What is an algorithm?

Definition

*An algorithm is a **set of unambiguous instructions** designed to solve a problem.*

History

The earliest algorithms, originating from the name **Mūsā al-Khwārizmī**, a Persian mathematician from the 9th century. For more information, visit https://mathematical-tours.github.io/algorithms/.

Back to ancient civilizations, such as the Egyptians and Babylonians, developed algorithms for **basic arithmetic operations**, like addition and multiplication. Euclid's algorithm, developed around 300 BCE, is **one of the earliest known algorithms** and is used to find the greatest common divisor (GCD) of two numbers.

Question

- Are you aware of any algorithm?

Question

- Are you aware of any algorithm?

- Do you know how they work?

- Do you think they work perfectly?

- Can they be biased or make non-optimal decisions?

Notes

- The representation (or sometimes translation) into a programming language is not reciprocal: **not every program is an algorithm.**

- For example, reactive programs (handling input/output) or those containing animations do not terminate because they are always waiting for input. They do not constitute algorithms in the strict sense.

- Algorithms are language-agnostic; they describe the logic and steps needed to solve a problem, but not the specific coding details.

Example: Euclid's algorithm

One of the earliest algorithm: Euclid's algorithm to compute the greatest common divisor of two integers a and b:

Example: Euclid's algorithm

One of the earliest algorithm: Euclid's algorithm to compute the greatest common divisor of two integers a and b:

```python
def gcd(a, b):
    while b != 0:
        t = b
        b = a % b
        a = t
    return a

gcd(10, 20) # 10
```

10

Example: Euclid's algorithm

One of the earliest algorithm: Euclid's algorithm to compute the greatest common divisor of two integers a and b:

In [1]:
```python
def gcd(a, b):
    while b != 0:
        t = b
        b = a % b
        a = t
    return a

gcd(10, 20) # 10
```

Out[1]:   10

In [2]:
```python
assert gcd(12, 18) == 6   # GCD of 12 and 18 is 6
assert gcd(1071, 462) == 21   # GCD of 1071 and 462 is 21
assert gcd(0, 8) == 8   # GCD of 0 and 8 is 8
assert gcd(25, 0) == 25   # GCD of 25 and 0 is 25
assert gcd(-12, 18) == 6   # GCD of -12 and 18 is 6
```

How do you check an algorithm is correct?

- **Mathematical Proof:** a formal and rigorous method of demonstrating that an algorithm is correct.
- **Code Review:** a collaborative process where one or more peers review the code implementation of an algorithm.
- **Test Cases:** sets of inputs and expected outputs used to validate that an algorithm produces correct results.

For **test cases:**, the `assert` statement is used to check whether a given condition evaluates to `True`, then the program continues to execute normally. If the condition is `False`, an `AssertionError` exception is raised, and the program stops executing.

How do you check an algorithm is correct? (cont.)

In [3]:

```python
def add(a, b): # function to test
    return a + b

assert add(2, 3) == 5, "Test Case 1 Failed"
assert add(-1, 1) == 0, "Test Case 2 Failed"
assert add(0, 0) == 0, "Test Case 3 Failed"
assert add(10, -5) == 5, "Test Case 4 Failed"

print("All test cases passed!")
```

```
All test cases passed!
```

Exercice: x power n

An algorithm (and tests) that calculates :

Exercice: x power n

An algorithm (and tests) that calculates :

```python
def puissance(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        temp = puissance(x, n // 2)
        return temp * temp
    elif n < 0:
        temp = puissance(x, -(n + 1) // 2)
        return 1 / (temp * temp * x)
    else:
        temp = puissance(x, (n - 1) // 2)
        return temp * temp * x
```

Exercice: x power n

An algorithm (and tests) that calculates :

In [4]:
```python
def puissance(x, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        temp = puissance(x, n // 2)
        return temp * temp
    elif n < 0:
        temp = puissance(x, -(n + 1) // 2)
        return 1 / (temp * temp * x)
    else:
        temp = puissance(x, (n - 1) // 2)
        return temp * temp * x
```

In [5]:
```python
assert puissance(2, 3) == 8
assert puissance(5, 0) == 1
assert puissance(3, -2) == 1/9
assert puissance(2, 10) == 1024
assert puissance(2, -3) == 1/8
assert puissance(2, 1) == 2
```

Exercice: The sum of the first n integers

An algorithm (and tests) that calculates :

Exercice: The sum of the first n integers

An algorithm (and tests) that calculates :

```python
def sum_n(n):
    return n*(n+1)/2

assert sum_n(1) == 1   # 1
assert sum_n(2) == 3   # 1 + 2
assert sum_n(3) == 6   # 1 + 2 + 3
assert sum_n(4) == 10  # 1 + 2 + 3 + 4
assert sum_n(5) == 15  # 1 + 2 + 3 + 4 + 5
assert sum_n(1000) == 500500 # ..
```

Exercice: Leap year

Write a function `is_leap_year` that takes a year as input and returns `True` if it's a leap year and `False` otherwise. The function follows the rules for leap year determination:

- A year that is divisible by 4 is a leap year.
- However, a year that is divisible by 100 is not a leap year, unless...
- The year is also divisible by 400, in which case it is a leap year.

E.g 2000 is a leap year, 2020 is a leap year.

Exercice: Leap year (cont.)

Exercice: Leap year (cont.)

In [7]:
```python
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False

test_years = [2020, 2100, 2400]

for year in test_years:
    if is_leap_year(year):
        print(f"{year} is a leap year.")
    else:
        print(f"{year} is not a leap year.")
```

```
2020 is a leap year.
2100 is not a leap year.
2400 is a leap year.
```

Exercice: Leap year (cont.)

In [7]:
```python
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False

test_years = [2020, 2100, 2400]

for year in test_years:
    if is_leap_year(year):
        print(f"{year} is a leap year.")
    else:
        print(f"{year} is not a leap year.")
```

```
2020 is a leap year.
2100 is not a leap year.
2400 is a leap year.
```

Another possible test: compare to the Python isLeap from the `calendar` module.

Exercice: Leap year (cont.)

In [7]:
```python
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False


test_years = [2020, 2100, 2400]

for year in test_years:
    if is_leap_year(year):
        print(f"{year} is a leap year.")
    else:
        print(f"{year} is not a leap year.")
```

```
2020 is a leap year.
2100 is not a leap year.
2400 is a leap year.
```

Another possible test: compare to the Python isLeap from the `calendar` module.

In [8]:
```python
import calendar

def is_leap_year(year):
    return calendar.isleap(year)
```

Exercice: Find a number in a list

Given a list of integer, return a specific number provided as parameter

Exercice: Find a number in a list

Given a list of integer, return a specific number provided as parameter

In [9]:
```python
def search_element_in_list(element, list):

    for i in list:
        if i == element:
            return True
    return False

element_list = [1, 2, 3, 4, 5]
element_to_find = 3
result = search_element_in_list(element_to_find, element_list)
assert result == True, f"Expected True, but got {result}"
```

Exercice: Find a number in a list

Given a list of integer, return a specific number provided as parameter

In [9]:
```python
def search_element_in_list(element, list):

    for i in list:
        if i == element:
            return True
    return False

element_list = [1, 2, 3, 4, 5]
element_to_find = 3
result = search_element_in_list(element_to_find, element_list)
assert result == True, f"Expected True, but got {result}"
```

Another type of test is to compare with a built-in Python function:

In [10]:
```python
def search_element_in_list_python(element, lst):
    return element in lst

assert search_element_in_list(element_to_find, element_list) == search_
```

Algorithms properties

Properties

An algorithm possesses the following properties (among others):

- Communicable
- Efficient
- Complete, terminates, and correct
- Deterministic

Communicate algorithms

There are different ways to write algorithms. There is no optimal one, it depends on the context. Examples of contexts are:

- Plain language (pseudo-code)
- Formalization such as an equation
- A software specification
- Implementation in a programming language

Plain language (pseudo-code)

The pseudocode is a way to write algorithms in a human-readable way. It is not a programming language, but it is close to it. It is a way to communicate algorithms. E.g. for Euclid's algorithm:

- Divide a by b, and you get the remainder r.
- Replace a with b.
- Replace b with r.
- Continue as long as it's possible; otherwise, you get the GCD (Greatest Common Divisor).

or

```
function gcd(a, b)
    while b ≠ 0
        t := b;
        b := a mod b;
        a := t;
    return a;
```

Equation

You can use mathematical equations and notations to describe certain aspects of the algorithm's behavior or to express mathematical relationships within the algorithm.

- $\sum_{i=1}^{n} x_i$

- $Fn = Fn - 1 + Fn - 2$

- $\mu = (\Sigma x) / N$

- $PR_{t+1}(P_i) = \sum_{P_j} \frac{PR_t(P_j)}{C(P_j)}$

Graphics

Graphical representations of algorithms are visual ways to illustrate the flow, logic, and structure of an algorithm. They are often used to aid in understanding, designing, and communicating algorithms, especially in algorithm design and computer science education. There are various types of graphical representations, and the choice depends on the complexity and purpose of the algorithm.
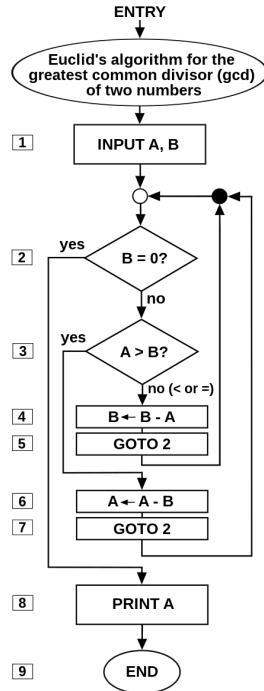
Code (Python)

Code (Python, Java, ..); example in Python:

```python
def gcd(a, b):
    while b != 0:
        t = b
        b = a % b
        a = t
    return a
```

In Java:

```java
public class GCD {
    public static int gcd(int a, int b) {
        while (b != 0) {
            int t = b;
            b = a % b;
            a = t;
        }
        return a;
    }
}
```

Discussion on the type of representation

There are different ways to express an algorithm, depending on the context and the level of formalization required.

- **Graphical representation** is more accessible and provides an overview, allowing for the detection of errors, patterns, etc. Humans have better perception abilities in the visual space than in text.

- **Pseudo-language** has the characteristic of being flexible, close to both human and computer languages, and independent of a programming language. However, it is often defined ambiguously and requires additional effort for implementation.

- Finally, **implementation (e.g., Python)** has the advantage of being immediately testable. However, it can be very strict (must be correct) and sometimes challenging to read if one is not familiar with the language. This also depends on the programmer.

Efficiency

> *An algorithm is considered **efficient** if it minimizes the consumption of resources required to perform it.*

Efficiency is relative to various criteria (values we want to measure) that need to be calculated (theoretically) or measured (empirically) in order to understand what is happening. Note that it is necessary to use large values of n to obtain a representative behavior. Among these criteria:

- Execution time

- Required memory space

- Disk storage space

- Etc.

We will see later that the concept of **Complexity** is based on one of these criteria and allows independence from the technology used (language, computer, compiler, etc.).

Example:

In genomics, it is common to compare two sequences (of genes) of lengths N and M (e.g., `TAG CAC` and `TGC TTG`).

- The number of comparisons is $N \times M$.

- If the size of the sequences doubles, then the number of comparisons... quadruples!

- $(2 \times N) \times (2 \times M) = 4 \times (N \times M)$.

- Now, if we want to align 3 sequences, it becomes N^3.

In practice, it becomes challenging to find a solution quickly (especially when comparing more than 2 sequences).

$\rightarrow$ The same applies to long sequences.

$\rightarrow$ Therefore, it is necessary to have an efficient algorithm (in the case of sequence comparison, consider the BLAST algorithm (Basic Local Alignment Search Tool)).

Other properties

Other qualities of an algorithm (beyond being simple and understandable):

Completeness: *An algorithm must be complete, meaning that for a given problem, it provides a solution for each of the inputs.*

Termination: *An algorithm must terminate within a finite time.*

Correctness: *An algorithm must be correct and terminate by providing a result that is the solution to the problem it is supposed to solve.*

→ All of this is very difficult to prove (formal proof, etc.)!

Algorithms patterns

An algorithm has a **pattern**, which is a way to classify algorithms based on their properties.

- There are several ways to design algorithms, either based on performance constraints or based on the structural style.

- There is not a single unique algorithm for a given problem.

Examples of patterns (main ones):

- By purpose
- By implementation (e.g., **recursion**, functional, etc.)
- By **design paradigm** (Divide and Conquer, etc.)
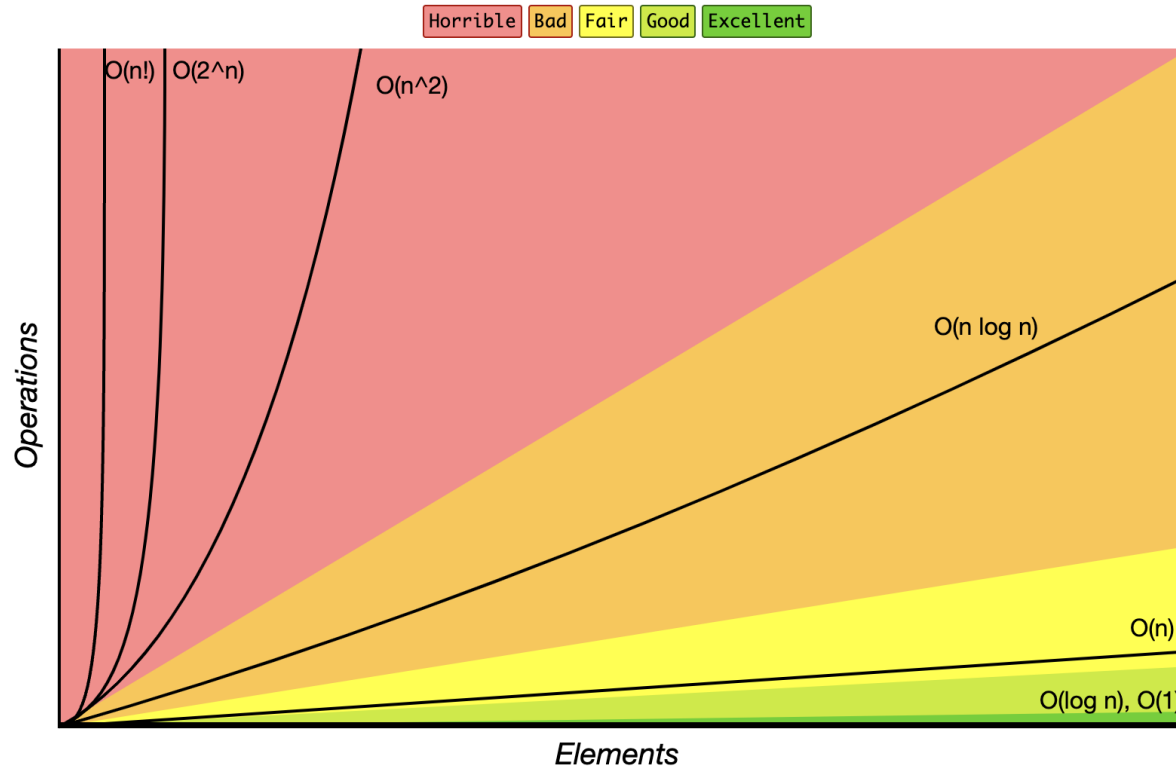- By **complexity**

Complexity

What is complexity?

> The **complexity of an algorithm** is the formal estimation of the amount of resources required to execute an algorithm. These resources can include time, memory space, storage, etc.

There are different types of complexity:

- **Best Case:** The *smallest* number of operations the algorithm will have to execute on a dataset of a fixed size.

- **Worst Case:** This is the *largest* number of operations the algorithm will have to execute on a dataset of a fixed size.

- **Average Case:** This is the *average* of the algorithm's complexities on datasets of a fixed size.

Note: It is often the worst-case analysis that is chosen (provides an upper performance limit). The complexity in terms of the number of operations is typically the most studied.

Big-O Complexity Chart

Horrible Bad Fair Good Excellent

O(n!) O(2^n) O(n^2)

O(n log n)

O(n)

O(log n), O(1)

Operations

Elements

Exercice: find the complexity

```python
def maximum(L):
    m=L[0]
    for i in range(1,len(L)):
      if L[i]>m:
        m=L[i]
    return m
```

Exercice: find the complexity

```python
def maximum(L):
    m=L[0]
    for i in range(1,len(L)):
      if L[i]>m:
        m=L[i]
  return m
```

(goes through the whole list in the worst case scenario)

Intuition behind the complexity calculation

Notation	Complexity	Intuition
$\mathcal{O}1$	Constant	First or nth element of a list, ...
$\mathcal{O}logn$	Logarithmic	Divide in half and repeat, ...
$\mathcal{O}n$	Linear	Traverse data, ...
$\mathcal{O}nlogn$	Quasi-Linear	Divide in half and combine, ...
$\mathcal{O}n^2$	Quadratic	Traverse data with 2 loops, ...
$\mathcal{O}2^n$	Exponential	Test all combinations, ...
$\mathcal{O}n^k$, k >2	Polynomial	Traverse data with k loops, ...
$\mathcal{O}n!$	Factorial	Test all paths (graph), ...

Exercice: find the complexity

In [11]:
```python
def nocc(x,L):
    n=0
    for y in L:
        if x==y:
            n=n+1
    return n
```

Exercice: find the complexity

```
In [11]:   def nocc(x,L):
               n=0
               for y in L:
                   if x==y:
                       n=n+1
               return n
```

(goes through the whole list in the worst case scenario)

Exercice: find the complexity

```
In [12]:
def maj(L):
    xmaj=L[0]
    nmaj=nocc(xmaj,L)
    for i in range(1,len(L)):
        if nocc(L[i],L)>nmaj:
            xmaj=L[i]
            nmaj=nocc(L[i],L)
    return xmaj
```

Exercice: find the complexity

```
In [12]: def maj(L):
           xmaj=L[0]
           nmaj=nocc(xmaj,L)
           for i in range(1,len(L)):
             if nocc(L[i],L)>nmaj:
               xmaj=L[i]
               nmaj=nocc(L[i],L)
           return xmaj
```

Exercice: find the complexity

The complexity of an `is_even(n)` algorithm that takes an integer `n` as input and returns `True` if n is an even number and `False` otherwise.

Exercice: find the complexity

The complexity of an `is_even(n)` algorithm that takes an integer `n` as input and returns `True` if n is an even number and `False` otherwise.

In [13]:
```python
def is_even(n):
    return n % 2 == 0
```

Exercice: find the complexity

The complexity of an `is_even(n)` algorithm that takes an integer `n` as input and returns `True` if n is an even number and `False` otherwise.

In [13]:
```python
def is_even(n):
    return n % 2 == 0
```

Exercice: find the complexity

In [14]:
```python
def somcubes(n):
    s = 0
    while n>0:
        s = s+(n%10)**3
        n = n//10
    return s


def eq_somcubes(N):
  L = []
  for n in range(0, N+1):
    if n==somcubes(n):
      L.append(n)
    return L
```

Exercice: find the complexity

In [14]:
```python
def somcubes(n):
    s = 0
    while n>0:
        s = s+(n%10)**3
        n = n//10
    return s


def eq_somcubes(N):
  L = []
  for n in range(0, N+1):
    if n==somcubes(n):
      L.append(n)
    return L
```

(we seek numbers that are equal to the sum of the cubes of their digits).

Exercice: find the complexity

You have two sorted lists, `[1, 3, 8, 10]` and `[2, 3, 9]`, and you want to obtain a new merged list from these two lists (without using sorting functions like sort or sorted). What is the complexity?

Exercice: find the complexity

You have two sorted lists, `[1, 3, 8, 10]` and `[2, 3, 9]``, and you want to obtain a new merged list from these two lists (without using sorting functions like sort or sorted). What is the complexity?

We iterate through all the data once: .

```python
In [15]:  def merge_sorted_lists(list1, list2):
              merged_list = []
              i = j = 0

              while i < len(list1) and j < len(list2):
                  if list1[i] < list2[j]:
                      merged_list.append(list1[i])
                      i += 1
                  else:
                      merged_list.append(list2[j])
                      j += 1

              while i < len(list1):
                  merged_list.append(list1[i])
                  i += 1

              while j < len(list2):
                  merged_list.append(list2[j])
                  j += 1

              return merged_list
```

```python
In [16]:  list1 = [1, 3, 8, 10]
          list2 = [2, 3, 9]
          merge_sorted_lists(list1, list2)
```

```
Out[16]:  [1, 2, 3, 3, 8, 9, 10]
```

Example: Selection sort

Implement the selection sort which is described as pseudo-code below:

- Start with an unsorted list of elements.
- Find the smallest element in the unsorted portion of the list.
- Swap this smallest element with the first element in the unsorted portion.
- Now, consider the remaining unsorted portion (excluding the element that was just swapped).
- Repeat steps 2 to 4 until the entire list is sorted.
- The result is a sorted list in ascending order.
- The key idea is to repeatedly select the smallest element from the unsorted part of the list and move it to the beginning of the sorted part of the list. This process continues until the entire list is sorted.

Example: Selection sort (cont.)

Example: Selection sort (cont.)

In [17]:
```python
def selectionSort(l):
    for i in range(0, len(l)):
      min = i
      for j in range(i+1, len(l)):
          if(l[j] < l[min]):
              min = j
      tmp = l[i]
      l[i] = l[min]
      l[min] = tmp
    return l

if __name__=="__main__":
    liste = [54,26,93,17,77,31,44,55,20]
    selectionSort(liste)
    print(liste) # [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

[17, 20, 26, 31, 44, 54, 55, 77, 93]

Example: Selection sort (cont.)

```python
def selectionSort(l):
    for i in range(0, len(l)):
      min = i
      for j in range(i+1, len(l)):
          if(l[j] < l[min]):
              min = j
      tmp = l[i]
      l[i] = l[min]
      l[min] = tmp
    return l

if __name__=="__main__":
    liste = [54,26,93,17,77,31,44,55,20]
    selectionSort(liste)
    print(liste) # [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

```
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Complexity is on the order of .

Complexity Calculation

There isn't just one but several methods to calculate the complexity of an algorithm, depending on its properties (and the desired precision of the complexity). Here are the main approaches:

- **Reduction of the code to a known case** and combination of complexities. For example, two loops $(O(\log N))$ result in an overall complexity of $O(n^2 \log(n))$.

- **Reduction to a family of known functions** and calculation of the relative growth rate (limit).

- **Empirical calculation by displaying execution times** as a function of the problem size. It's worth noting that this is independent of the power of the machine.

Data structures

Standard data structures

Included in Python ([documentation](#))

- `int` : Integer, typically 4 bytes in size.
- `long` : Long integer, can be 4 or 8 bytes in size.
- `float` : Real number.
- `str` : String, a sequence of characters (with Unicode conversion).
- `bool` : Boolean, representing True or False.
- `tuple` : Tuple, an ordered collection of elements, e.g., `(1, 2, "ECL", 3.14)`.
- `list` : List, an ordered and mutable collection of elements.
- `set` : Set, an unordered collection of unique elements.
- `dict` : Dictionary, a collection of key-value pairs, e.g., `{'small': 1, 'large': 2}`.

You can check the data type of a variable or object

```python
print(int)
print(type(int))
assert isinstance(3, int)
```

Standard data structures (cont.)

- `range` : A range, representing a sequence of values to generate.
- `complex` : Complex number, e.g., `1j` is one of the square roots of -1.
- `file` : File, for handling file input/output.
- `None` : Represents the absence of a value (equivalent to `void` in some contexts).
- `exception` : Exception, for handling errors and exceptional conditions.
- `function` : Function, a reusable block of code.
- `module` : Module, a file containing Python code and definitions.
- `object` : Object, a generic data type representing any Python object.

Advanced data structures

Not included in Python, often achieved using standard structure and object-oriented programming:

- **Linked Lists**: A data structure where elements are linked together with pointers.

- **Stacks**: A linear data structure that follows the Last-In-First-Out (LIFO) principle.

- **Queues**: A linear data structure that follows the First-In-First-Out (FIFO) principle.

- **Priority Queue**: A data structure that stores elements and returns the one with highest (or lowest) priority.

Advanced data structures (cont.)

- **Heaps**: Tree-based data structure that is often used to implement priority queues.

- **Deques (Double-Ended Queues)**: A linear data structure that allows elements to be added or removed from both ends with constant-time complexity.

- **Trees**: A hierarchical data structure with a root node and child nodes.

- **Graphs**: A non-linear data structure consisting of nodes and edges.

- **Hash Tables (Dictionaries)**: A data structure that allows efficient key-value mapping and retrieval.

Data structures complexity

- **List:** Sequence of elements with dynamic resizing and allow for constant-time access to elements by index.

- **Dictionary:** Allow key-based operations such as insertion, retrieval, and deletion.

- **Set:** Enable et operations like union, intersection, and difference,.

Understanding the complexities of these built-in data structures is essential for selecting the right one for specific programming tasks and optimizing the performance of Python programs.

Dictionnaries

A **dictionary** in Python is an unordered collection of key-value pairs.

- **Keys** in a dictionary must be unique and immutable, meaning you can use strings, numbers, or tuples as keys.

- **Values** can be of any data type.

Dictionaries are useful for a wide range of applications, such as:

- Storing and retrieving configuration settings.
- Counting the frequency of elements in a dataset.
- Representing data in a structured way, such as JSON.

Example: Creating a Dictionary in Python

```python
>>> phonebook = {'bob': 7387, 'alice': 3719, 'jack': 7052}
>>> phonebook['alice']
3719
```

- Implemented as a Python dictionary.
- Raises a `KeyError: 'missing'` exception if accessing an undefined key.
- A good practice is to use `.get("attr", "")` to return a default value if the key doesn't exist.
- We will see that they are widely used for memoization to avoid recomputing certain calculations (e.g., dynamic programming).

Example: Creating a Dictionary in Python

Here's an example of how to create a dictionary in Python:

```python
# Create a dictionary to store information about a person
person = {
    "name": "John Doe",
    "age": 30,
    "city": "New York"
}

# Access values using keys
print("Name:", person["name"])
print("Age:", person["age"])
print("City:", person["city"])
```

In this example, we've created a dictionary named `person` that contains information about an individual. We access the values stored in the dictionary using their respective keys.

Output:

```
Name: John Doe
Age: 30
City: New York
```

.

Question: Count words in a list (using a dictionnary)

Write an algorithm that takes two parameters:

- `stri` : A list of words.
- `n` : An integer.

And returns how many words in the list appear exactly `n` times, and return that count.

Question: Count words in a list (using a dictionnary)

Question: Count words in a list (using a dictionnary)

In [18]:
```python
def countWords(stri, n):

    m = dict()
    for w in stri: # m {'hate': 2, 'love': 4, 'peace': 4}
        m[w] = m.get(w, 0) + 1

    res = 0
    for i in m.values():
        if i == n:
            res += 1

    return res
```

In [19]:
```python
s = [ "hate", "love", "peace", "love",
    "peace", "hate", "love", "peace", "love", "peace" ]

print(countWords(s, 4)) # 2
```

2

Exercice: detect duplicates from a list (using dicts)

Write an algorithm validates the following:

```python
assert duplicatas([1,2]) == False
assert duplicatas([1,2,1]) == True
```

Exercice: detect duplicates from a list (using dicts)

Write an algorithm validates the following:

```python
assert duplicatas([1,2]) == False
assert duplicatas([1,2,1]) == True
```

In [20]:

```python
def duplicatas(L):
    d = {}
    for x in L:
        if x in d:
            return True
        d[x] = True
    return False

assert duplicatas([1,2]) == False
assert duplicatas([1,2,1]) == True
```

Exercice: algorithm optimization (using dicts)

Optimize this algorithm all integers such that $A^2 + B^2 = C^2 + D^2$ with A, B, C, D ranging from 1 to 1000.

```python
n = 1000
for a in range(1, n+1):
    for b in range(1, n+1):
        for c in range(1, n+1):
            for d in range(1, n+1):
                if a**2 + b**2 == c**2 + d**2:
                    print(a, b, c, d)
```

Exercice: algorithm optimization (using dicts) (cont.)

```python
n = 1000
result_map = {}

for c in range(1, n+1):
    for d in range(1, n+1):
        result = c**2 + d**2
        if result in result_map:
            result_map[result].append((c, d))
        else:
            result_map[result] = [(c, d)]

for a in range(1, n+1):
    for b in range(1, n+1):
        result = a**2 + b**2
        if result in result_map:
            matching_pairs = result_map[result]
            for pair in matching_pairs:
                print(a, b, pair)
```

- A first loop uses a dictionary `result_map` to store pairs (c, d) that yield the same result $c^2 + d^2$.
- A second loop iterates through $a^2 + b^2$ values and checks if there are matching pairs in `result_map`.

Sets

A **set** in Python is an unordered collection of unique elements. It is similar to a mathematical set and has the following characteristics:

1. **Uniqueness**: Sets do not allow duplicate elements.

2. **Unordered**: Sets do not have a specific order.

3. **Mutable**: Sets are mutable, which means you can add or remove elements after creating a set.

4. **No Indexing**: Sets do now allow to access elements by their index.

5. **Common Set Operations**: Sets support various set operations such as union, intersection, difference, etc.

Sets (cont.)

```python
# Creating a set
my_set = {1, 2, 3, 4, 5}

# Creating an empty set
empty_set = set()
```

Common set operations include:

- **Adding Elements**: You can add elements to a set using the `add()` method.

- **Removing Elements**: Elements can be removed from a set using the `remove()` or `discard()` method.

- **Set Operations**: You can perform operations like union (`|`), intersection (`&`), difference (`-`), and more between sets.

- **Checking Membership**: You can check if an element is in a set using the `in` operator.

- **Iterating**: You can iterate through the elements of a set using a `for` loop.

Sets are commonly used for tasks where uniqueness and set operations are essential.

Set Operations in Python

Method	Description
`add()`	Adds an element to the set.
`clear()`	Removes all elements from the set.
`copy()`	Returns a copy of the set.
`difference()`	Returns the difference of two sets.
`intersection()`	Returns the intersection of two sets.
`pop()`	Removes and returns a random element from the set.
`union()`	Returns the union of two sets.
`isdisjoint()`	Returns `True` if the sets have no elements in common.
`issubset()`	Returns `True` if the set is a subset of another set.
`issuperset()`	Returns `True` if the set contains another set.

There are many other set operations available in Python, and `frozenset` can be used to create an immutable set.

For more details, refer to the Python documentation.

Exercice: remove duplicatas from a list (using sets)

Write an algorithm validates the following:

```python
assert duplicatas_sets([1,2]) == False
assert duplicatas_sets([1,2,1]) == True
```

Exercice: remove duplicatas from a list (using sets)

Write an algorithm validates the following:

```python
assert duplicatas_sets([1,2]) == False
assert duplicatas_sets([1,2,1]) == True
```

In [21]:
```python
def duplicatas_sets(L):
    s = set()
    for x in L:
        if x in s:
            return True
        s.add(x)
    return False
```

Exercice: remove duplicatas from a list (using sets)

Write an algorithm validates the following:

```python
assert duplicatas_sets([1,2]) == False
assert duplicatas_sets([1,2,1]) == True
```

In [21]:
```python
def duplicatas_sets(L):
    s = set()
    for x in L:
        if x in s:
            return True
        s.add(x)
    return False
```

In [22]:
```python
def duplicatas_sets2(nums):
    return True if len(set(nums)) < len(nums) else False

assert duplicatas_sets2([1,2]) == False
assert duplicatas_sets2([1,2,1]) == True
```

Exercice: find pairs duplicates (using sets)

In a list, return the values that occure exactly 2 times.

Exercice: find pairs duplicates (using sets)

In a list, return the values that occure exactly 2 times.

In [23]:
```python
def find_duplicate_pairs_optimized(lst):
    seen = set()
    duplicate_pairs = []

    for num in lst:
        if num in seen:
            duplicate_pairs.append((num, num))
        seen.add(num)

    return duplicate_pairs
```

In [24]:
```python
input_list = [2, 3, 5, 2, 7, 3, 8, 5]
find_duplicate_pairs_optimized(input_list)
```

Out[24]:   `[(2, 2), (3, 3), (5, 5)]`

Exercice: find words typed with a single row on a keyboard (using sets)

You can determine words that can be typed with a single row of letters on a keyboard using sets in Python.

```
In [25]:  words = ['Velo', 'Ecole', 'Informatique', 'Etroit']
```

```
In [26]:  not {"q"}.difference("qsdfghjklm") # check if a letter is in the set
```

Out[26]: True

Exercice: find words typed with a single row on a keyboard (using sets)

You can determine words that can be typed with a single row of letters on a keyboard using sets in Python.

```
In [25]: words = ['Velo', 'Ecole', 'Informatique', 'Etroit']
```

```
In [26]: not {"q"}.difference("qsdfghjklm") # check if a letter is in the set
```

Out[26]: True

```
In [27]: def check_keyboard(words):
             result = []
             for w in words:
                 ws = set([c.lower() for c in w])
                 if not ws.difference("azertyuiop") \
                     or not ws.difference("qsdfghjklm") \
                     or not ws.difference("wxcvbn"):
                     result.append(w)
             return result
```

```
In [28]: check_keyboard(words) == ['Etroit'] # for a French keyboard
```

Out[28]: True

```
In [29]: check_keyboard(["data"]) == []
```

Out[29]:   True

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

Common Data Structure Operations

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