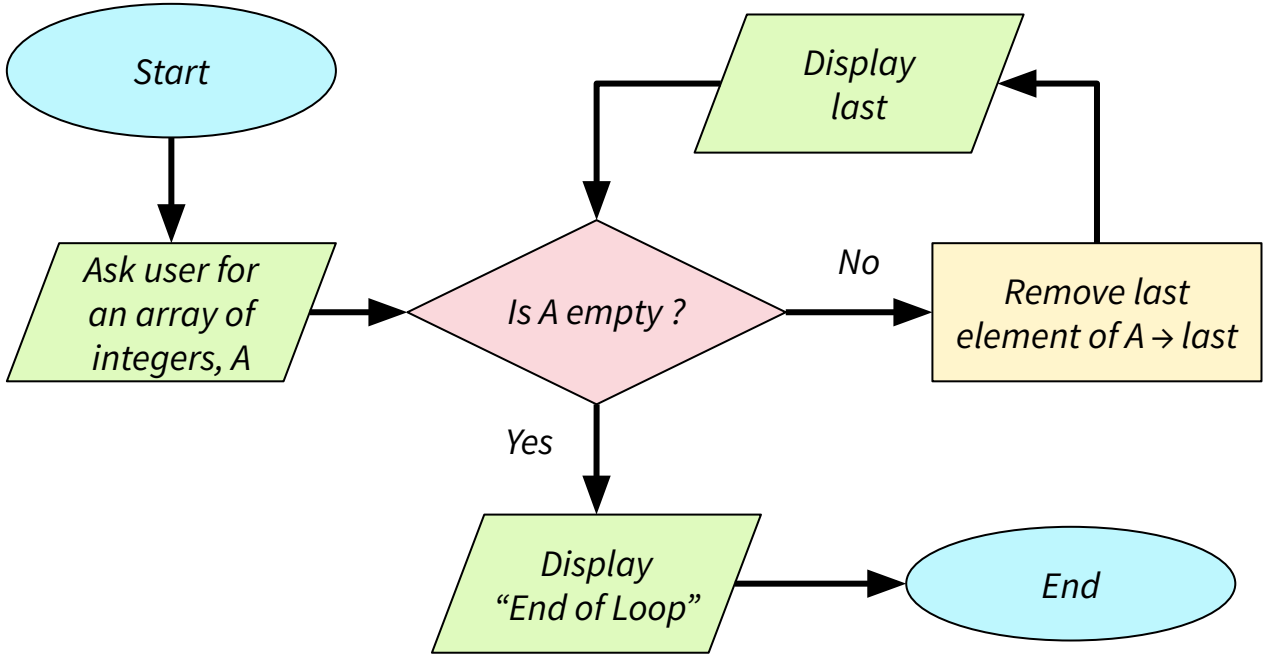


Flowcharts – Conditions

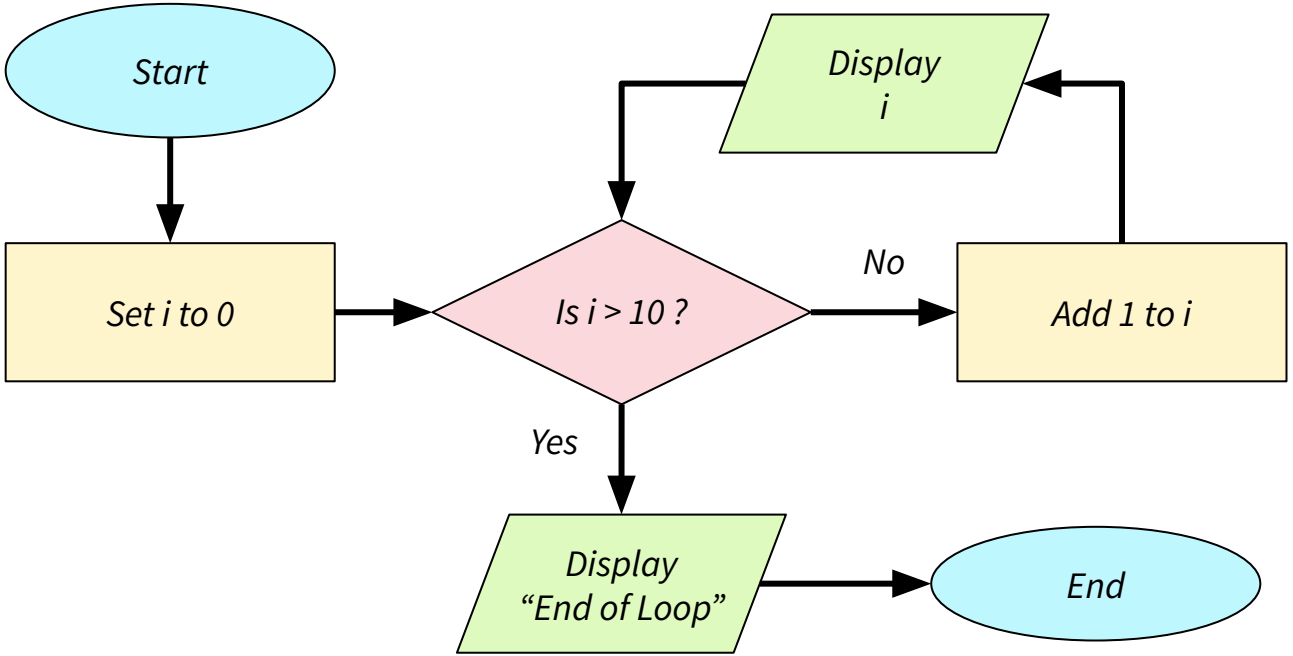
From each condition, two or more paths emerge, representing the possible outcomes (e.g., "Yes" or "No"). Each path leads to different steps in the process, depending on the answer to the condition.



Flowcharts – Loops



Flowcharts – Loops (with a counter)



Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

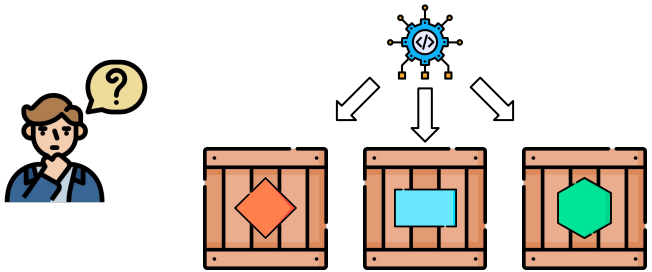
- 5 - (EN) Exercises - Flowcharts

An Attempt of Algorithm Classification

Classifying algorithms is a task that is as fascinating as it is challenging. The difficulty primarily arises from the vast and intricate landscape of algorithmic methods, where distinct categories often blend and overlap. Algorithms, by their nature, are not constrained to rigid classifications; instead, they fluidly move across various methods and strategies, making the task of pigeonholing them into neat categories a complex endeavor.

As per Wikipedia page, multiple possible classifications can co-exist and have their own merits, such as by implementation, by design paradigm, by field of study, by complexity ...

However, most of these methods are overlapping which highly complicates classification. Moreover, the evolution of algorithmic research continuously adds new layers to this complexity. As new problems emerge and existing ones are understood more deeply, the algorithmic landscape shifts, with new categories emerging and existing ones evolving.



A Classification by Implementation

The classification **by implementation** focuses on how algorithms are **executed and structured**. This approach includes aspects like how an algorithm processes information and how it's implemented in terms of computational steps. Each sub-classification within the implementation classification presents a kind of duality.

Recursive by calling themselves with modified inputs.

or

Non-recursive by not using self-referential methods.

Serial by doing one step at a time.

or

Parallel by doing multiple steps at a time.

Or distributed across multiple machines.

Deterministic by having predictable outcomes.

or

Non-Deterministic by having randomness and/or probabilistic elements.

Exact by having a precise solution.

or

Approximate by having a solution that is close enough to the optimal one.

A Classification by Design

The classification **by design** refers to the fundamental methodology or strategy an algorithm uses to solve a problem. This classification is more about the **conceptual approach**. The different methods of the design classification can be used in different parts of the same algorithm, but they usually don't operate simultaneously in the exact same segment of the algorithm.

Brute-force tries every possible solution.

Randomized algorithms uses random choices to find solutions.

Backtracking builds solutions incrementally and abandons them if they lead to dead ends.

Divide and conquer breaks a problem into smaller non-overlapping parts and combines the results.

Reduction of complexity involves transforming complex problems into simpler ones.

Search and enumeration focuses on searching through a data structures (graphs, arrays) and identifying elements.

A Classification by Optimization Method

The classification **by Optimization** focuses on the **kind of problem** the algorithm is solving, and more specifically on algorithms that aim to find the best solution among many possibilities. It differs from the Design and Implementation classifications in its focus on the nature of the problem being solved rather than the method or structure of the algorithm itself.

Linear Programming maximizes or minimizes a function with linear constraints.

Greedy Method makes the locally optimal choice at each step.

Dynamic Programming solves the same smaller problem multiple times within the context of the larger problem.

Heuristic Method focuses on practical methods that produce good enough solutions, often based on experience, intuition, or common sense.

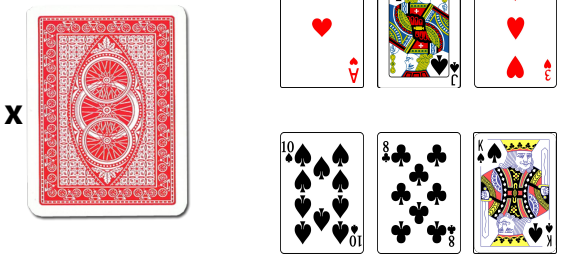
The Brute Force Method

Brute Force involves systematically trying every possible solution to find the correct one. This method is straightforward and guarantees finding the solution if it exists, but it can be very **time-consuming**, especially for problems with a large number of possibilities.

If you're searching for the best combination of 3 cards in a Deck of 20 cards, you would have to compare all the possible combinations:

$$\binom{20}{3} = \frac{20!}{3!(20-3)!} = 1140$$

20 x



If you're guessing a password with all the characters from the ASCII table (33 to 126), brute force would mean trying every possible combination until you find the right one.

Password Size	Number of Combinations	*Estimated time to find the password*
4	74 805 201	< 1 sec
8	> 5 million billion (5.10^{15})	~ 2 days
12	> 4.10^{23}	> 50 years
16	> 3.10^{31}	> 4 billion years

Brute Force – Password Finding Algorithm

```
SET target_password TO user input AS a string

FUNCTION bruteForcePassword(target_password , max_length, characters)
        FOR length FROM 1 TO max_length DO
                GENERATE all possible combinations of characters of size length
                FOR each possible combination of characters DO
                        IF combination matches target_password THEN
                                RETURN “Password Found”
                        END IF
                END FOR
        END FOR
        RETURN “Password Not Found”
END FUNCTION

SET result TO bruteForcePassword(target_password, max_length, characters)
DISPLAY result
```

Brute Force – Password Finding Algorithm

It took more than 30 seconds to find the password on my computer.

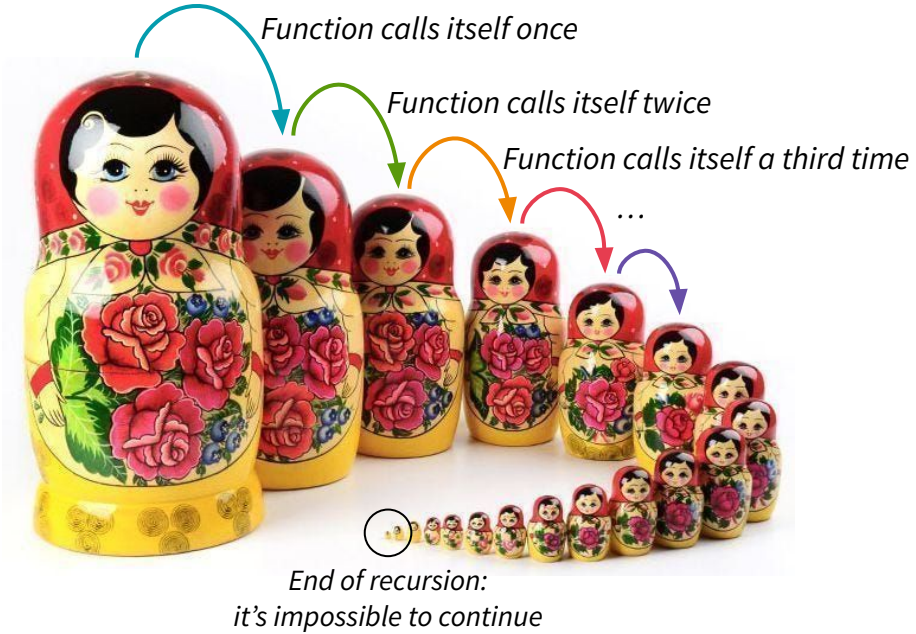
The **itertools** module (documentation) provides a collection of tools for efficient looping and iteration (creating iterators, generating combinations and permutations.

```python
import itertools

def brute_force_password(target_password, maximum_length, characters):
    for length in range(1, maximum_length + 1):
        combinations = itertools.product(characters, repeat=length)
        for combination in combinations:
            if ''.join(combination) == target_password:
                return "Password Found"
    return "Password Not Found"

target_password = "H3llo"
maximum_length = 5
characters = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'

result = brute_force_password(target_password, maximum_length, characters)
print(result)
```

Recursion

Recursion is a method where a function calls itself to solve a problem. It breaks down the problem into smaller, similar problems, solves each smaller problem with the same function, and combines these solutions to solve the original problem. The process repeats until it reaches a basic, easily solvable condition.

It is like a stack of Russian nesting dolls. Opening one doll reveals a smaller one inside, and you keep opening each smaller doll until you reach the smallest one.



Recursion - The Factorial Problem

The **factorial of a number** is a classic example of a problem that can be elegantly expressed and solved using a recursive algorithm. From a mathematical perspective, the problem can be stated as the following:

$$n! = 1 \times 2 \times 3 \times \cdots \times n$$

The factorial function naturally lends itself to a recursive definition because it can be broken down into simpler, smaller instances of itself. Specifically, the factorial of a number n can be expressed in terms of the factorial of n-1. This is evident from the mathematical definition:

$$\begin{aligned}
n! &= n \times (n-1)! \\
n! &= n \times (n-1) \times (n-2)! \\
n! &= n \times (n-1) \times (n-2) \times (n-3)! \\
&= \cdots \\
n! &= n \times (n-1) \times (n-2) \cdots \times 3 \times 2 \times 1
\end{aligned}$$

$$\begin{aligned}
5! &= 5 \times 4! \\
4! &= 4 \times 3! \\
3! &= 3 \times 2! \\
2! &= 2 \times 1! \\
1! &= 1
\end{aligned}$$

Recursion - The Factorial Problem

```python
def factorial(n):

    if n == 1:
        return 1

    else:
        return n * factorial(n - 1)
```

Recursive Case

It's where the function recursively calls itself until it stops by reaching the Base case instead.

Base Case

It's the exit of the function. After this case, no more recursive calls are made.

1. First, when you call `factorial(4)`, the function returns `4 * factorial(4 - 1)`. This is where recursion happens. The function calls itself, but with n - 1.

2. Thus, to get the answer of factorial(4), it waits for factorial(3), and so on:
   1. `factorial(4)` waits for `factorial(3)`
   2. `factorial(3)` waits for `factorial(2)`
   3. `factorial(2)` waits for `factorial(1)`

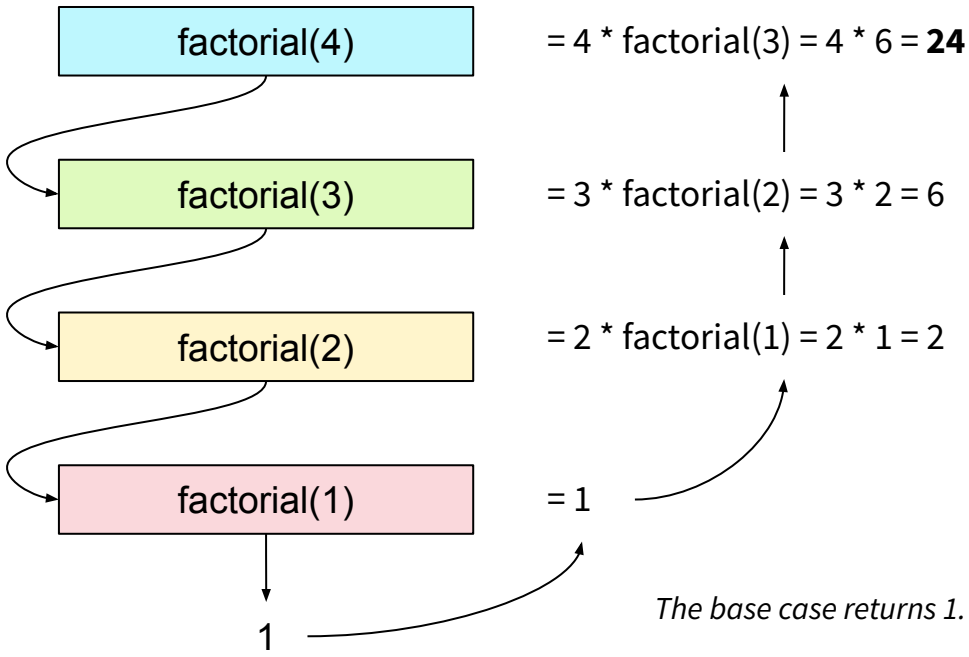
3. Once n becomes 1, `factorial(1)` returns 1 because of the Base Case, and no more recursive calls are made.

4. Now, each waiting function call resolves. The original call `factorial(4)` now has all the information to complete and returns the final result, 24.

Recursion - The Factorial Problem

```python
def factorial(n):

    if n == 1:
        return 1

    else:
        return n * factorial(n - 1)
```

Recursion is similar to a **stack** where elements are added and removed from the top in a last-in-first-out order. When a recursive function is called, each call is placed on the top of the stack, and as each call completes, it's removed from the stack.



Infinite Recursion & Recursion Depth - Stack Overflow

Infinite recursion occurs when a recursive function lacks a proper base case or fails to move towards it, leading to the function calling itself indefinitely which will make the program crash due to memory exhaustion, usually giving you a Stack Overflow error. Indeed, **stack overflow** happens when the recursion depth exceeds the stack's capacity, meaning there's no more memory to handle additional function calls.

```python
def bad_factorial(n):
    if n == 0:
        return 1
    else:
        return bad_factorial(n+1)/(n+1)

factorial(1)
```

RecursionError

The **recursion limit in Python** is set to 1000 primarily as a safety measure to prevent infinite recursion.

For deeper recursion needs, Python allows programmers to manually increase the recursion limit using `sys.setrecursionlimit()` from the sys module.

In practice, if a task requires deeper recursion than the default limit, it's often a sign that an **iterative approach or a different algorithm might be more suitable**.

Recursion Limits

Recursion is an elegant solution but it has its limits and isn't always the best approach for algorithm design.

Memory Usage

Each recursive call adds to the call stack, which can lead to high memory usage for deep recursion or large inputs. This might cause a stack overflow.

Performance

Recursive calls can be slower due to the overhead of repeated function calls and stack management.

Readability and Complexity

While recursion can make some algorithms elegant and simple, it can also make them hard to understand and debug, especially for those not familiar with the concept.

Not Always Necessary

Iterative solutions using loops can often replace recursion, especially in cases where recursion's depth is a concern, and they can be more efficient in terms of memory and performance.

Exercises

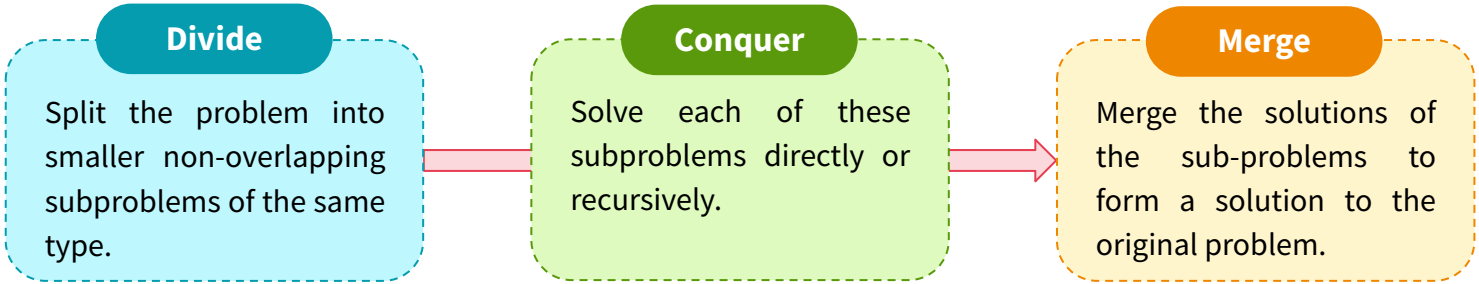
Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 6 - (EN) Exercises - Recursion

The Divide and Conquer Method

The **Divide and Conquer** method is a strategy for solving complex problems in three main steps. The key idea is to divide the problem until it becomes simple enough to solve directly.



In many Divide and Conquer algorithms, **recursion** is used to handle the conquer step, where the problem is broken down into smaller subproblems that are similar to the original problem. However, while many Divide and Conquer algorithms use recursion, recursion itself is a broader concept. In short, Divide and Conquer and Recursion methods intersect but are not subsets or supersets of each other.

Divide and Conquer – The Karatsuba Algorithm

The **Karatsuba algorithm** is an efficient divide and conquer method to multiply large numbers.
Traditional multiplication of two n-digit numbers requires about **n^2 single-digit products**:

$$\begin{aligned} 123 \times 456 = {} & 10^4 \times (1 \times 4) \\ & + 10^3 \times (1 \times 5 + 2 \times 4) \\ & + 10^2 \times (1 \times 6 + 3 \times 4 + 2 \times 4) \\ & + 10^1 \times (2 \times 6 + 3 \times 5) \\ & + 3 \times 6 \end{aligned}$$

There are 9 single-digit products plus some additions and shifts (multiplications by powers of 10)

Instead of this, the **Karatsuba algorithm** proceed to:

Divide

The Karatsuba algorithm splits each large number into two halves as follows:

$$X = 10^{n/2} \times a + b$$
$$Y = 10^{n/2} \times c + d$$

Divide and Conquer – The Karatsuba Algorithm

Conquer

Instead of multiplying these directly, Karatsuba's algorithm involves three multiplications of smaller numbers:

$$(a+b) \times (c+d) \qquad b \times d \qquad a \times c$$

These multiplications are simpler and can be performed more quickly. **This step may be applied recursively** to these smaller products if they are still large.

Merge

After these multiplications, the original product can be reconstructed using these smaller products, with some additions and shifts (multiplications by powers of 10):

$$X \times Y = 10^n \times ac + 10^{n/2} \times ((a+b)(c+d) - ac - bd) + bd$$

Divide and Conquer – The Karatsuba Algorithm

Divide

Conquer

bd = 23 x 56

ac = 4

(a+b)(c+d) = 24 x 60

Recursion.

bd and **(a+b)(c+d)** multiplications are still too big, so they should also be calculated with the Karatsuba algorithm **recursively**.

However, the **Karatsuba Algorithm is not better than the traditional multiplication algorithm for small numbers**. Thus, they can be calculated using the traditional multiplication algorithm.

Merge

$$X \times Y = 10^4 \times ac + 10^2 \times ((a+b)(c+d) - ac - bd) + bd$$

The 3 smaller multiplications can then be used in the final formula.

Divide and Conquer – The Karatsuba Algorithm

```python
def karatsuba(x, y):
    if x < 10 or y < 10:
        return x * y

    size = max(len(str(x)), len(str(y)))
    m = size // 2

    a, b = x // 10**m, x % 10**m
    c, d = y // 10**m, y % 10**m

    bd = karatsuba(b, d)
    ab_cd = karatsuba((a + b), (c + d))
    ac = karatsuba(a, c)

    return (ac * 10**(2 * m)) + ((ab_cd - ac - bd) * 10**m) + bd
```

Base Case – Perform a regular multiplication when one of the number is a digit (0-9).

Recursive Call – Three different and non-overlapping subproblems are created. Each of them can be recursive (i.e. to create new subproblems).

Recursive Case – This step exploits the Karatsuba formula explained earlier.

Divide and Conquer Variants

Two subtle variations of the Divide and Conquer approach exist:

Decrease and Conquer

This variant focuses on **reducing a problem to a single smaller instance in each step**, rather than multiple subproblems.

This approach differs from regular Divide and Conquer, which typically involves dividing a problem into multiple independent subproblems, solving each recursively, and then combining the solutions.

Transform and Conquer

This variant focuses on **transforming the original problem into a more manageable form**, solving the transformed problem, and sometimes transforming the solution back to the original form.

This differs from classic Divide and Conquer as it emphasizes problem transformation as a key step in problem-solving, rather than dividing the problem into subproblems.

Decrease and Conquer - Exponentiation by Squaring

The **Exponentiation by Squaring** method efficiently computes large powers by dividing the problem into smaller parts, using the fact that:

Even formula $$a^{2n} = (a^n)^2$$

Odd formula $$a^{2n+1} = a \times (a^n)^2$$

This method significantly reduces the number of multiplications needed compared to the naive method of multiplying the base number by itself repeatedly.

$$a^{10} = a \times a \times a \times a \times a \times a \times a \times a \times a \times a$$

Even formula $$a^{10} = a^5 \times a^5$$

$$a^5 = a \times a \times a \times a \times a$$

Odd formula $$a^5 = a \times a^2 \times a^2$$

$$a^2 = a^1 \times a^1$$

$$a^1 = a$$

a^{10} cost 9 multiplication steps using the naive method. However, it only cost 7 steps with the Exponentiation by Squaring method:

- 3 divisions by 2 – 10 to 5, 5 to 2 and 2 to 1
- 4 multiplications
  - a^1 x a^1
  - a x a^2 x a^2 (*a^2 has already been calculated before*)
  - a^5 x a^5 (*a^5 has already been calculated before*)

Decrease and Conquer - Exponentiation by Squaring

```python
def power(a, n):

    if n == 0:
        return 1

    if n == 1:
        return a

    half_power = power(a, n // 2)

    if n % 2 == 0:
        return half_power * half_power

    else:
        return a * half_power * half_power
```

Base Cases:

- **n = 0** – Any number raised to the power of 0 is equal to 1.
- **n = 1** – Any number raised to the power of 1 is the number itself.

Recursive Call – This step halves the problem size, aligning with the decrease and conquer approach.

Recursive Case for Even Exponent – This step exploits the property that $a^{2k} = (a^k)^2$ for even exponents.

Recursive Case for Odd Exponent – This step exploits the property that $a^{2k+1} = a(a^k)^2$ for odd exponents.

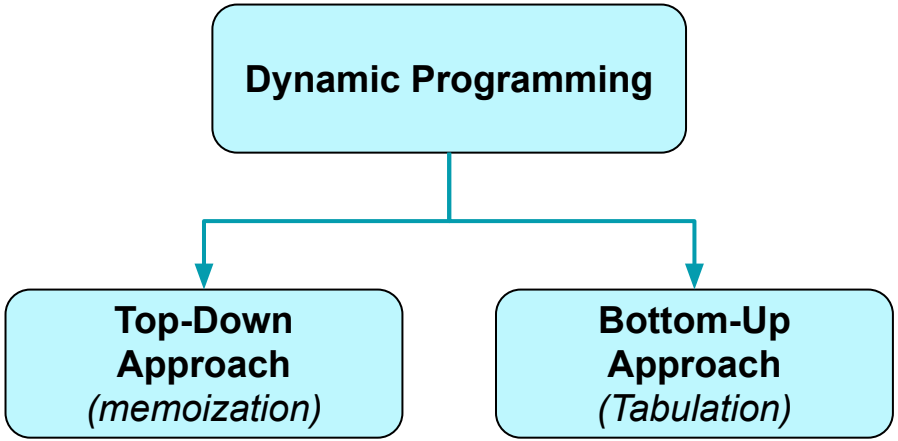
Dynamic Programming

Dynamic Programming is an **Optimization method** for solving complex problems by breaking them down into simpler subproblems and solving each of these subproblems just once. The key idea is to avoid redundant calculations, typically by using a table to store the results of subproblems.

Dynamic Programming is different than **Divide and Conquer**.

Indeed, **Divide and Conquer** splits a problem into independent subproblems whereas **Dynamic Programming** is used when subproblems overlap.

Moreover, **Dynamic Programming** saves the results of these subproblems to avoid redundant work whereas **Divide and Conquer** doesn't.



Dynamic Programming - Fibonacci

Fibonacci numbers form a sequence where each number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on.

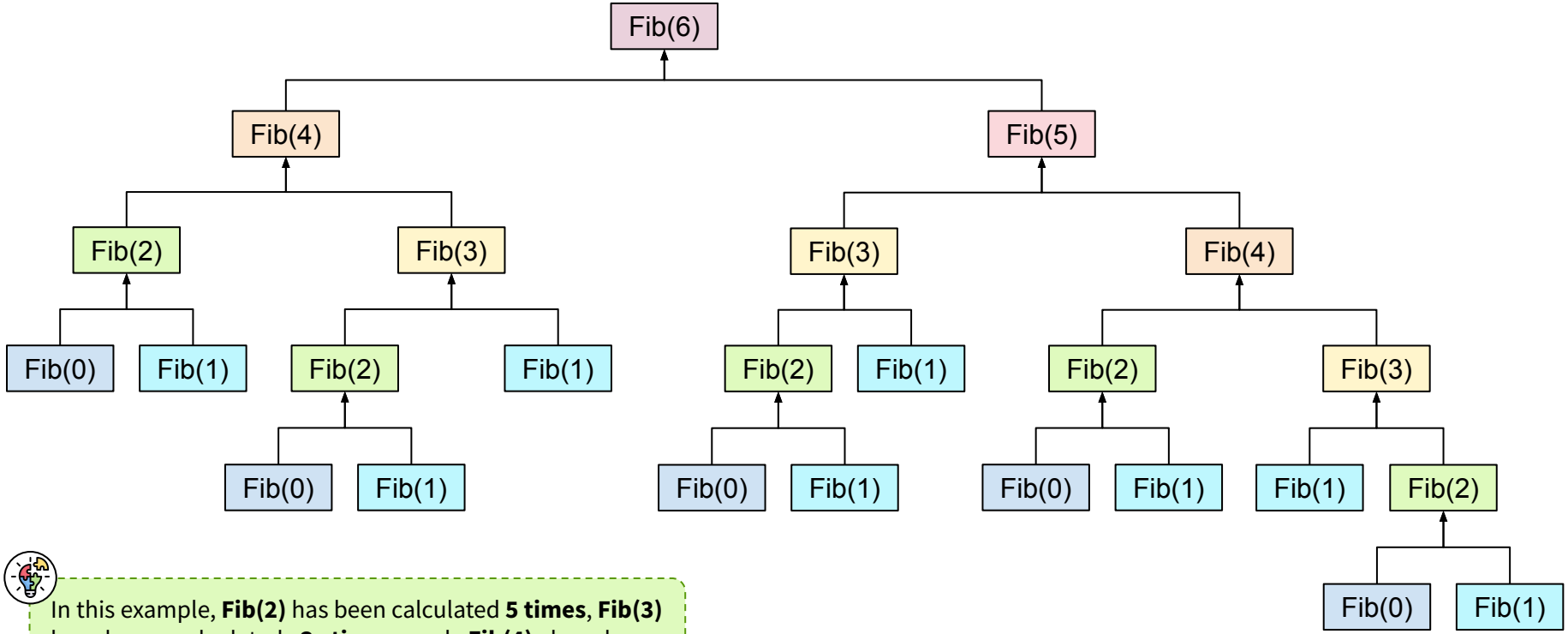
A **naive, non Dynamic Programming, recursive approach** for Fibonacci numbers can directly implement this definition. This approach is considered naive because it involves a lot of redundant computations.

For example, to calculate fibonacci(5), it needs to calculate fibonacci(4) and fibonacci(3). But to calculate fibonacci(4), it again calculates fibonacci(3), and so on. This results in an exponential number of function calls for larger numbers, which is highly inefficient.

```
0 + 1 = 1
    1 + 1 = 2
        1 + 2 = 3
            2 + 3 = 5
                3 + 5 = 8
                    5 + 8 = 13
```

```python
def fibonacci(n):

    if n <= 1:
        return n

    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

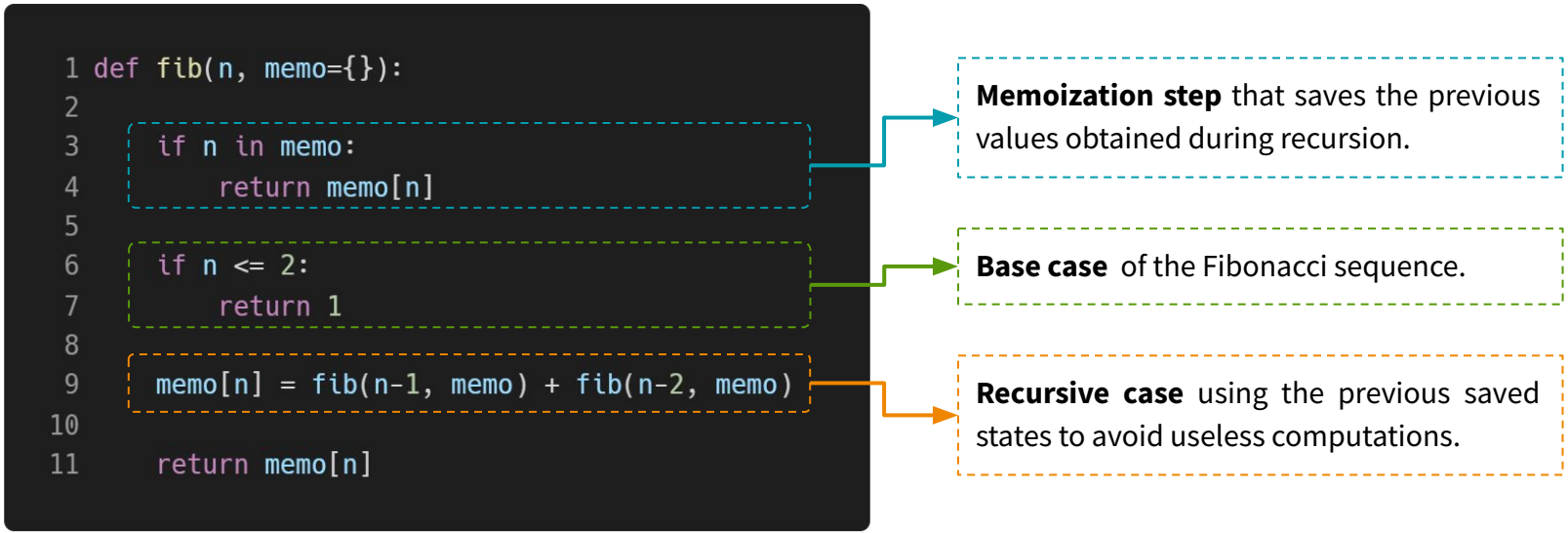
Dynamic Programming - Fibonacci



In this example, **Fib(2)** has been calculated **5 times**, **Fib(3)** has been calculated **3 times** and **Fib(4)** has been calculated **2 times**, leading to 7 redundant calculations.

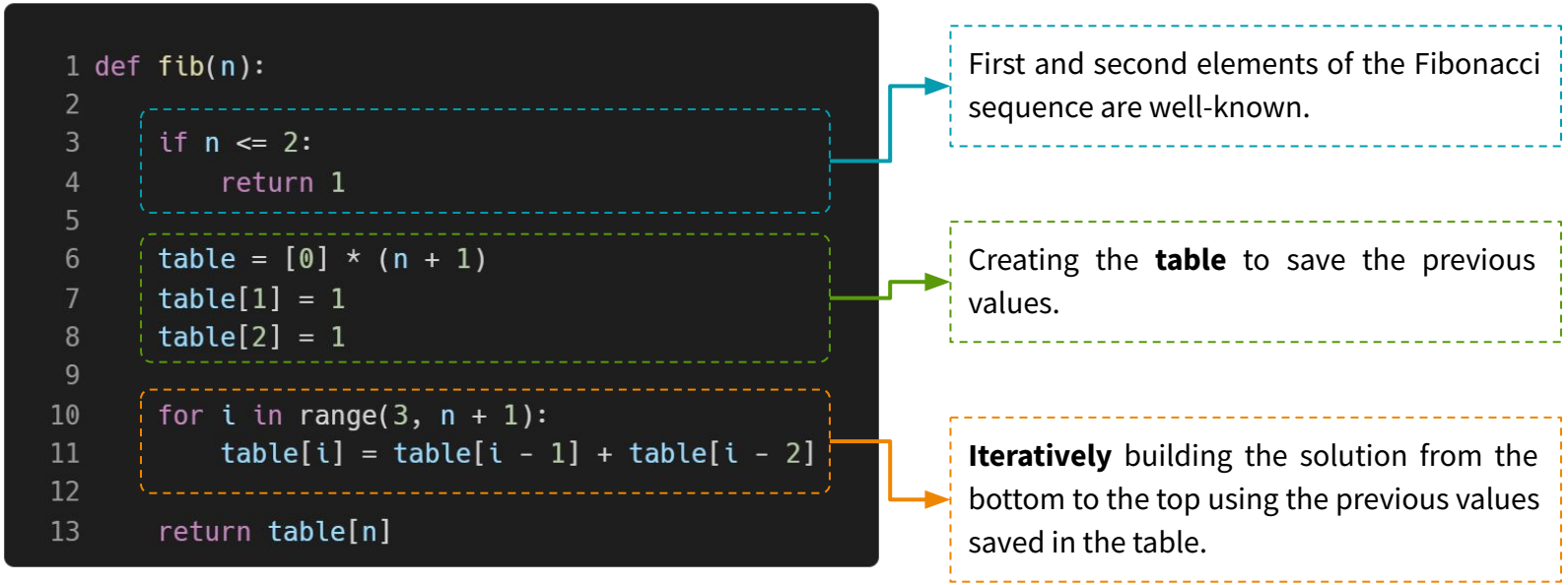
Dynamic Programming – Memoization

The **Top-Down approach** in Dynamic Programming starts with the original problem and breaks it down into smaller subproblems recursively. As each subproblem is solved, its result is stored – memoized – to avoid redundant calculations for the same subproblem in the future.



Dynamic Programming – Tabulation

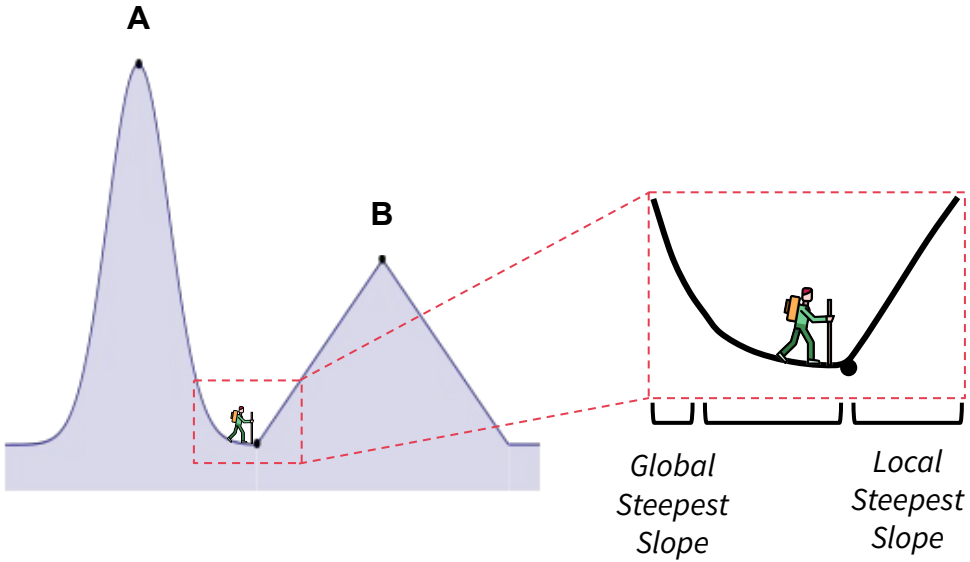
The **Bottom-Up approach** in Dynamic Programming involves starting from the simplest subproblems and iteratively building up the solution to larger subproblems. This approach often uses a table to systematically store the results of these subproblems. This approach in does not use recursion but relies on **iterative** methods instead.



Greedy Algorithms

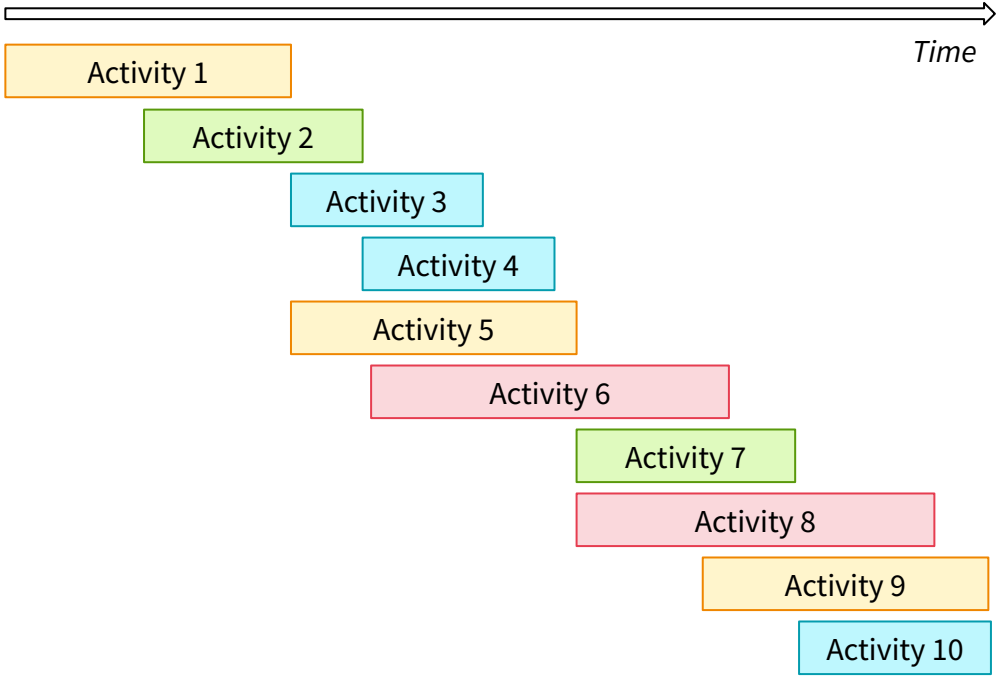
A **Greedy algorithm** is another type of **optimization method** that makes the most optimal choice at each step. They pick the **locally optimal solution**, with the hope that these local solutions will lead to a globally optimal solution. They are often simpler and faster than other approaches, but they don't always provide the best solution for all problems.

At each step, a greedy algorithm makes a choice that seems to be the best at that moment. This is known as making a locally optimal choice, without considering the bigger problem as a whole.

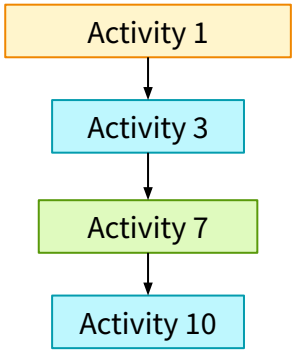


Greedy Algorithms – The Activity Selection Problem

Suppose we have a **set of activities**, each with a start time and a finish time. The goal is to select the maximum number of activities that do not overlap in their time slots.



The greedy algorithm for this problem selects the activity that finishes first, then chooses the next activity that starts after the previously selected one finishes, and so on.



Greedy Algorithms – The Activity Selection Problem

```python
def select_activities(activities):

    # sort based on the activity's ending time !
    activities.sort(key=lambda x: x[-1])
    selected_activities = [activities[0]]

    for i in range(1, len(activities)):
        if activities[i][1] >= selected_activities[-1][-1]:
            selected_activities.append(activities[i])

    return selected_activities

# (start, end)
activities = [("Activity 1", 1, 4),
              ("Activity 2", 3, 5),
              ("Activity 3", 0, 6),
              ("Activity 4", 5, 7),
              ("Activity 5", 8, 9),
              ("Activity 6", 5, 9)]
print(select_activities(activities))
```

Greedy Algorithm Correctness

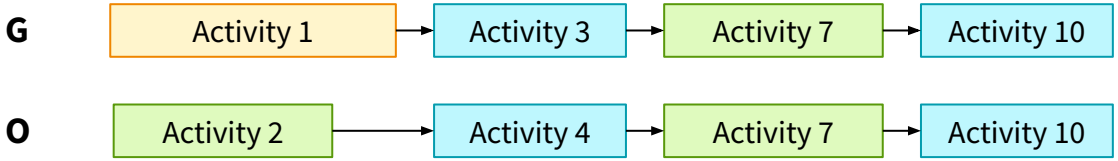
For a greedy algorithm to be correct, it is often necessary to prove that a greedy choice leads to an optimal solution. The two primary methods for proving the optimality of a greedy algorithm are:

- **Greedy Stays Ahead** – It involves demonstrating that, at each step of the algorithm, the greedy choice is at least as good as the corresponding step in an optimal solution, thereby ensuring that the greedy solution never falls behind in terms of progress.
- **Exchange Arguments** – It shows that any optimal solution can be incrementally transformed into the greedy solution without degrading its quality, thus proving that the greedy solution is also optimal.

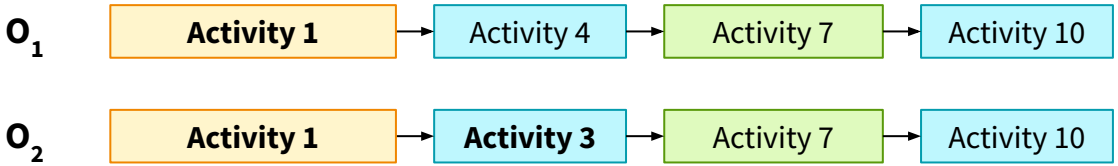
1. First, **defines two solutions** – a greedy algorithm and an optimal solution.
2. Then, **compare the solution** – they must be different at some points.
3. **Replace one optimal solution step by the appropriate greedy one**. It should not worsen the optimal solution quality.
4. **Repeat the exchange process**. If the quality of the optimal solution didn't change, then the greedy solution can also be considered as an optimal solution.

Exchange Arguments – The Activity Selection Problem

Let G be the set of activities selected by the greedy algorithm, and let O be any other set of activities that represents an optimal solution.



Exchange the first differing activity in O with the corresponding one in G. The key here is that since the greedy algorithm chooses the activity that finishes the earliest, the replaced activity in O cannot finish earlier than the one in G. Continue this process for the next activities where G and O differ.



O has been transformed into G without decreasing the number of activities.
Thus, it demonstrates that the set of activities chosen by G is optimal.

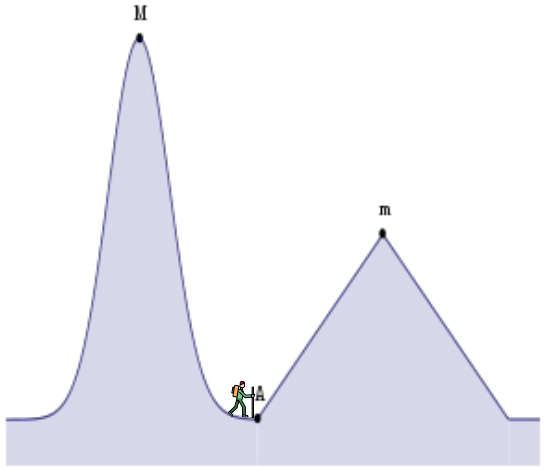
Greedy Algorithms - Main Confusions

Due to their nature of making local optimal choices at each step Greedy algorithms can sometimes be mistaken for other types of algorithms, especially those used for solving optimization problems.

Dynamic Programming

DP ensures an optimal solution by considering all possible combinations of subproblems, which can be more computationally intensive.

DP would involve the hiker analyzing different routes based on past experiences and collected data like maps or previous hikes, breaking the journey into segments, and then choosing a path that maximizes the overall elevation gain.



Backtracking

Backtracking is exhaustive and explores many possibilities, while greedy algorithms are faster but may miss the global optimum by making locally optimal choices.

The hiker starts climbing and keeps track of his path. He explores multiple paths and if encounters a path leading to a lower elevation or a dead-end, he backtracks to the previous decision point and tries a different route.

Randomized Algorithms

Randomized algorithms employs a degree of randomness as part of their logic. They make random choices during their execution to solve problems, often leading to simpler and faster algorithms compared to their deterministic counterparts. Despite their use of randomness, these algorithms are used to solve deterministic problems.

They can be used for approximating numerical answers to problems where deterministic algorithms may be too slow (value of Pi, optimizing complex functions …), in machine learning where they help in handling large datasets and high-dimensional spaces (stochastic gradient descent, random forests …), or even to ensure the unpredictability necessary for security like cryptography (generating keys that are hard to guess …).

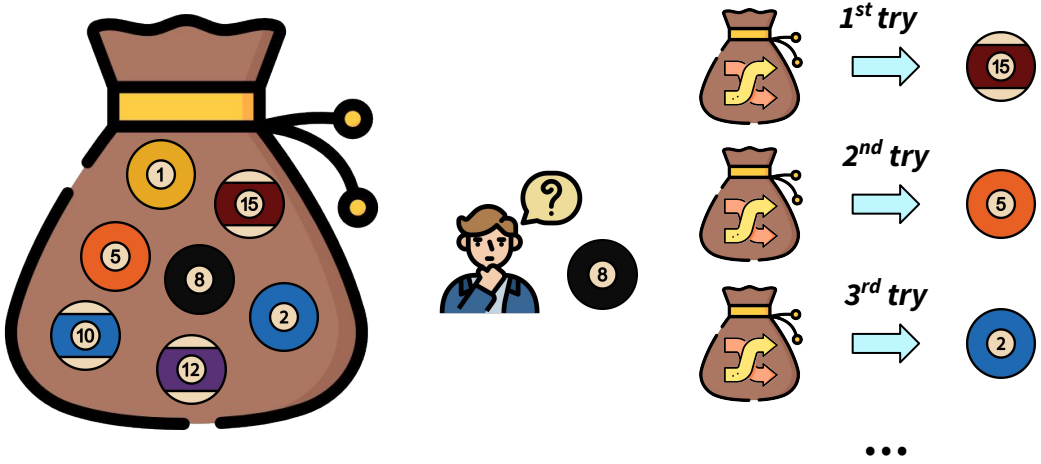
Las Vegas
Algorithms

Monte Carlo
Algorithms

Las Vegas Algorithms

Las Vegas algorithms use randomness to guide their process and always produce either a correct result or an inability to find a result. The running time can vary due to the random choices made, and thus, it may consume a lot of resources to find the correct answer.

In short, the **correctness** of a las vegas algorithm is **certain** but the **running time** is **probabilistic**.



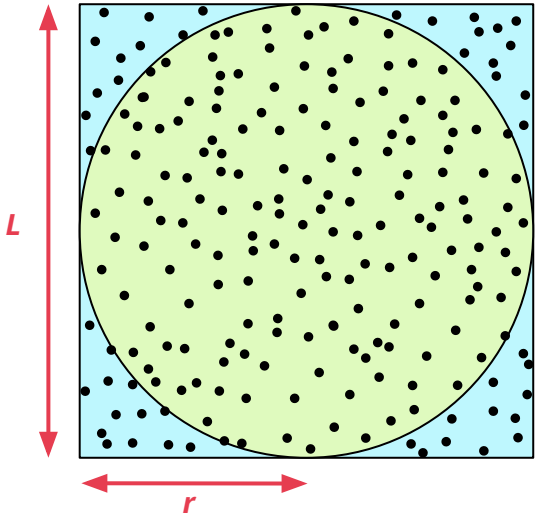
Suppose you are searching for the billiard ball n°8 is the bag. A Las Vegas algorithm would randomly draw one ball from the bag and then put it back … or not !

You may be lucky by drawing the ball n°8 at the start or, (very) unlucky by not drawing it before 100 tries.

Monte Carlo Algorithms

Monte Carlo algorithms use randomness in such a way that they allow a probability of error in the output. They trade off accuracy for speed and are used when a fast approximate solution is acceptable, or when an exact solution is infeasible. They often provide solutions more quickly than both Las Vegas and deterministic algorithms, particularly in complex problems.

In short, the **correctness** of a monte carlo algorithm is **probabilistic** but the **running time** is **certain**.



$$\frac{circle_{area}}{square_{area}} = \frac{\pi r^2}{L^2}$$

$$\frac{circle_{area}}{square_{area}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

$$\pi = 4 \times \frac{circle_{area}}{square_{area}}$$

$$\pi \approx 4 \times \frac{\text{Number of points in the circle}}{\text{Number of points in the square}}$$

Exercises

Estimate Pi by randomly generating points within a square that contains an incircle, and counting how many fall inside of each.

1. Randomly generate points within the square. The coordinates of these points (x, y) should be within [-r, r] for both x and y.

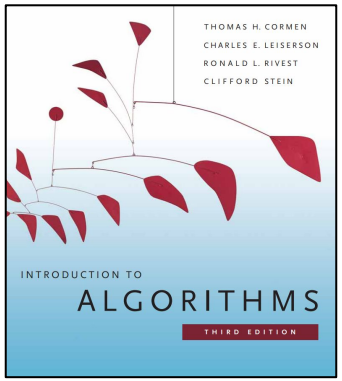
2. For each point, determine if it lies inside the circle centered at the origin with radius r. A point (x, y) lies inside the circle if $x^2 + y^2 \leq r^2$

3. The ratio of the number of points inside the circle to the total number of points generated, multiplied by 4, gives an approximation of π.

Tips: You can use the random module from Python or the one from NumPy.

Is that all ?

It's important to realize that you have only just begun to explore the field of algorithms. This part offered a starting point, but the world of algorithms is much larger and constantly evolving. Beyond these basics, there are many more complex and diverse algorithmic concepts to discover, from advanced problem-solving techniques to the latest developments in machine learning. To truly understand and master algorithms, continuous learning, research, and practice are essential. So, see this as the beginning of your journey into the deep and fascinating world of algorithms, where there's always more to learn and explore.



One of the best book to study algorithms.
Click on the image to get PDF.

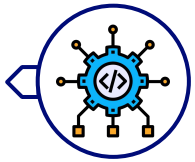
A GitHub repository that contains useful links about Algorithms and Data Structures.
Click on the image to get the Link.

Exercises

For each of the following tasks determine the most effective algorithmic approach and explain why:

1. Planning an efficient route for a road trip
2. Organizing a bookshelf efficiently by genre, author etc.
3. Designing a school timetable that schedules classes and teachers
4. Budgeting for a family vacation by allocating funds to activities
5. Optimizing space utilization in a truck
6. Finding the best deals in a sale by comparing discounts on various products

04

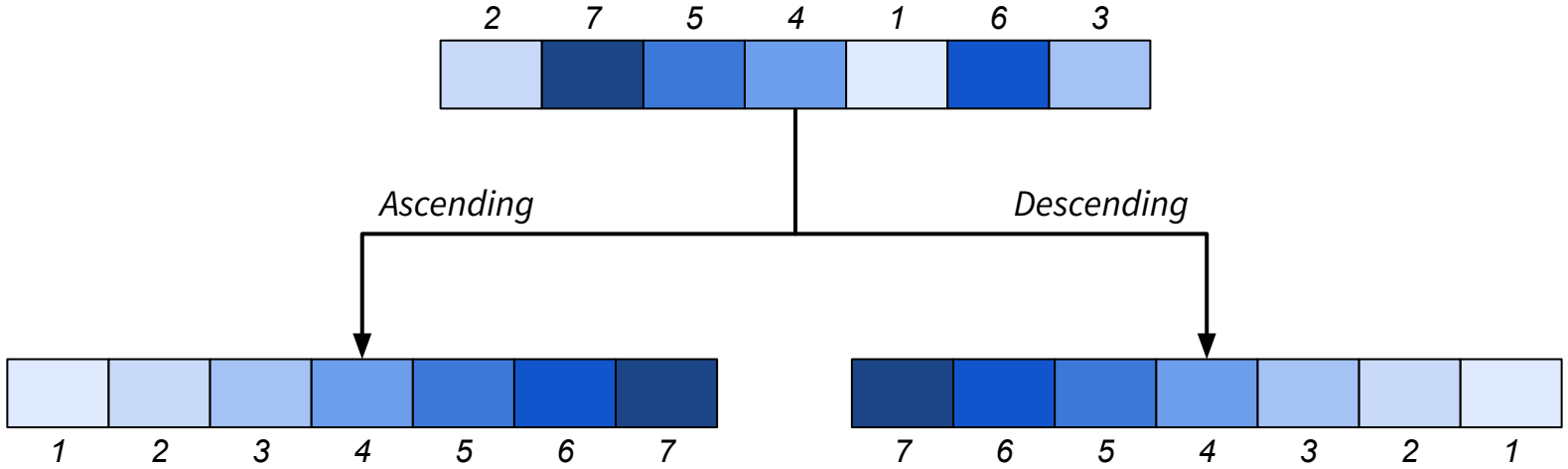


Essential Algorithms

Introduction to Algorithms & Data Structures

Sorting

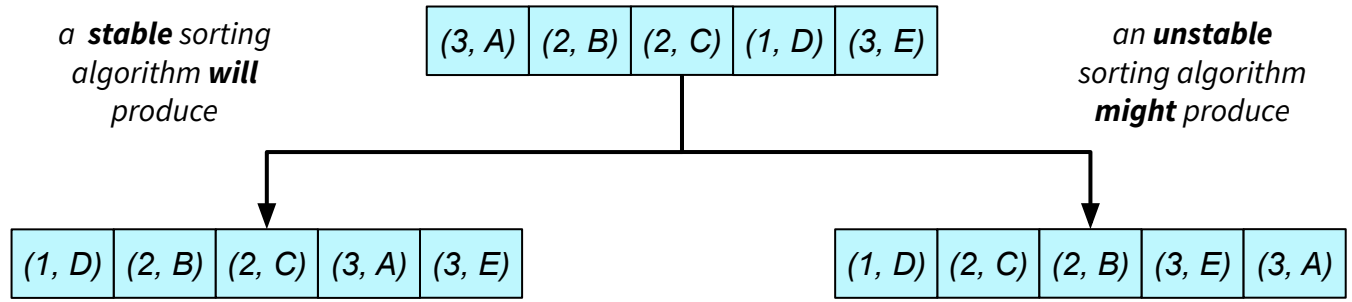
Sorting algorithms are a class of algorithms designed to rearrange elements of an array into a specific order, typically in numerical or alphabetical order. These algorithms are fundamental in computer science and programming because they are used to organize and manage data efficiently. The efficiency of a sorting algorithm is crucial as it directly impacts the performance of complex data processing tasks and systems.



Sorting Stability

Stability is a property that determines whether the algorithm preserves the relative order of equal elements in the sorted output as it was in the input. Conversely, an algorithm is **unstable** if it does not guarantee this property.

Stability is important when sorting data that has multiple attributes. For example, consider a list of people that you first sort by age and then by name. With a stable sort, if two people are the same age, their order in the sorted list will be determined by their names. Moreover, if you sort again only by names, the relative order of people with the same name will still be determined by their ages.



Sorting – Python

```python
mylist = [2,8,3,0,6,4,0,9,5]

mylist.sort()

print(mylsit)
```

[0,0,2,3,4,5,6,8,9]

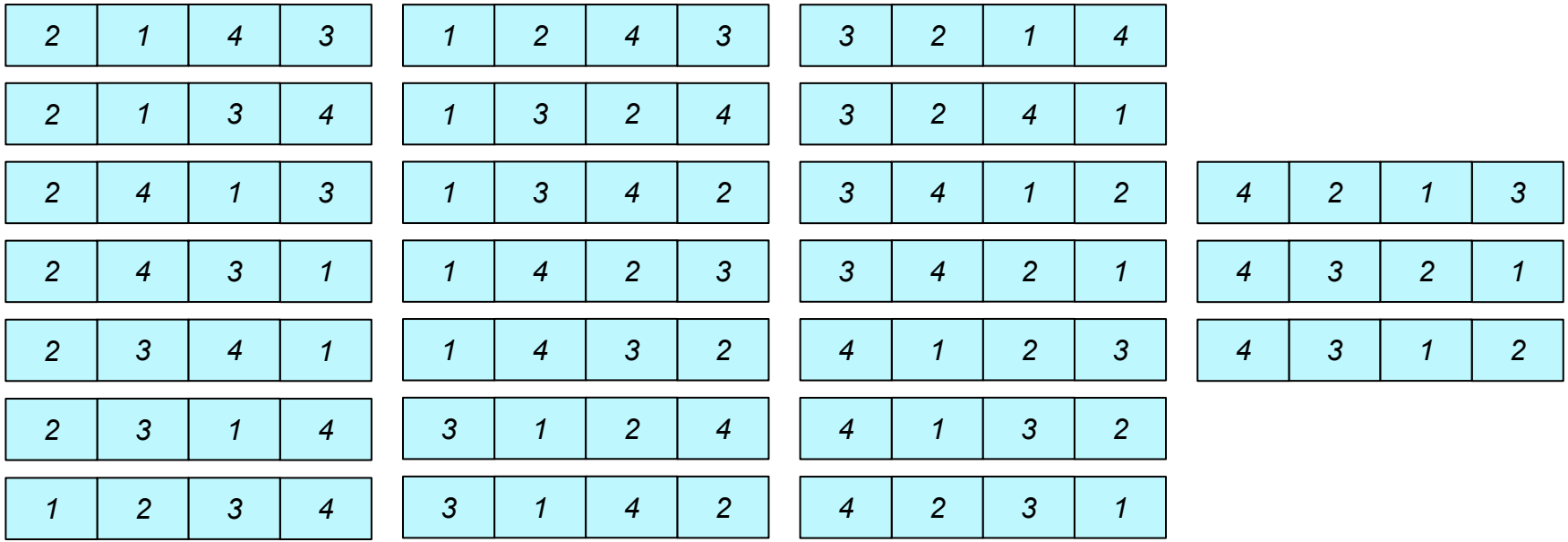
```python
mylist = [2,8,3,0,6,4,0,9,5]

sorted_list = sorted(mylist)

print(sorted_list)
```

[0,0,2,3,4,5,6,8,9]

*What is behind the **.sort()***
*and the **sorted()** function ?*

Sorting – Brute Force approach

A typical **brute force sorting** method involves generating all permutations of the list and checking each to see if it's sorted. This method is highly inefficient but provides a clear understanding of the complexity of sorting and the importance of more sophisticated algorithms.



Exercises

Create an algorithm that can sort any one dimensional array.

Sort the following arrays:

1. [4, 2, 7, 3, 2, 2, 8]
2. [-1, 3, -5, 2, 0, -8]
3. [5]
4. [0, 1, 2, 3, 4, 5, 6]
5. [5, 5, 5, 5, 5, 5, 5]
6. [6, 5, 4, 3, 2, 1, 0]

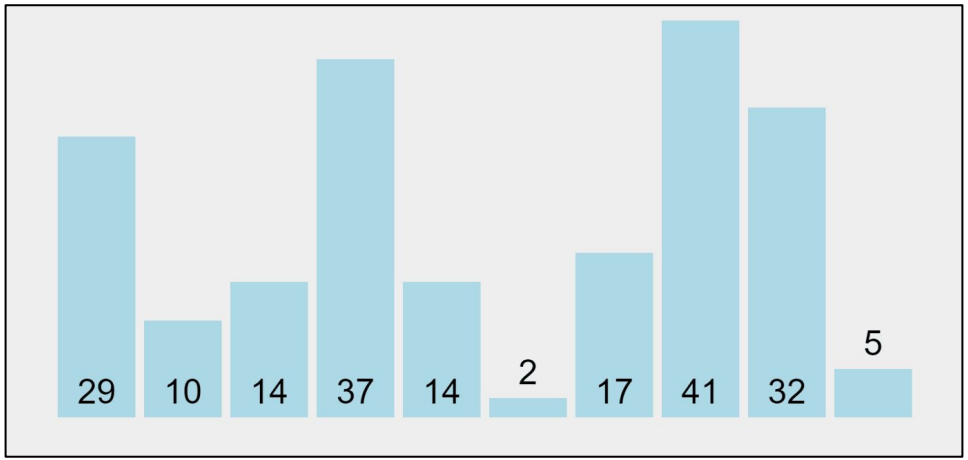
Don't search for an existing algorithm online.

Build your own from scratch.

Explain your logic and assess the efficiency of your algorithm.

Sorting – Bubble Sort

The **Bubble Sort algorithm** is a simple sorting algorithm that repeatedly steps through the array to be sorted, compares adjacent items, and swaps them if they are in the wrong order. Despite its simplicity, Bubble Sort is not efficient for large arrays. It is a stable sort algorithm.

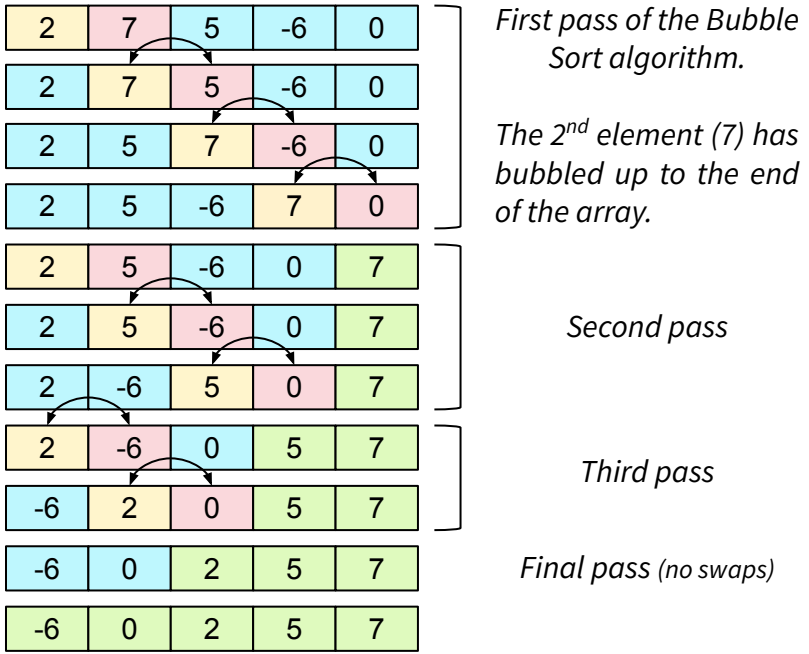


GIF Source : medium.com

Sorting – Bubble Sort

1. The algorithm starts at the beginning of the array.
2. It compares adjacent pairs of elements: if the first element of the pair is greater than the second, the algorithm **swaps** them.
3. The process continues for each pair of adjacent elements. This means that after the first pass, the largest element is located at the end of the array.
4. The entire process is repeated for the remaining array excluding the last element which already sorted.

The array is completely sorted when no more swaps are required.



Exercises

Create a Bubble Sort algorithm.

Sort the following arrays:

1. [4, 2, 7, 3, 2, 2, 8]
2. [-1, 3, -5, 2, 0, -8]
3. [5]
4. [0, 1, 2, 3, 4, 5, 6]
5. [5, 5, 5, 5, 5, 5, 5]
6. [6, 5, 4, 3, 2, 1, 0]

Don't search for an existing algorithm online.

Build your own from scratch.

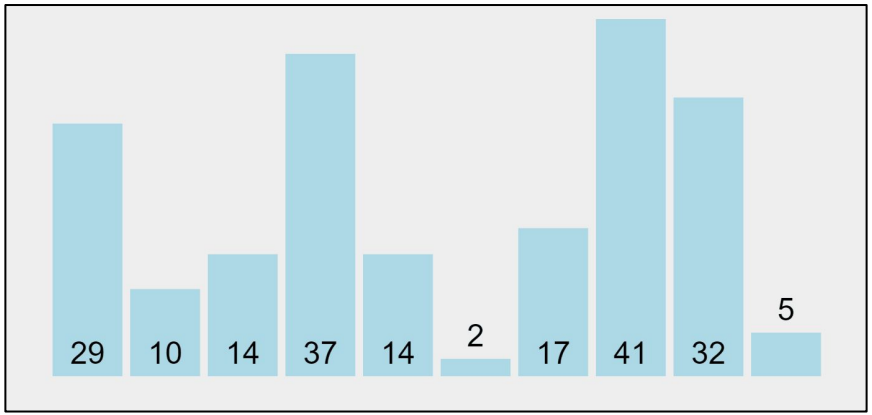
(**Bonus Exercise**) Implement an early stopping method when no swaps occurred in a traversal.
(**Bonus Exercise**) Create a recursive Bubble Sort algorithm.

Sorting – Selection Sort

Selection Sort is a simple comparison-based sorting algorithm. It is an easy to understand and implement algorithm, but it is inefficient for large arrays. The algorithm divides the input array into two parts:

- a sorted subarray of items which is built up from left to right at the front of the array,
- a remaining unsorted subarray occupying the rest of the array.

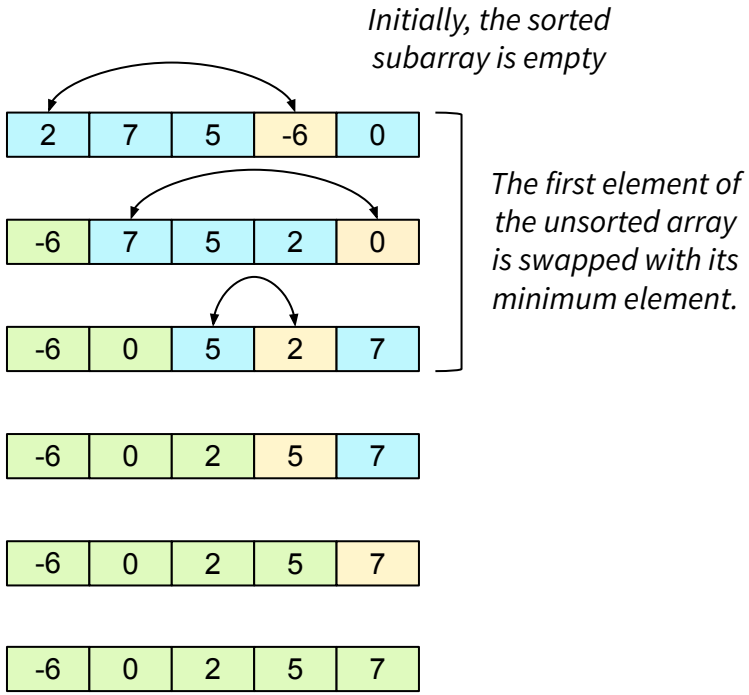
Initially, the sorted subarray is empty, and the unsorted subarray is the entire input array. The default implementation is not stable.



GIF Source : datasciencecentral.com

Sorting – Selection Sort

1. In each iteration, the algorithm looks for the smallest element in the unsorted subarray.
2. Once the minimum element is found, it is swapped with the first unsorted element in the array.
3. After the swap, the sorted subarray grows by one element, and the unsorted subarray shrinks by one element.
4. This process is repeated, moving the new unsorted subarray boundary one element to the right, until the whole array is sorted.



Exercises

Create a Selection Sort algorithm.

Sort the following arrays:

1. [4, 2, 7, 3, 2, 2, 8]
2. [-1, 3, -5, 2, 0, -8]
3. [5]
4. [0, 1, 2, 3, 4, 5, 6]
5. [5, 5, 5, 5, 5, 5, 5]
6. [6, 5, 4, 3, 2, 1, 0]

Don't search for an existing algorithm online.

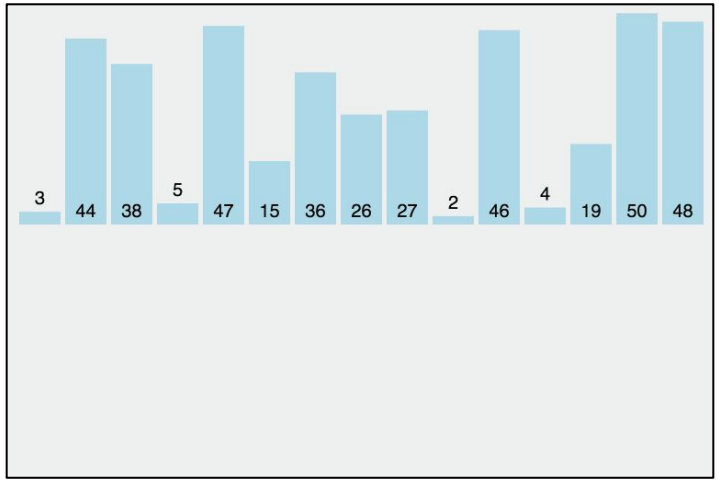
Build your own from scratch.

(**Bonus Exercise**) Create a recursive Selection Sort algorithm.

Sorting – Insertion Sort

Insertion Sort is a simple and efficient sorting algorithm, especially efficient for sorting small lists or partially sorted lists. It builds the final sorted array one item at a time. Its method of operation is similar to the way you might sort playing cards in your hands.

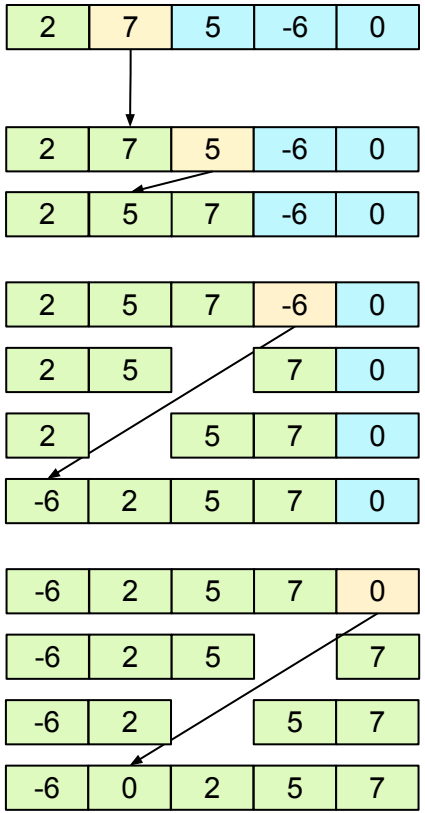
It is **stable** sorting algorithm and it is faster than the Bubble and the Selection sort algorithms when the array is already partially sorted.



GIF Source : medium.com

Sorting – Insertion Sort

1. The algorithm starts with the second element of the array.
2. It compares this element with the ones before it and **inserts it** in the correct position within the sorted subarray. This means moving all larger elements in the sorted subarray to the right to make space. It can be done by **pushing** the elements to make room for the current element.
3. Repeat the process for each element in the array. The array is fully sorted when each element has been inserted into its proper place.



Step 1

7 > 2 – Don't do anything.

Step 2

5 < 7 – Push it to the right

Place the 5 at the previous position of the 7.

Step 3

-6 < 7 – Push it to the right

-6 < 5 – Push it to the right

-6 < 2 – Push it to the right

No more elements. Place the -6 at the previous position of the 2.

Step 4

0 < 7 – Push it to the right

0 < 5 – Push it to the right

0 < 2 – Push it to the right

0 > -6 – Place the 0 at the previous position of the 2.

Exercises

Create a Insertion Sort algorithm.

Sort the following arrays:

1. [4, 2, 7, 3, 2, 2, 8]
2. [-1, 3, -5, 2, 0, -8]
3. [5]
4. [0, 1, 2, 3, 4, 5, 6]
5. [5, 5, 5, 5, 5, 5, 5]
6. [6, 5, 4, 3, 2, 1, 0]

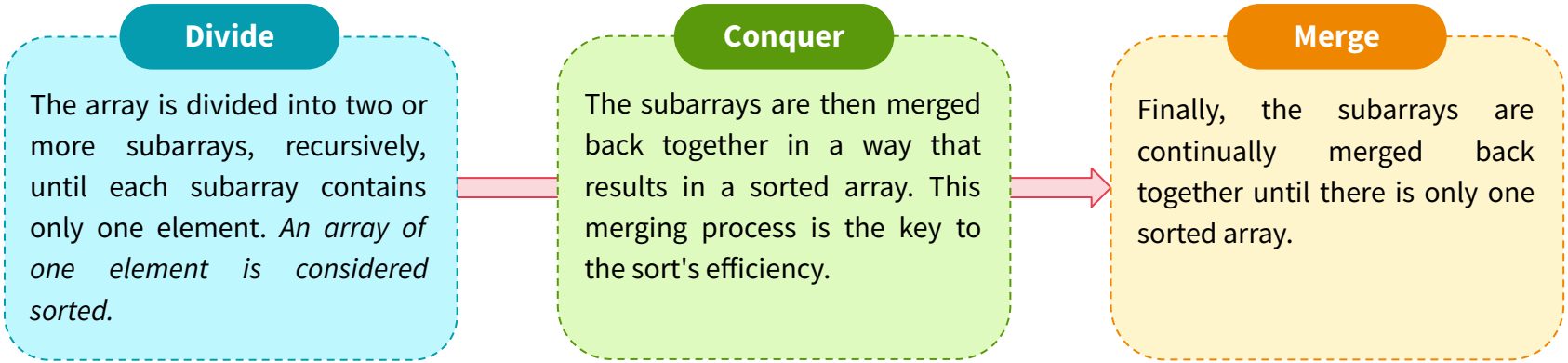
Don't search for an existing algorithm online.

Build your own from scratch.

(**Bonus Exercise**) Create a recursive Insertion Sort algorithm.

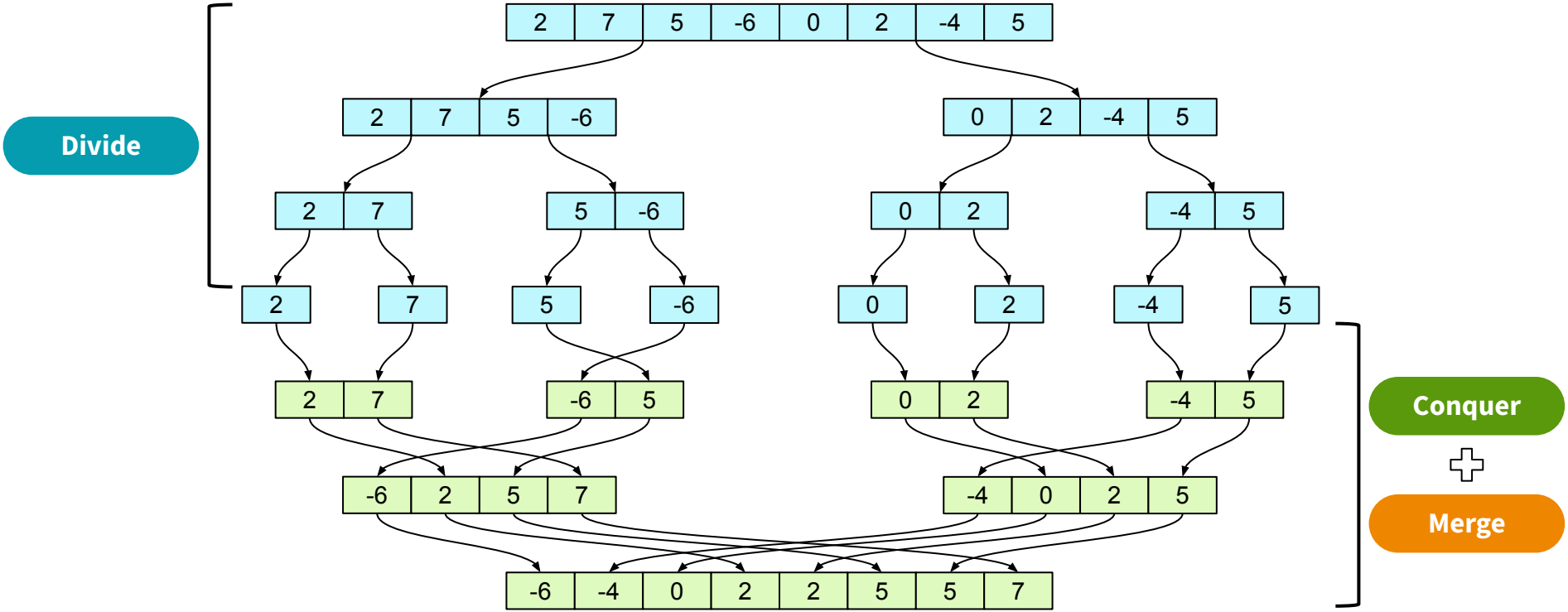
Sorting – Merge Sort

Merge Sort is a highly efficient, stable, and comparison-based sorting algorithm, widely used for its performance and simplicity in handling large datasets. It's a classic example of the **divide and conquer strategy** in algorithm design. Merge Sort requires additional space proportional to the size of the input array. Thus, it might not be the best choice for memory-constrained environments.

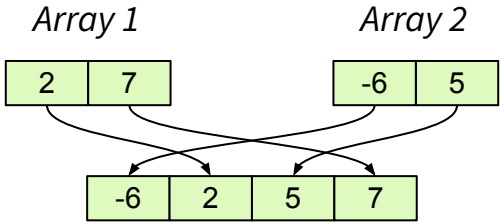


During the merge step, two subarrays are combined into one sorted array. This is done by comparing the elements of the subarrays and transferring the smaller element to the new array, and then moving the comparison to the next element of the subarray from which the element was transferred.

Sorting – Merge Sort



Sorting – Merge Sort



1. Start by comparing the **first elements** of each array.
   Here, compare 2 from Array 1 with -6 from Array 2. Since -6 is smaller, it is added first to the merged array.
   → [-6]

2. Repeat the process.
   Now compare 2 from Array 1 with 5 from Array 2. This time 2 is smaller.
   → [-6, 2]

3. Continue until one array is exhausted.
   Then, append the remaining elements of the other array to the merged array .
   → [-6, 2, 5]
   → Array 2 is exhausted, add the remaining elements of Array 1 ([7]) to the merged array : [-6, 2, 5] + [7]

Exercises

Create a Merge Sort algorithm.

Sort the following arrays:

1. [4, 2, 7, 3, 2, 2, 8]
2. [-1, 3, -5, 2, 0, -8]
3. [5]
4. [0, 1, 2, 3, 4, 5, 6]
5. [5, 5, 5, 5, 5, 5, 5]
6. [6, 5, 4, 3, 2, 1, 0]

Don't search for an existing algorithm online.

Build your own from scratch.

Sorting – Quick Sort

Quick Sort is an **unstable divide-and-conquer sorting** algorithm known for its efficiency in handling large datasets. It operates by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

Quick Sort is often one of the fastest sorting algorithms for average-sized and large arrays, and has a relatively low memory space usage compared to algorithms like Merge Sort. However, it can sometimes be much slower than Merge Sort, particularly when the smallest or largest element is consistently chosen as the pivot.

Divide

The first step is to select a pivot element from the array. The array is partitioned into two parts based on the pivot. Elements less than the pivot are moved to its left, and elements greater than the pivot are moved to its right.

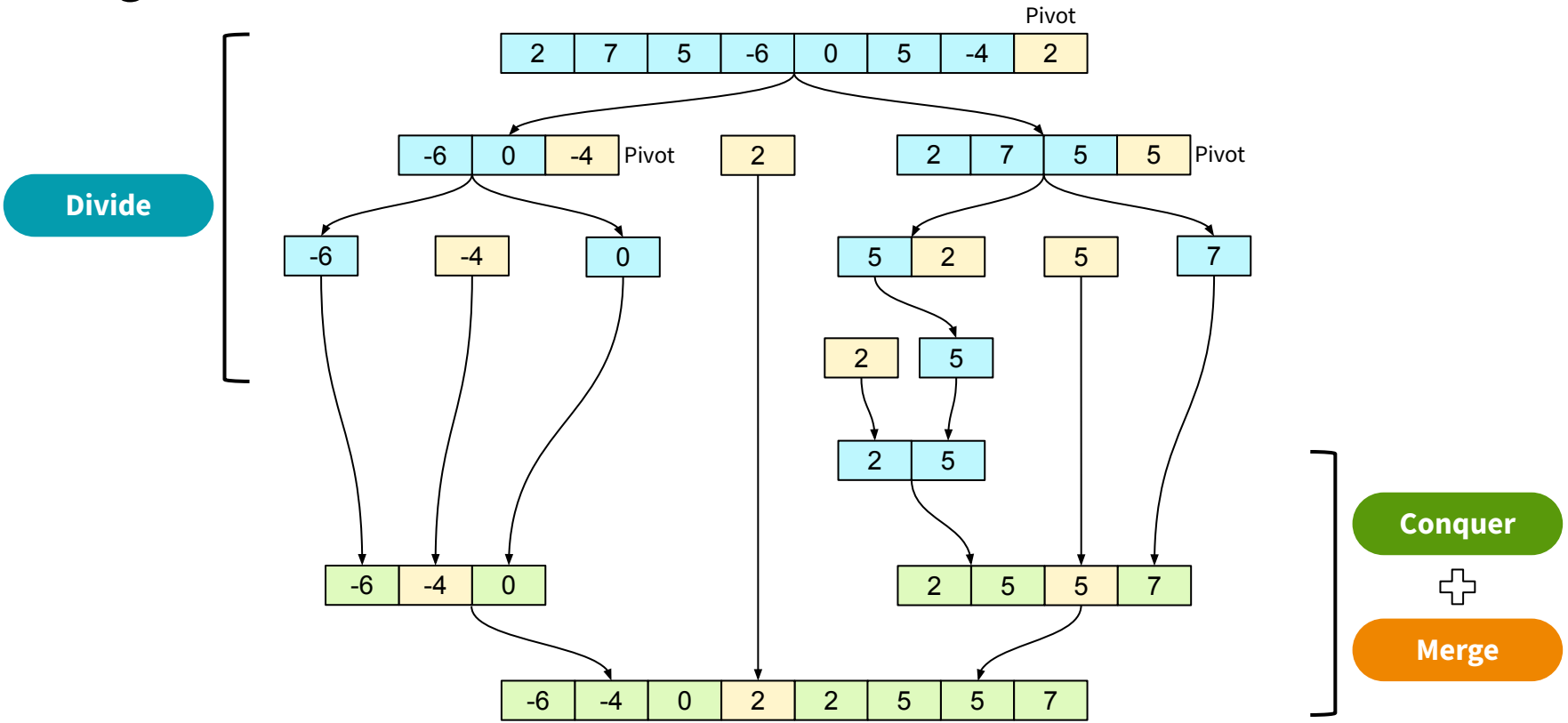
Conquer

The divide step is applied recursively. After partitioning, the array is divided into two sub-arrays and the pivot is now in its final sorted position. The algorithm is recursively applied to the left and right sub-arrays.

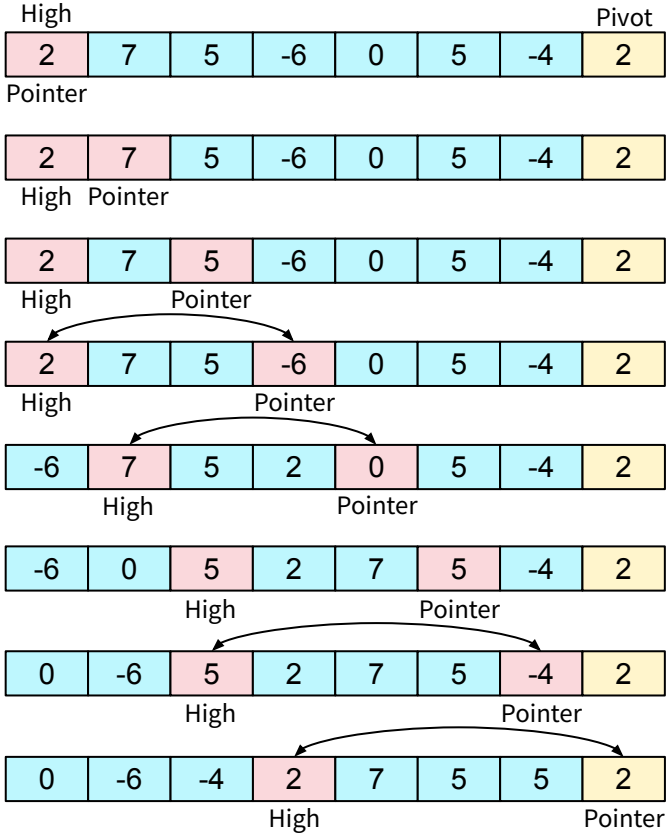
Merge

Unlike Merge Sort, Quick Sort does not have an explicit combine or merge step. The merging is implicit in the partitioning process.

Sorting – Quick Sort



Sorting – Quick Sort



The **pointer** (2) is not less than the **pivot** (2), so the pointer is increased.

Same (7 is not less than 2).

Same (5 is not less than 2).

The **pointer** (-6) is less than the **pivot** (2), so the pointer is swapped with the high element (2). Then, both the pointer and the high element location are incremented.

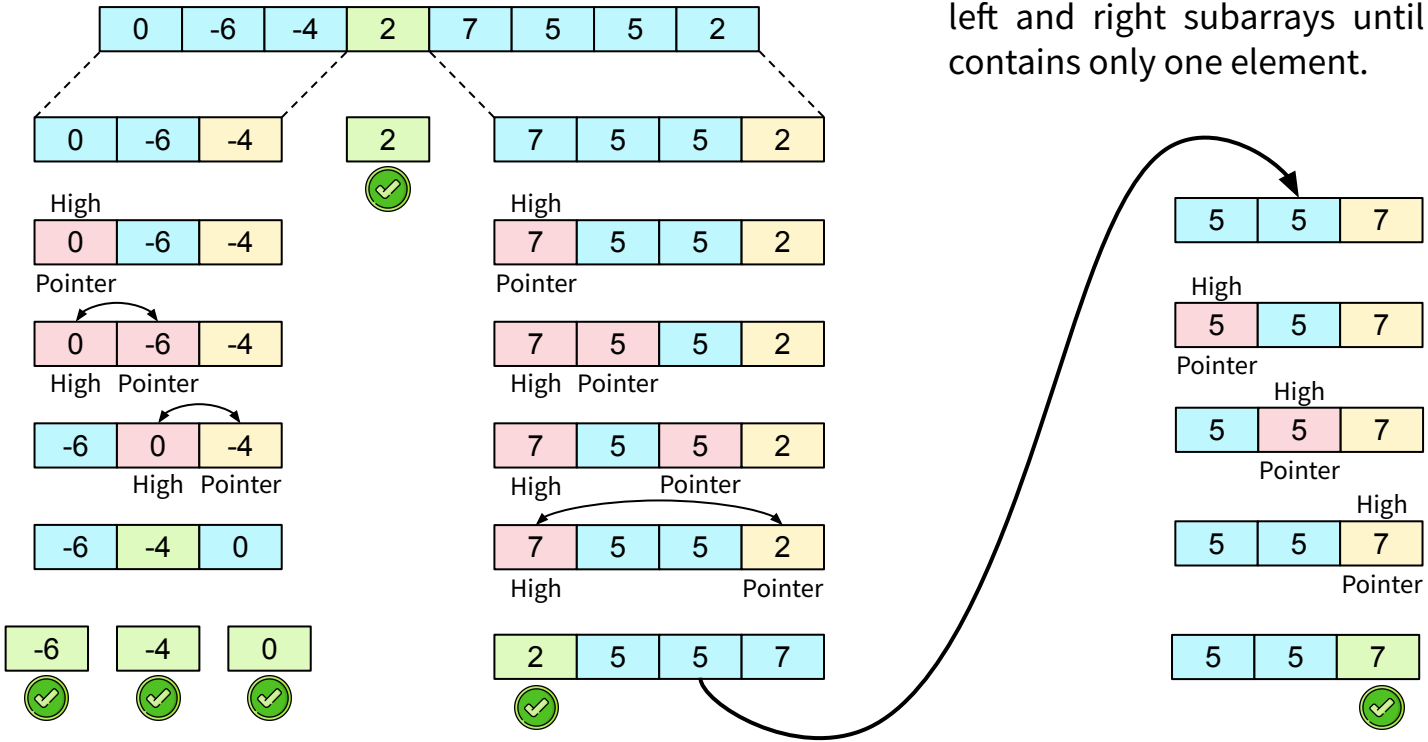
Same (0 is less than 2).

The **pointer** (5) is not less than the **pivot** (2), so the pointer is increased.

The **pointer** (-4) is less than the **pivot** (2), so the pointer is swapped with the high element (5). Then, both the pointer and the high element location are incremented.

The **pointer** is on the same location as the **pivot** (2), so the high element location is swapped with the pivot.

Sorting – Quick Sort



Then, the steps are **recursively** applied to the left and right subarrays until each subarray contains only one element.

It continues for the pivot left subarray ([5,5]) …

Sorting – Quick Sort

The **choice of the pivot** is crucial as it significantly impacts the algorithm's performance.

First/Last Element as Pivot

Simple but can lead to bad performances if the array is already sorted or nearly sorted, as it doesn't guarantee a good split of the array.

Random Element as Pivot

Reduces the probability of experiencing bad performances but does not eliminate it. It offers good performance on average.

Median as Pivot

Leads to a well-balanced split, reducing the chances of bad performance. However, finding the median can be expensive.

Median-of-Three as Pivot

Provides a good balance between computational complexity and ensuring a well-balanced split. It is a commonly used approach.

Exercises

Create a Quick Sort algorithm.

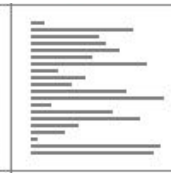
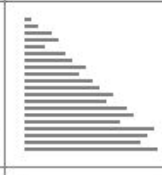
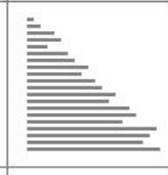
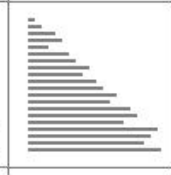
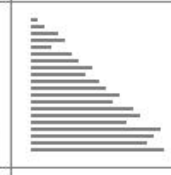
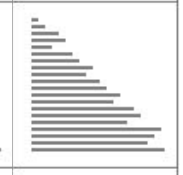
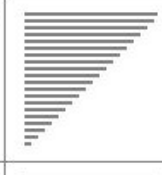
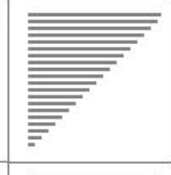
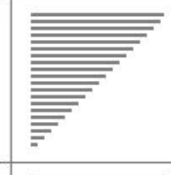
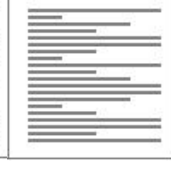
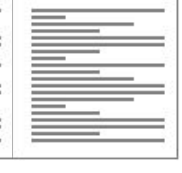
Sort the following arrays:

1. [4, 2, 7, 3, 2, 2, 8]
2. [-1, 3, -5, 2, 0, -8]
3. [5]
4. [0, 1, 2, 3, 4, 5, 6]
5. [5, 5, 5, 5, 5, 5, 5]
6. [6, 5, 4, 3, 2, 1, 0]

Don't search for an existing algorithm online.

Build your own from scratch.

Sorting Algorithms – Efficiency Comparison

		Insertion Sort	Selection Sort	Bubble Sort	Merge Sort	Quick Sort	Quick Sort (median of three)
Random Array	3 7 5 -6 0 4 -4 2						
Nearly Sorted Array	-4 -6 2 0 4 3 5 7						
Reversed Array	7 5 4 3 2 0 -4 -6						
Few Unique Array	7 5 5 2 2 0 2 0						

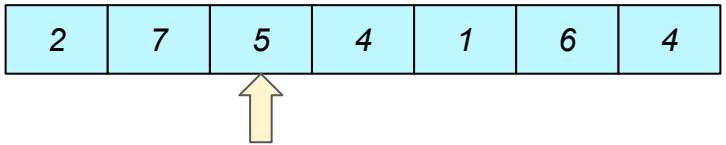
GIF Source : xybernetics.com

Searching

Searching algorithms find specific data among a collection of data. It is used to determine whether a specific data element is present in a data set or to find its location. They can generally be divided into two main categories based on the nature of the data they work with and their approach to finding the target element.

Unsorted Data
Search Algorithms

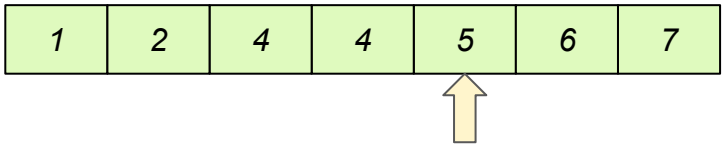
These algorithms are designed to work on datasets where elements are not in any specific order.



The element 5 is at position 2 in the array

Sorted Data
Search Algorithms

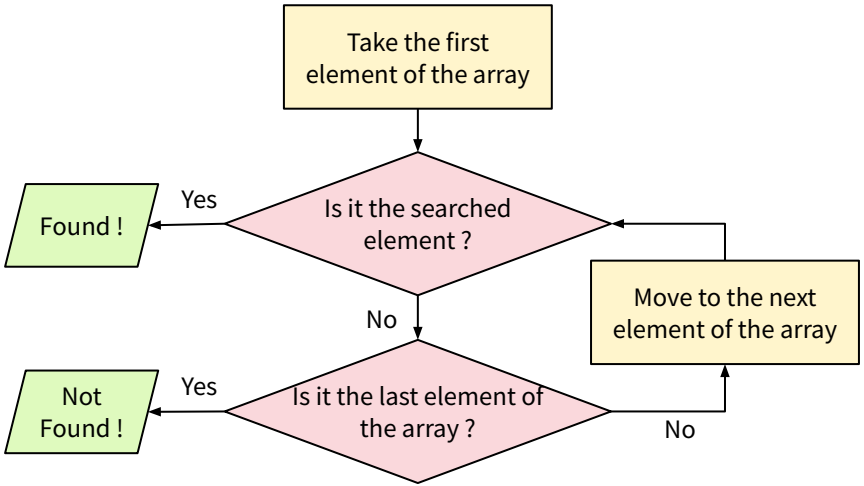
These algorithms are more efficient when the data is presorted in some order such as ascending or descending.

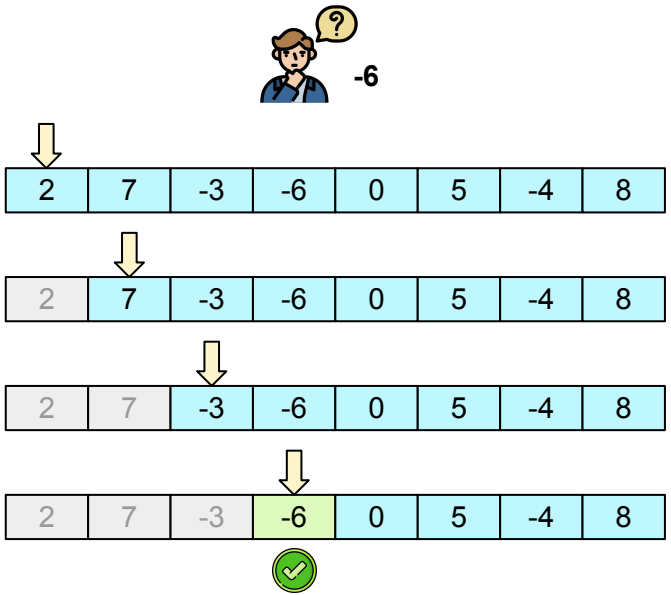


The element 5 is at position 4 in the array

Searching – Linear Search

Linear search or **sequential search** is used to find a particular element in a array by sequentially checking each element of the data structure until it finds the target element, or until it reaches the end of the structure, indicating that the element is not present.





Exercises

Don't search for an existing algorithm online.

Build your own from scratch.

1. **Create a Linear Search algorithm.**
   Return the position of the target element if it exists.

2. **Modify the algorithm to find all the occurrences of the target element.**

In both exercises, find the number 5 in following arrays:

1. [4, 2, -7, 3, -2, 2, 8, -7, 6, 4]
2. [-1, 3, **5**, -2, 0, -8, 3, -5, 4, -9]
3. [-2, 3, 4, -2, 0, -8, 3, -5, 4, **5**]
4. [**5**, 2, -3, 2, 0, -3, 4, -1, 7, 1]
5. [6, 3, **5**, -2, 0, **5**, 3, -5, -4, **5**]

Searching – Binary Search

Binary search is an efficient algorithm for finding an element in a sorted array, especially with large datasets. It operates on the divide-and-conquer principle, repeatedly halving the search space by comparing the target with the middle element of the current range.

However, binary search necessitates that the array be pre-sorted which is not suitable for unsorted data as it would require more computations to sort the array. It's an ideal choice for situations where the overhead of sorting is justified by frequent searches in large datasets.

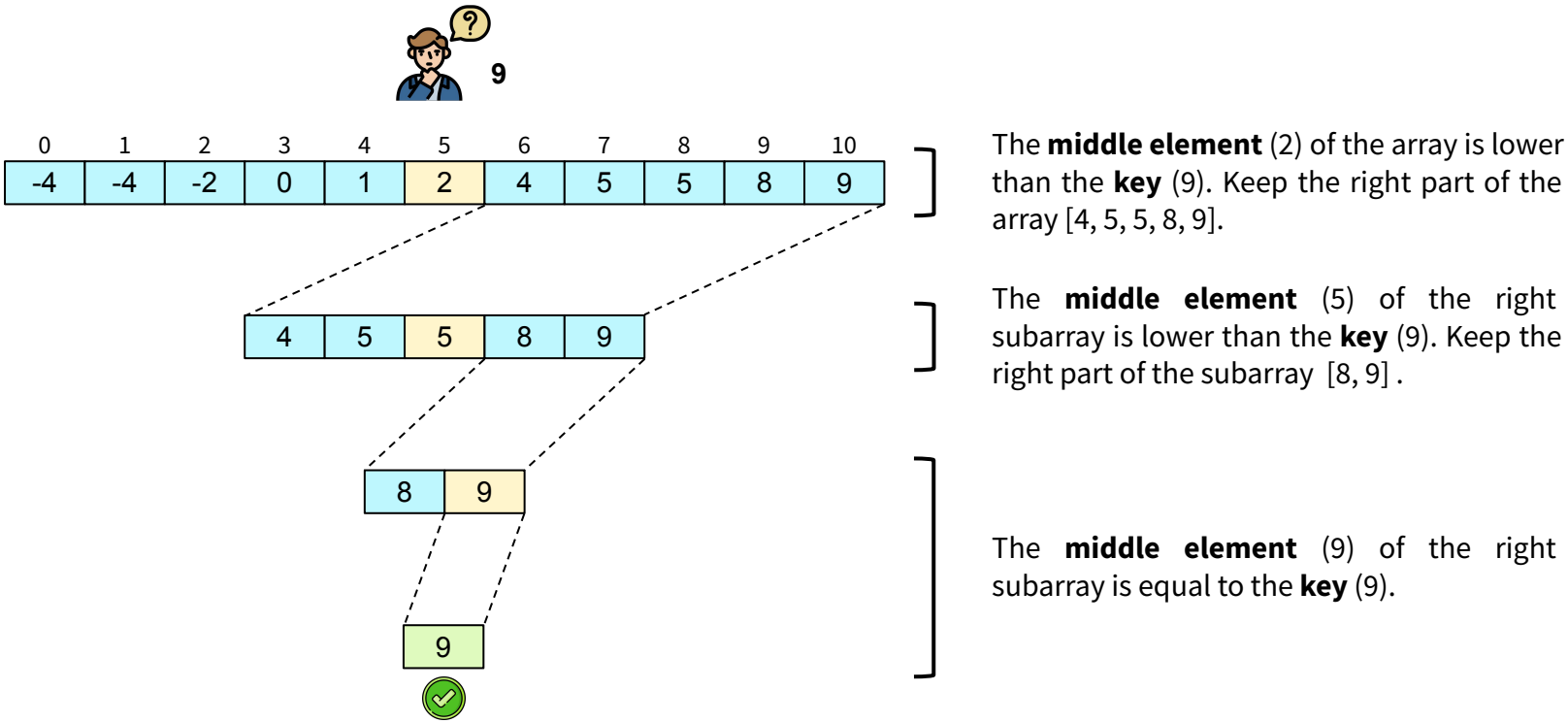
Divide

The Binary Search divide step involves splitting the data set into two halves by locating the middle element of the current segment of the array.

Conquer/Merge

The conquer step is about deciding which half of the array to consider for the next step. This decision is based on a comparison between the target value and the middle element. If the target value is less than the middle element, the search continues in the left half; if greater, in the right half.

Searching – Binary Search



The **middle element** (2) of the array is lower than the **key** (9). Keep the right part of the array [4, 5, 5, 8, 9].

The **middle element** (5) of the right subarray is lower than the **key** (9). Keep the right part of the subarray [8, 9] .

The **middle element** (9) of the right subarray is equal to the **key** (9).

Exercises

Don't search for an existing algorithm online.

Build your own from scratch.

Create a Binary Search algorithm.
Return the position of the element if it exists.

Find the number 5 in following arrays:

1. [-8, -5, -5, -4, -1, 2, 4, 7, 8, 8, 9]
2. [**5**, 6, 8, 8, 8, 10, 11, 11, 15, 16]
3. [-12, -8, -7, -7, -2, -1, -1, 0, 4, **5**]
4. [3, 3, 3, 3, 4, 4, 4, 4, 4, **5**, 6, 6, 6]
5. [-8, -6, -5, -4, -1, 2, 4, **5**, **5**, 8, 9]

(Bonus Exercise) Modify the algorithm to find all the occurrences of the target element.

Binary Search Trees

Trees facilitate several operations including insertion, deletion, traversal, and also **searching** which all depends on the tree type. In this part, we will focus on **Binary Search Trees** (BST) which has the following properties:

Left & Right Children

Each parent node of a BST may have up to two children: the left child and the right child.

Ordered Arrangement

For each node, all elements in the left subtree are less than the node's key, and all elements in the right subtree are greater than the node's key.

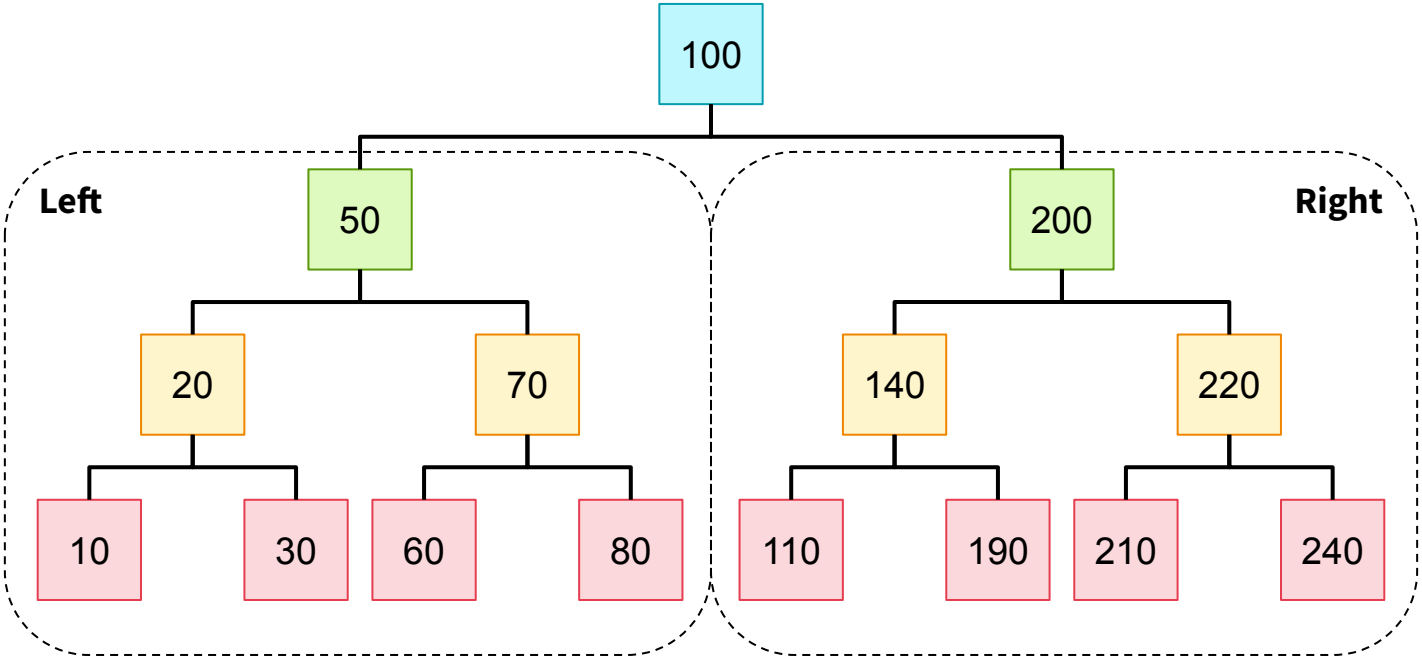
No Duplicates

BSTs typically do not allow duplicate values. This rule can sometimes be lifted.

Fast Operations

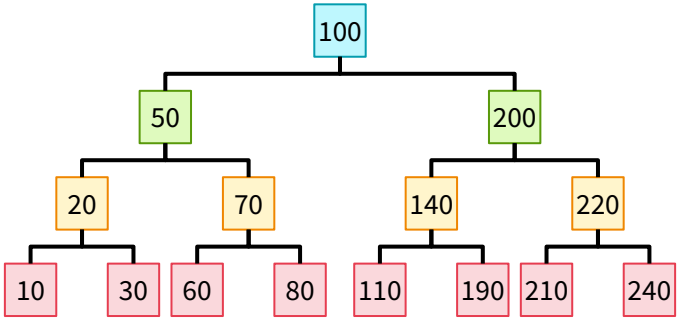
BSTs allow efficient insertion, deletion, searching, and traversal.

Binary Search Trees



All the values of the left subtree are lesser than the root node, and all the values of the right subtree are greater than the root node. This property is propagated to all the subtrees of the subtrees.

Binary Search Tree - Searching for a Value



1. Begin the search at the root node of the BST.
2. Compare the value to be searched with the value of the current node.
   a. If they are **equal**, the search is successful.
   b. If the value is **less than** the current node's value, move to the **left** child.
   c. If the value is **greater than** the current node's value, move to the **right** child.
3. Repeat the comparison process down the tree, choosing left or right child nodes based on the comparison at each step.
4. The value is not in the BST if a leaf node is reached without finding it.

80

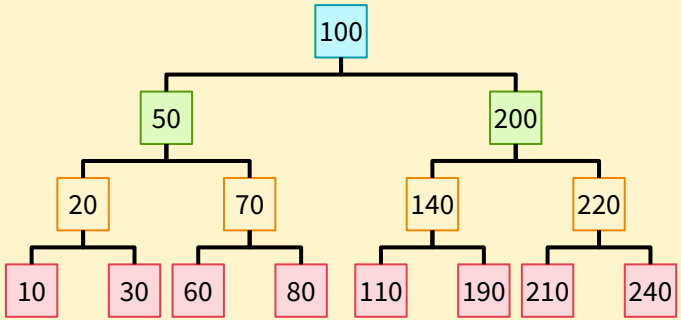
80 < 100 ⇨ ***Go left***

80 > 50 ⇨ ***Go right***

80 > 70 ⇨ ***Go right***

80 = 80 ⇨ ***Found !***

Exercises



Create a BST Searching algorithm.

Use the enclosed tree to search for elements.

Don't search for an existing algorithm online.

Build your own from scratch.

Visiting Each Node in a BST – Traversal

Traversal in a Binary Search Tree is the process of visiting each node in the tree in a particular order. The main traversal two strategies are:

Depth-First Traversal *(DFT)*

Inorder *traversal*

Traverse the **left** subtree, visit the **root** node, and then traverse the **right** subtree. It produces a sorted sequence of the elements in a BST.

Preorder *traversal*

Visit the **root** node, traverse the **left** subtree, and then traverse the **right** subtree. When nodes need to be visited before their children.

Postorder *traversal*

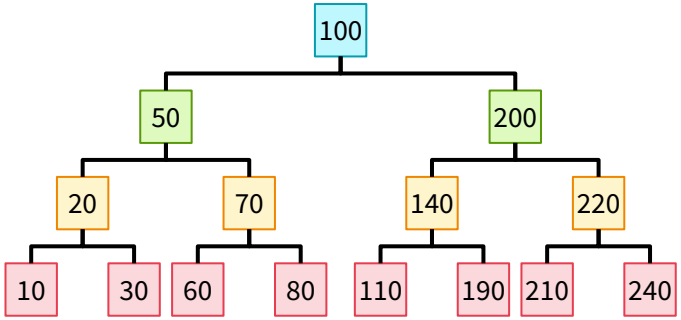
Traverse the **left** subtree, traverse the **right** subtree, and then visit the **root** node. When nodes need to be visited before their parent.

Breadth-First Traversal *(BFT)*

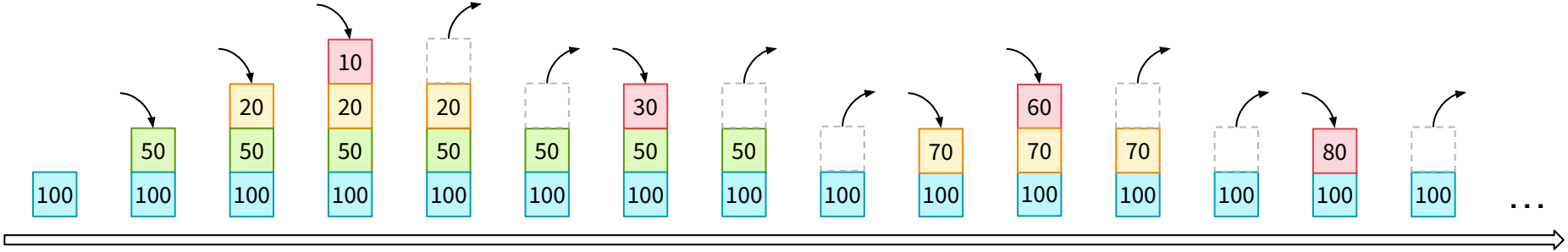
Level-order *traversal*

Visit every node on a level before moving to a lower level. It provides a view of the tree level by level.

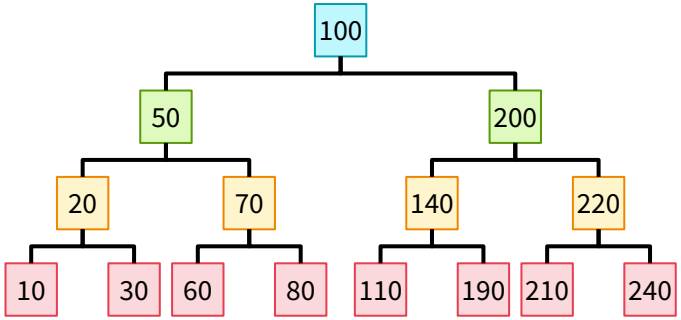
Depth-First-Search – Inorder Traversal



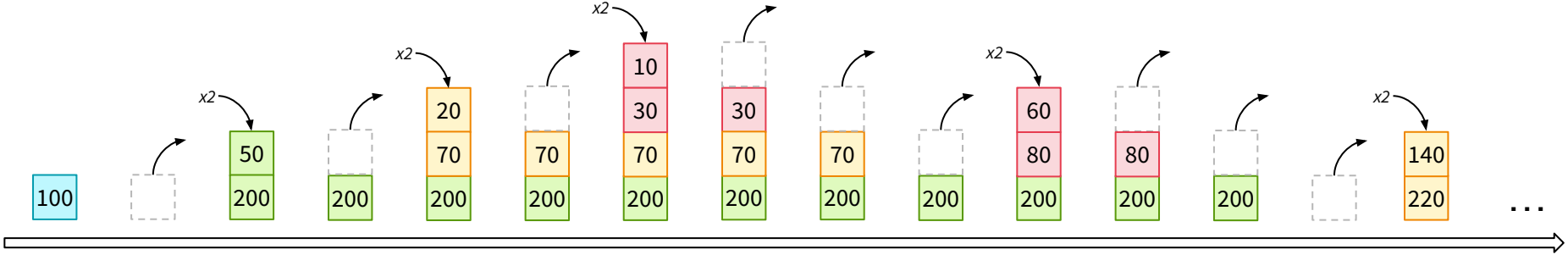
1. Start with the root node. Go to the leftmost node, **pushing** all the nodes encountered along the way onto the stack.
2. **Pop** the top node from the stack and visit it.
3. After visiting a node, if it has a right child, **push** the right child to the stack and then go to its leftmost child, **pushing** all the nodes encountered along the way onto the stack.
4. Repeat steps 2 and 3 until the stack is empty.



Depth-First-Search – Preorder Traversal



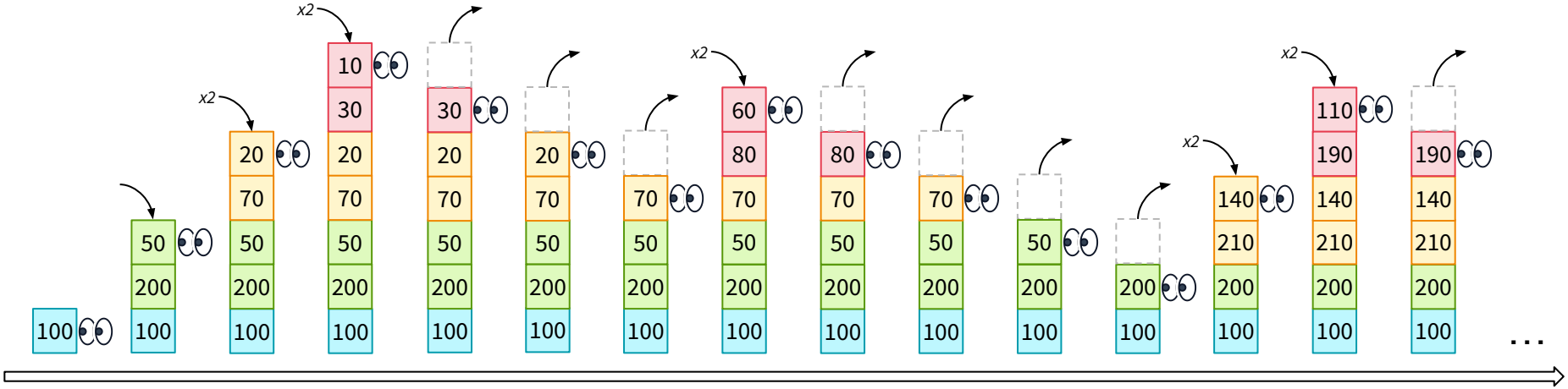
1. **Push** the root node to the stack.
2. **Pop** the top node from the stack and visit it.
   a. If the node has a right child, **push** the right child to the stack.
   b. If the node has a left child, **push** the left child to the stack.
3. Continue the process by repeating steps 2 until the stack is empty.



Depth-First-Search – Postorder Traversal

10 30 20 60 80 70 50 110 190 140 210 240 220 200 100

1. **Push** the root node onto the stack**.**
2. **Peek** 👀 at the top node of the stack (current node).
   a. If it has an unvisited right child, **push** this child onto the stack.
   b. If it has an unvisited left child, **push** this child onto the stack.
   c. If it has no child or its children have been visited, mark it as visited, **pop** it from the stack and add it to the traversal.
3. Repeat step 2 until the stack is empty.



Exercises

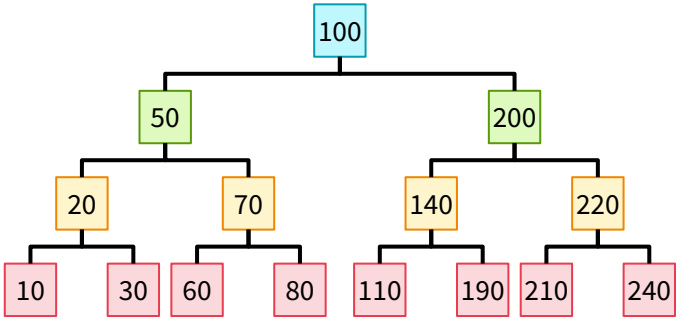
1. **Create an iterative Inorder Traversal algorithm.**
   (Bonus Exercise) Use a recursive approach.
2. **Create an iterative Preorder Traversal algorithm.**
   (Bonus Exercise) Use a recursive approach.
3. **Create an iterative Postorder Traversal algorithm.**
   (Bonus Exercise) Use a recursive approach.

Tips: Use a stack for the iterative algorithms.

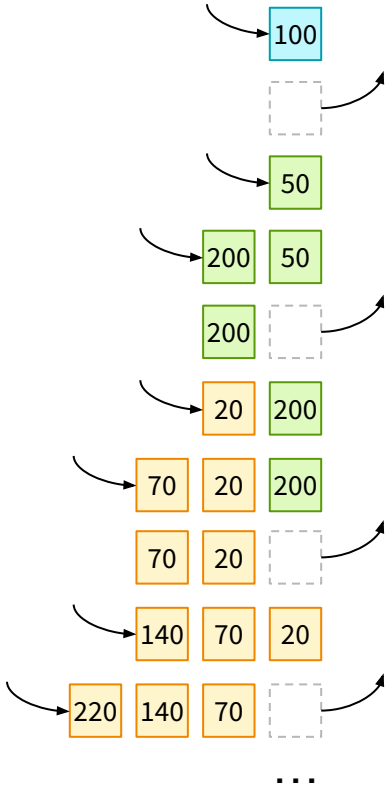
Don't search for an existing algorithm online.

Build your own from scratch.

Breadth-First-Search – Level-Order Traversal



1. Start with the root node. Initialize a queue and **enqueue** the root node.
2. While the queue is not empty:
   a. **Dequeue** the front node from the queue**,** visit it and add it to the traversal.
   b. **Enqueue** the left child of the visited node if it exists.
   c. **Enqueue** the right child of the visited node if it exists.
3. Repeat the process of dequeuing a node and enqueuing its children until the queue is empty.



Exercises

Create a level-order Traversal algorithm.

Tips: Use a queue.

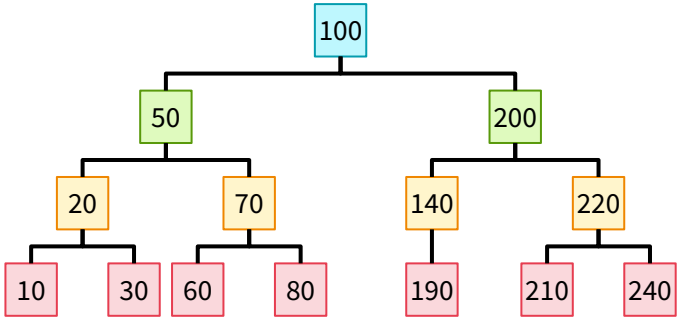
Don't search for an existing algorithm online.

Build your own from scratch.

BST – Node Addition

Adding a node to a Binary Search Tree is similar to searching for a node:

1. Start at the root node.
2. Compare the new and current node values:
   a. If the value of the new node is **less than** the current node's value, move to the left child.
   b. If the value is **greater**, move to the right child.
3. Repeat the step 2 until the instructions ask to move to an empty node position.
4. Insert the new node there.



90

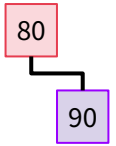
90 < 100 ⇨ ***Go left***

90 > 50 ⇨ ***Go right***

90 > 70 ⇨ ***Go right***

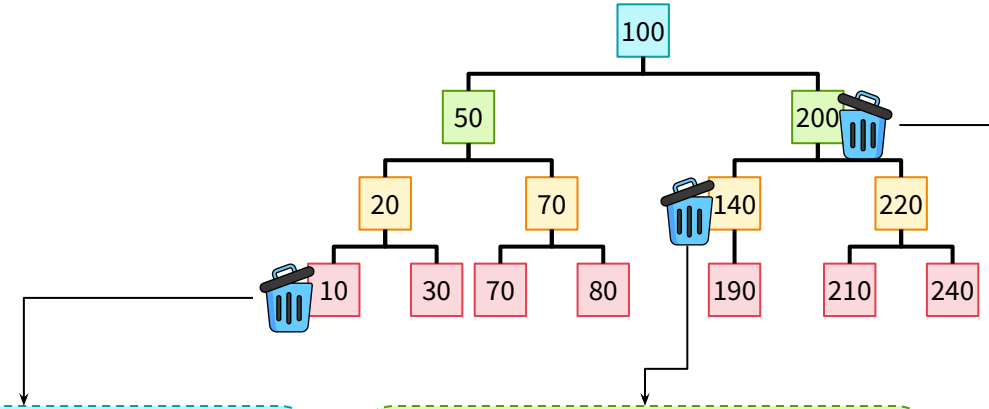
90 > 80 ⇨ ***Go right***

There is no node at the right of 80. Insert the new node here.



BST – Node Removal

Removing a node from a Binary Search Tree is a bit more complex than inserting one, as it involves three main scenarios.



Node with **No Children**

The simplest case: simply remove the node from the tree and update the parent node's information for the child to None.

Node with **One Child**

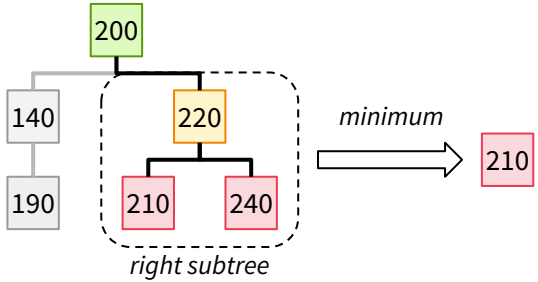
If the node has only one child, remove the node and replace it with its child. Update the parent of the node being deleted to point to the child of the node.

Node with **Two Children**

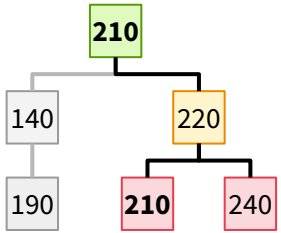
The most complex case:

See next slide

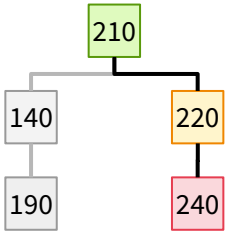
BST – Removal of a Node with Two Children



Step 1 – Look for the smallest node, also called the inorder successor, in the target node's right subtree. It's the node with the smallest value that's larger than the target node's value. To find the smallest number, follow the leftmost path starting from the target.



Step 2 – Copy the value from the inorder successor to the target node.



Step 3 – Remove the inorder successor. It can't have a left child because it's the smallest node in the subtree. Thus, you can remove it using the node with one or no children methods. ***Finally, make sure the tree is correctly restructured*** *after removing the in-order successor.*

Exercises

Create a Node Addition algorithm for a BTS.

(Bonus Exercise) Create a Node Removal algorithm for a BTS.

Don't search for an existing algorithm online.

Build your own from scratch.

Visiting Each Node in a Graph – Traversal

Graph traversal is a method of exploring a graph by visiting each node in a systematic way. There are mainly two methods of graph traversal:

Depth-First Search

This method is **going deep into one path as far as possible before backing up**. Starting at a node, you move along one edge to another node and keep doing this until you can't go any further. Then, you **backtrack** and try a different path.

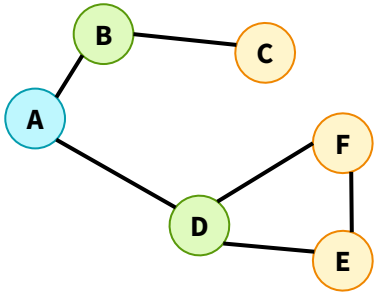
Imagine it like exploring a maze, going down one path completely before trying a different turn.

Breadth-First Search

This method is **exploring all the nearby nodes first before moving on**. Starting at a node, you visit all its immediate neighbors first. Once these are visited, you move on to the neighbors of these neighbors, and so on.

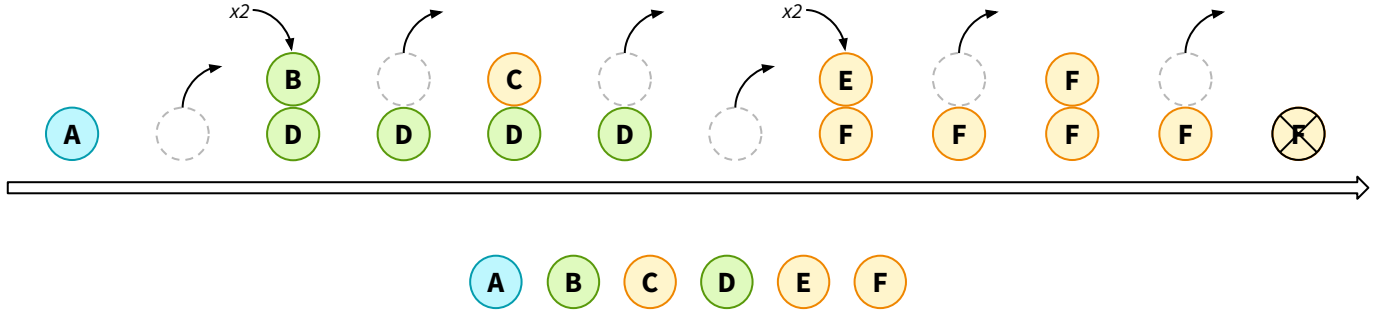
Think of it like a ripple in a pond, where the ripple expands evenly outwards from the point where a stone was dropped.

Graph Traversal – Depth-First Search

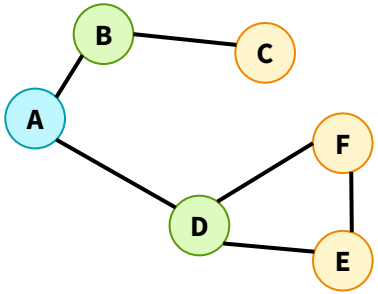


For a given graph, multiple DFS traversal orders exist.

1. **Push** the starting node onto the stack.
2. **Pop** at the top node of the stack (current node).
3. If the current node has not been visited then,
   a. Mark it as visited and add it to the traversal.
   b. **Push** the unvisited adjacent nodes onto the stack.
4. Repeat step 2 and 3 until the stack is empty.

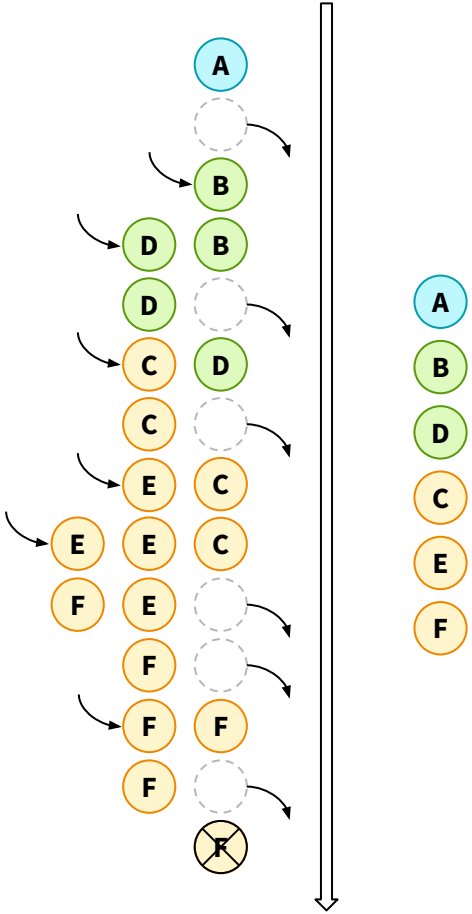


Graph Traversal – Breadth-First Search



For a given graph, multiple DFS traversal orders exist.

1. **Enqueue** the starting node onto the queue.
2. **Dequeue** the front node of the queue (current node).
3. If the current node has not been visited then,
   a. Mark it as visited and add it to the traversal.
   b. **Enqueue** the unvisited adjacent nodes into the queue.
4. Repeat step 2 and 3 until the queue is empty.



Exercises

Which representation of a Graph should be used for a Graph Traversal ?
Adjacency List or Adjacency Matrix ?

1. **Create an <u>iterative</u> Depth-First Search algorithm for a Graph.**
   (Bonus Exercise) Use a recursive approach.
2. **Create an <u>iterative</u> Breadth-First Search algorithm for a Graph.**

Don't search for an existing algorithm online.

Build your own from scratch.

The Shortest Path

In an **unweighted graph**, the shortest path from a single source node to all other node can be found using Breadth-First Search because it explores nodes in order of the number of edges crossed. However, in a **weighted graph**, the shortest path is determined by the total weight of the edges, and not just the number of edges. There are two cases:

Positive Weights Only

In these graphs, every edge has a weight that is greater or equal to 0. The **shortest path from one node to another cannot become shorter by adding more edges**, because each edge adds a non-negative amount to the path length.

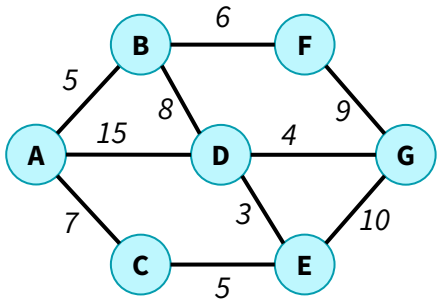
Negative & Positive Weights

These graphs contain edges that can have negative weights. The presence of negative weight edges means that **a path could potentially be shortened by including certain edges**. Thus, finding the shortest path in such graphs is more complex.

For example, negative cycles can theoretically reduce the path length indefinitely.

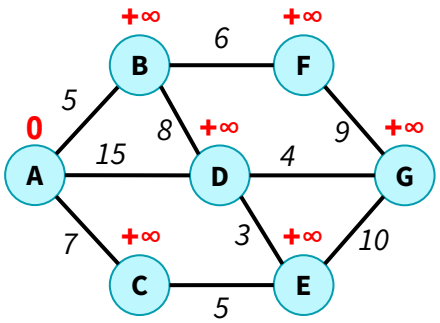
Dijkstra's Algorithm

Dijkstra's algorithm is used for finding the shortest path from a single source node to all other nodes in a graph with **positive edge weights only**.

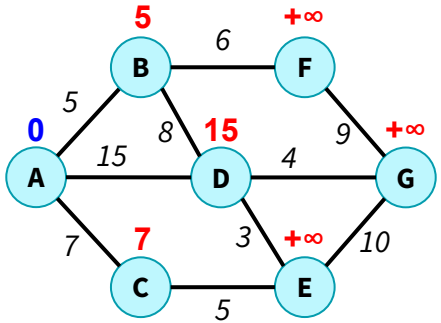


1. It starts by setting the distance to the source node as 0 and all other distances as infinity.
2. Choose the unvisited node with the shortest calculated distance from the start node (current node).
3. Consider all the neighbors of the current node.
   a. Calculate the distance to each neighbor as the sum of the distance to the current node and the edge weight from the current node to that neighbor.
   b. Update the distance to each neighbor if this new calculated distance is less than the previously recorded distance.
4. Mark the current node as visited.
5. Repeat steps 2, 3 and 4 until all the nodes are visited.

Dijkstra's Algorithm



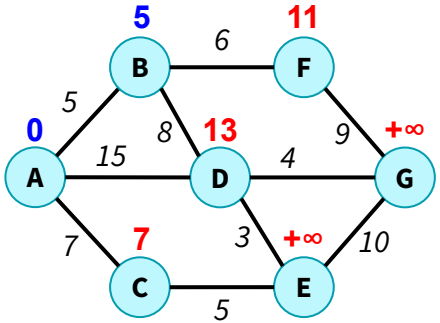
Set the distance to the source node as **0** and all other distances as **+∞**.



The current node is **A** because **0** is the shortest remaining distance to **A**. The unvisited neighbors of **A** are **B**, **D** and **C**.

The distance between **A** and **B** is **0+5** which is lesser than **+∞**. The distance between **A** and **C** is **0+7** which is lesser than **+∞**. The distance between **A** and **D** is **0+15** which is lesser than **+∞**.

A is now considered as visited.

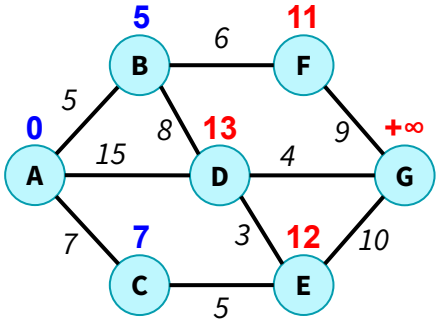


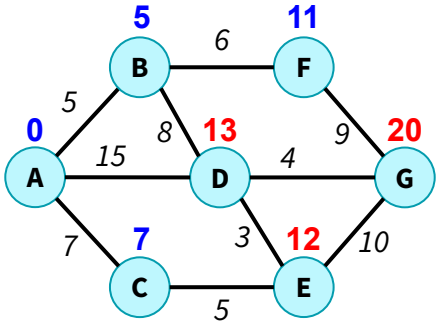
The current node is **B** because **5** is the shortest remaining distance to **A**. The unvisited neighbors of **B** are **D** and **F**.

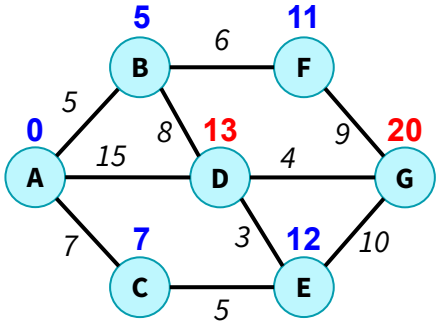
The distance between **A** and **F** (*through B*) is **5+6** which is lesser than **+∞**. The distance between **A** and **D** (*through B*) is **5+8** which is lesser than **15**.

B is now considered as visited.

Dijkstra's Algorithm







The current node is **C** because **7** is the shortest remaining distance to **A**. The unvisited neighbor of **C** is **E**.

The distance between **A** and **E** (through C) is **5+7** which is lesser than **+∞**.

C is now considered as visited.

The current node is **F** because **11** is the shortest remaining distance to **A**. The unvisited neighbor of **F** is **G**.

The distance between **A** and **G** (*through B and F*) is **11+9** which is lesser than **+∞**.

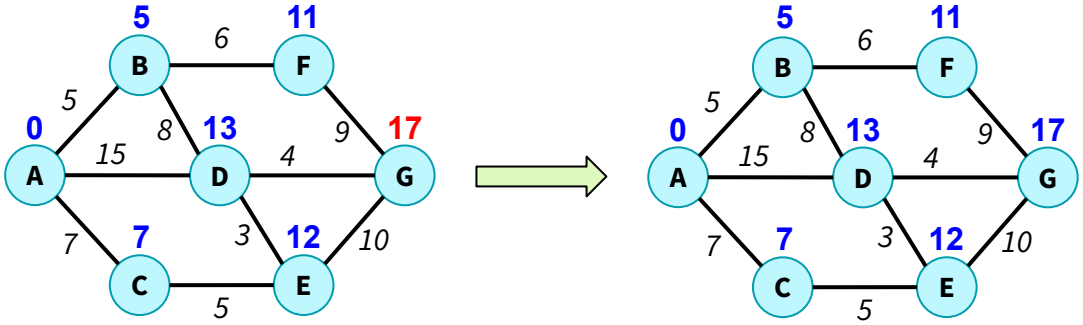
F is now considered as visited.

The current node is **E** because **12** is the shortest remaining distance to **A**. The unvisited neighbors of **E** are **D** and **G**.

The distance between **A** and **D** (*through C and E*) is **12+3** which is greater than **13**. The distance between **A** and **G** (*through C and E*) is **12+10** which is greater than **20**.

E is now considered as visited.

Dijkstra's Algorithm



The current node is **D** because **13** is the shortest remaining distance to **A**. The unvisited neighbor of **D** is **G**.

The distance between **A** and **G** (through B and D) is **13+4** which is lesser than **20**.

D is now considered as visited.

There is only one remaining unvisited node, the algorithm stops.

G is now considered as visited.

The efficiency of the algorithm depends on how quickly the unvisited node with the shortest distance can be found at each step.

This step is generally done using a **simple linear search** by iterating through all the nodes, or a **priority queue**.

Exercises

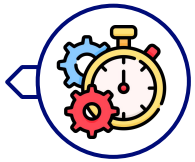
Which representation of a Graph should be used for Dijkstra's Algorithm ?
Adjacency List or Adjacency Matrix ?

Create a Dijkstra's Algorithm using a linear search to find the unvisited node with the shortest distance

Don't search for an existing algorithm online.

Build your own from scratch.

05

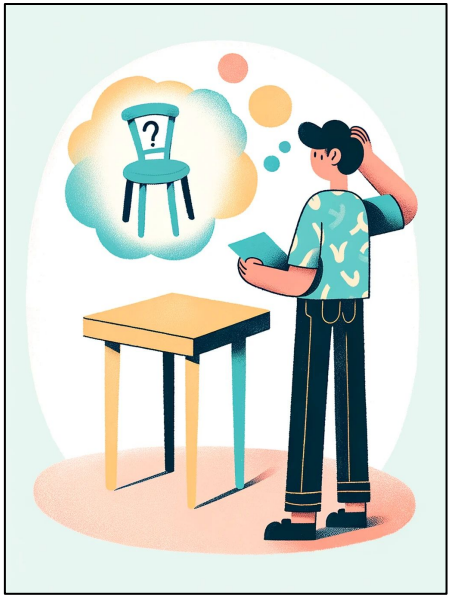


Algorithmic Complexity

Introduction to Algorithms & Data Structures

Complexity

Complexity is used to evaluate the efficiency of algorithms. It comes in two flavors: **time complexity** and **space complexity**.



Time and space complexity can be seen as setting up tables and chairs for guests. If you increase the number of guests, you need more tables and chairs, which takes more time and space to set up.

Time Complexity

It refers to the **time taken by an algorithm to complete its task**. Time complexity is measured in terms of the number of basic operations the algorithm performs – and it's often expressed as a function of the size of the input.

Space Complexity

It refers to the **memory used by an algorithm during its task** including the space for variables, data structures, and function calls. Like time complexity, space complexity is also measured as a function of the size of the input.

Complexity

However, in many algorithms an increase in time complexity doesn't necessarily correlate with an equivalent increase in space complexity.

The **time complexity** is like the time it takes to prepare a meal and the **space complexity** can be thought of as the counter space in your kitchen.

If you're cooking for a few people, it might not take very long. But as the number of guests increases, the preparation time generally increases as well. However, regardless of whether you're cooking for four people or forty you will use roughly the same amount of space to prepare the food.

Best, Average and Worst Case Scenario

Worst, **best**, and **expected case scenarios** are used to describe the efficiency of an algorithm under different conditions. Each scenario provides insight into how the algorithm might perform depending on the nature of the input data. Let's take a simple linear search as example.

7	-3	-6	0	5	-4	8

Best case scenario

It describes the situation where the algorithm is the **most efficient**. In this situation it takes the minimum amount of time or space to complete the task. It is often less practical for real-world application because it represents an ideal condition.

7	-3	-6	**0**	5	-4	8

Expected case scenario

It is often the most practical and realistic as it considers the **average performance** of the algorithm.

Determining the average-case scenario is more complex as it requires understanding the distribution of inputs.

7	-3	-6	0	5	-4	**8**

Worst case scenario

It describes the situation where the algorithm is the **least efficient**. In this situation it takes the maximum amount of time or space to complete the task. Understanding the worst-case scenario ensures that the algorithm can handle the most challenging situations.

Asymptotic Notation

Asymptotic notation is used to describe the efficiency of different algorithms by providing a way to compare their relative efficiency, especially for large input sizes.

Big O Notation

$O(n), O(n^2)$ …

It gives the **upper bound of an algorithm's running time/memory consumption**, showing the maximum amount of time/space the algorithm can take/use.

Big Ω Notation

$\Omega(n), \Omega(n^2)$ …

The big-Ω notation is the opposite of the big-O notation. It gives the **lower bound of an algorithm's running time/memory consumption**.

Big Θ Notation

$\Theta(n), \Theta(n^2)$ …

It shows that the **lower and upper bounds of an algorithm's performance are the same**.

Upper & Lower Bound vs Best & Worst Case Scenario

The concepts of case scenarios (best, worst, expected) and bounds (upper - O, lower - Ω, tight - Θ) in algorithm analysis are **independent**.

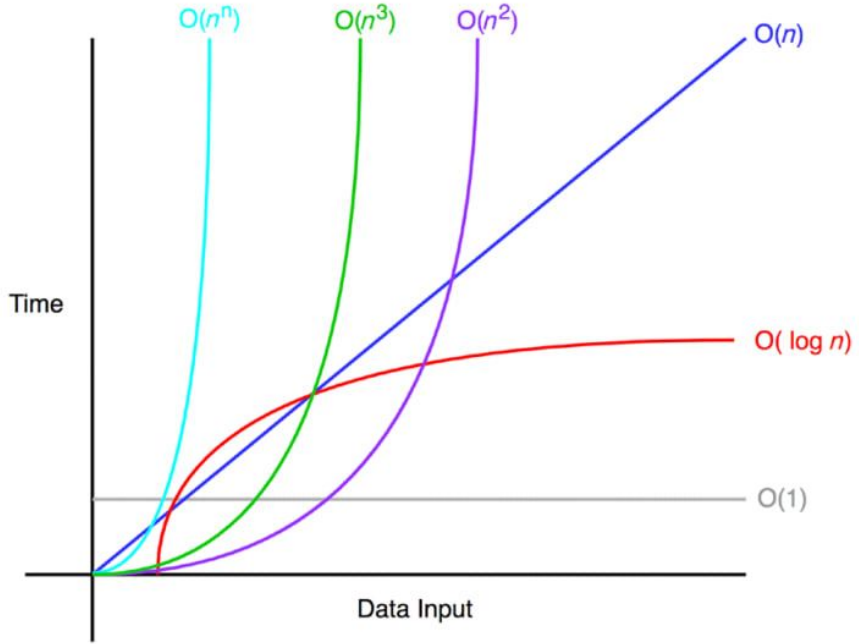
	Best Case	**Worst Case**
Upper Bound	**Upper** Bound of the **Best** Case	**Upper** Bound of the **Worst** Case
Lower Bound	**Lower** Bound of the **Best** Case	**Lower** Bound of the **Worst** Case

*When both the upper bound and the lower bound of the same scenario are the identical, the **bound is said to be tight** (**Big Θ**).*

The upper bound of the worst case refers to the worst performance in the worst possible scenario while the lower bound of the best case refers to the best performance in the best possible scenario.

In practice, the focus is often on the **upper bound of the worst case** and the **lower bound of the best case**, as these are usually the most informative for typical algorithmic performance evaluation.

Asymptotic Notation – Big O



Source : Dev.to

Common Big O Classification

O(***1***)	Constant complexity
O(***log(n)***)	Logarithmic complexity
O(***n***)	Linear complexity
O(***n log(n)***)	Linearithmic complexity
O(n^2)	Quadratic complexity
O(2^n)	Exponential complexity
O(***n!***)	Factorial complexity

Asymptotic Notation – Big O

The primary interest of the Big O is in the rate of growth of the algorithm's running time and memory consumption. Not the exact number of operations. In Big O notation, **constant factors are ignored** because they don't change the overall rate of growth of the complexity. Thus, more attention is given to the variables that change with the input size and less to those that remain constant.

$$O(3n + 5) = O(n)$$

$$O(n/2) = O(n)$$

Additionally, **only the highest order term is considered significant**, as it dominates the growth rate for large values of n. *In the following example, as n gets larger the n^2 term grows much faster than the n term.*

$$O(5n + n^2/2) = O(n^2)$$

Performance Analysis

The complexity of an algorithm can be determined by **analyzing the complexity of each significant part of the code**. Especially, some operations have a higher contribution to the complexity such as **loops**, **recursive calls** or **assignments**.

```python
def linear_search(arr, target):

    for i in range(len(arr)):

        if arr[i] == target:

            return i

    return -1
```

Linear Search Time and Space Complexity

Time Complexity: The best-case scenario is when the target element is the first element of the array. In this case, the algorithm only makes one comparison. Thus, the **best-case time complexity** is **O(1)**. The worst-case scenario occurs when the target element is not in the array. For an array of length n, the algorithm performs n comparisons. Thus, the **worst-case time complexity** is **O(n)**. On average, the algorithm checks half of the elements in the array before finding the target. This also results in a **time complexity** of **O(n)**.

Space Complexity: The algorithm uses a fixed amount of space for the index i and the target variable regardless of the input array size. The **best**, **average** and **worst case complexities** are **O(1)**.

Performance Analysis

```python
def unique_elements(arr):

    unique = set()

    for element in arr:

        unique.add(element)

    return len(unique_elements)
```

Unique Element Time and Space Complexity

Time Complexity: The algorithm iterates through each element of the array once. Moreover, the time complexity of adding an element to a set in Python is generally O(1) because Python sets are implemented as hash tables. Thus, the time complexity is **O(n)**.

Space Complexity: The best-case scenario is when all elements in the array are identical. The set will only contain one element regardless of the array's size. Thus, the best-case space complexity is **O(1)**. The worst-case scenario occurs when all elements in the array are distinct. The set will contain every element in the array. Thus, the worst-case space complexity is **O(n)**. On average, there is a mix of identical and distinct elements. Thus, the space complexity can be anything **between O(1) and O(n)**.

Performance Analysis

```python
def factorial(n):

    if n == 0 or n == 1:

        return 1

    else:

        return n * factorial(n - 1)
```

Factorial Time and Space Complexity

In the recursive factorial algorithm, the notions of best, worst, and average cases don't apply in the same way they do for algorithms where the distribution of elements in an array significantly affect performance.

Here, the complexity is a function of the single input n, and the algorithm's behavior is consistent regardless of the specific value of n excepted for the the base case.

Time Complexity: The function makes n recursive calls with constant amount of time. Thus, the time complexity is **O(n)**.

Space Complexity: Each recursive call is “stored” in a stack. Thus, the size of the call stack is directly proportional to n and the space complexity is **O(n)**.

Python's built-in Data Structure Performance

	Adding	Inserting	Removing	Accessing	Modifying	Copying
List	O(1)	O(n)[1]	O(n)[2]	O(1)[3]	O(1)	O(n)
Tuple	*N/A*	*N/A*	*N/A*	O(1)[3]	*N/A*	*N/A*
Dictionary[4]	O(1)	*N/A*	O(1)	O(1)	O(1)	O(n)
Set	O(1)	*N/A*	O(1)	*N/A*	*N/A*	O(n)

Time Complexities of different python method on List, Tuple, Dictionary and Sets

[1] *because it requires shifting elements after the addition.*

[2] *.pop() → it may need to shift elements after the removal. .remove() → it searches the element first and then removes it.*

[3] *slicing is O(k) where k is the number of elements in the slice.*

[4] *.items(), .keys() and .values() are O(1) but O(n) when converted to lists*

Amortized Analysis

Amortized analysis determines the average time/space consumption per operation over a sequence of operations. It is particularly useful in situations where an occasional operation might be costly but doesn't occur frequently enough to dominate the overall performance of the algorithm. There are three main methods: the **aggregate analysis**, the **accounting method** and the **potential method**.

Aggregate Analysis

Calculates the total time for a sequence of operations and divides it by the number of operations. This approach gives an average time per operation.

Accounting Method

Assigns different costs to different operations where some operations are charged more than their actual cost. This extra charge is used to pay for other operations that might require more time.

Potential Method

It is like a battery. It starts empty and charges when "simple" operations are performed on the data structure. On the contrary, "complex" operations consume the energy accumulated by simple ones.

Amortized Analysis – Dynamic Array

A classic example of amortized analysis is the dynamic array such as **Python's list**. When elements are added to a list using the append operation in Python it occasionally needs to resize itself to accommodate more elements. During this resizing, the list allocates a new array and copies all elements from the old array to the new one. This operation takes O(n) time.

	Adding
List *Regular Case*	**O(1)**
List *Resizing Case*	**O(n)**

Let's use the `.append()` method **n** times:

- the cost for **n** additions is **n**.
- Say that the resizing happens at power of two such as

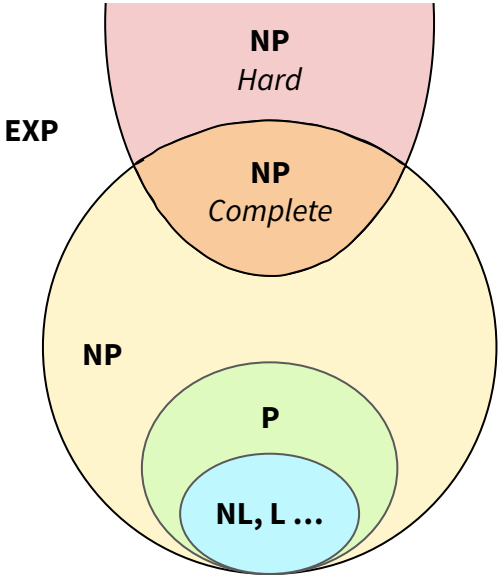
1, 2, 4, …, n/2, n

- Thus, the total cost to resize the list is **1 + 2 + 4 + … + n/2 + n** which is approximately **2n** (upper bound).

The amortized cost per addition in the list is the average over n operations which is less than **(n + 2n) / n = 3** and thus O(1).

Complexity Classes

Complexity classes are categories used to classify algorithms based on the resources they require.



	Definition
P *Polynomial Time*	Problems in P are considered efficiently solvable (polynomial time).
NP *Nondeterministic Polynomial Time*	Problems in NP are considered efficiently verifiable (polynomial time) but not always efficiently solvable.
NP-Complete	Problems in NP-complete are among the hardest in NP. If a solution is found to solve an NP-complete problem, all problems in NP could be solved efficiently.
NP-Hard	Problems in NP-hard are at least as hard as the hardest problems in NP. They are even tougher than NP-Complete problems.
EXP *Exponential Time*	Problems in EXP are considered solvable in an exponential time.

Timeit

The **timeit module** is part of the Python Standard Library. It is a tool for measuring the execution time. It works by executing a code repeatedly to get a more accurate measurement of the execution time. This repetition helps to mitigate the impact of other processes on the system that might affect the timing.

Testing a function's execution time can be done by passing the function and its arguments to `timeit.timeit()`. There are three key parameters:

- ***stmt*** *which the function to be tested (in a string or a callable format).*
- ***setup*** *which is the code that runs before stmt in order to not be included in the time measurement (imports …).*
- ***number*** *which is the number of time stmt is executed.*

The `timeit.repeat()` function can be used to repeat the `timeit.timeit()` function and get a more accurate measurement.

```python
import timeit

# Method 1
setup="""
n = 20
"""
stmt = """
fact = 1
for num in range(2, n + 1):
    fact *= num
"""
duration = timeit.timeit(stmt = stmt,
                         setup = setup,
                         number = 100000)

# Method 2
def factorial(n):
    fact = 1
    for num in range(2, n + 1):
        fact *= num
    return fact

duration = timeit.timeit(stmt = lambda: factorial(20),
                         number = 100000)
```

Exercises

Now, it's time to **practice**!

Complete the following **exercise sheet**:

- 13 - (EN) Exercises - Complexity