Introduction

This website contains the resources for BIA101 module - Introduction to Programming

Computational Problems and Algorithms

In the business world, many processes and tasks can be modeled as computational problems that require efficient algorithms to solve.

An **algorithm** is a step-by-step procedure or set of instructions for solving a specific problem or accomplishing a particular task.

Algorithms are crucial in computer programming and play a vital role in optimizing business operations, making data-driven decisions, and improving overall efficiency.

Types of Computational Problems

Computational problems in business settings can be broadly classified into the following categories:

- **Sorting and Searching**: These problems involve organizing and finding specific data items within large datasets. Examples include sorting employee records by salary or searching for a particular customer in a customer database.
- **Optimization Problems**: These problems involve finding the optimal solution among a set of possible solutions, often subject to constraints or limitations. Examples include determining the most cost-effective shipping routes, optimizing investment portfolios, or scheduling resources for maximum productivity.
- **Data Processing and Analysis**: These problems involve manipulating, transforming, and extracting insights from large datasets. Examples include processing financial transactions, analyzing market trends, or performing predictive analytics on customer behavior.
- **Graph and Network Problems**: These problems involve working with interconnected data structures, such as social networks, transportation networks, or supply chain networks. Examples include finding the shortest path between two locations, identifying influential nodes in a social network, or optimizing network flow for efficient distribution.

Algorithms and Their Importance

Algorithms play a crucial role in solving computational problems efficiently and effectively. Well-designed algorithms can:

- **Improve Efficiency**: By optimizing the steps and operations required to solve a problem, algorithms can significantly reduce the time and resources needed to find a solution, leading to improved productivity and cost savings.
- **Enhance Decision-Making**: Algorithms can process vast amounts of data and provide valuable insights, enabling businesses to make informed decisions based on data-driven analysis.
- **Automate Processes**: Many business processes, such as payroll calculations, inventory management, and customer segmentation, can be automated using algorithms, reducing manual effort and minimizing errors.
- **Enable Scalability**: As businesses grow, the amount of data and the complexity of problems increase. Efficient algorithms allow systems to handle larger datasets and more complex problems without sacrificing performance.

Examples of Algorithms:

- **Sorting Algorithms for Employee Records**: To efficiently manage employee information, businesses often need to sort employee records based on various criteria, such as name, salary, or department. Algorithms like Quicksort, Mergesort, or Heapsort can be used to efficiently sort large employee datasets.
- **Shortest Path Algorithms for Logistics**: Logistics companies need to find the most efficient routes for delivering goods to multiple locations. Algorithms like Dijkstra's algorithm or the A* algorithm can be used to determine the shortest or most cost-effective paths between distribution centers and destinations.
- **Recommendation Algorithms for E-commerce**: Online retailers use recommendation algorithms to suggest products to customers based on their purchase history, browsing behavior, and preferences. Collaborative filtering algorithms and content-based filtering algorithms are commonly used for this purpose.
- **Scheduling Algorithms for Resource Allocation**: In industries like manufacturing or project management, efficient resource allocation is crucial. Algorithms like the Hungarian algorithm or job scheduling algorithms can be used to optimally assign tasks to workers or machines, maximizing productivity and minimizing idle time.

- **Clustering Algorithms for Market Segmentation**: Businesses often need to segment their customer base for targeted marketing campaigns or product development. Clustering algorithms like K-means clustering or hierarchical clustering can be used to group customers based on similar characteristics or behaviors.

Basic Computer Programs

Computing has become an integral part of our daily lives, and understanding the basics of computer programming is crucial.

Input Output Flow

At its core, a computer program is a set of instructions that specify how to process some input data to produce an output.

This can be represented as a simple model:

```
Input -> Program -> Output
```

1. **Input**: This is the data or information that the program needs to perform its tasks. In a business context, the input can come from various sources, such as:
   - User input: For example, an employee entering sales figures or customer information into a company's system.
   - External data sources: Such as market data feeds, stock prices, or economic indicators that a financial analysis program might need to process.
2. **Program**: This is the set of instructions, written in a programming language like Python, Java, or C++, that defines how the input data should be processed. The program may perform various operations on the input data, such as:
   - Calculations: A payroll system that computes employees' salaries, taxes, and deductions based on their input data (hours worked, pay rates, etc.).
   - Data manipulation: A marketing automation program that segments customers based on their purchase histories and demographics.
   - Decision-making: A loan approval system that evaluates applicants' credit scores, income, and other factors to determine their eligibility.

3. **Output**: This is the result produced by the program after processing the input data. In a business context, the output can take many forms, including:
    - Reports and dashboards: A business intelligence tool that generates visualizations and reports based on sales data, customer analytics, or financial metrics.
    - Automated communications: A customer service chatbot that responds to customer inquiries based on its programming logic.

Programming Languages

Programming languages are the means by which humans communicate instructions to computers.

They are essentially a set of rules and syntax that allow programmers to write code that can be understood and executed by machines.

There are numerous programming languages available, each with its own strengths and weaknesses, designed for specific purposes or domains.

Some of the most popular programming languages for beginners include:

- Python: Known for its simplicity, readability, and versatility, Python is widely used in various domains, including web development, data analysis, machine learning, and scripting.
- Java: A robust and object-oriented language, Java is particularly useful for building desktop applications, mobile apps, and enterprise-level software.
- C++: A powerful and efficient language, C++ is widely used in systems programming, game development, and performance-critical applications.
- JavaScript: While initially designed for web browsers, JavaScript has evolved into a versatile language used for both front-end and back-end web development, as well as server-side scripting.

git version control

What is version control?

Version control is a system that records changes to files or sets of files over time.

It allows you to track modifications, revert to previous versions, and collaborate with others on the same codebase.

Version control systems are essential in software development, enabling multiple developers to work on the same project simultaneously while maintaining a organized and efficient workflow.

Benefits of Version Control:

- Track Changes: Version control systems keep a detailed log of every modification made to the code, including who made the change, when it was made, and why it was made. This makes it easy to track down and fix bugs, or revert to a previous working version if necessary.
- Collaboration: Multiple developers can work on the same codebase concurrently without overwriting each other's changes. Version control systems provide mechanisms for merging changes from different contributors into a unified codebase.
- Backup and Recovery: Version control acts as a backup system, allowing you to recover lost or damaged files by reverting to a previous version.
- Branching and Merging: Version control systems enable developers to create separate branches for new features or experiments, and then merge those changes back into the main codebase once they are stable and tested.
- Release Management: With version control, you can tag specific versions of your codebase, making it easier to manage and deploy releases.

git

Git is a distributed version control system that is widely used in software development.

It was created by Linus Torvalds in 2005 to manage the development of the Linux kernel, and has since become the de facto standard for version control in the industry.

Key Git Concepts:

- **Repository**: A repository (or "repo") is a collection of files and directories that Git tracks. It contains the complete history of changes made to the project over time.

- **Commit**: A commit is a snapshot of the repository at a particular point in time. Each commit records changes made to the files, along with a commit message describing the changes.
- **Branch**: A branch is an independent line of development within a repository. Branches allow you to work on new features or experiments without affecting the main codebase (often called the "main" or "master" branch).
- **Merge**: Merging is the process of combining changes from one branch into another. When you've completed work on a branch, you can merge it into the main branch to incorporate your changes.
- **Pull** Request: In collaborative environments, a pull request is a way to propose changes from your branch to be merged into the main codebase. Other team members can review the changes before they are merged.

Basic git workflow

Here's a typical workflow when working with Git:

- **Clone**: Create a local copy of the repository on your machine using `git clone <repo_url>`.
- **Branch**: Create a new branch for your feature or bug fix using `git checkout -b <branch_name>`
- **Edit**: Make changes to the codebase and save the files.
- **Stage**: Stage the modified files for committing using `git add <file_names>`
- **Commit**: Record the changes with a commit message using `git commit -m "Descriptive commit message"`
- **Push**: Upload your commits to the remote repository using `git push origin <branch_name>`
- **Pull Request (optional)**: If you're working in a collaborative environment, create a pull request to propose your changes for review and merging.
- **Merge**: Once your changes are approved, merge your branch into the main branch using `git merge <branch_name>`
- **Pull**: Periodically update your local repository with the latest changes from the remote repository using `git pull`

In BBI Class environment:

Usually in your classes you will be running these commands only:

- `git add .` - to stage all changes
- `git commit -m "Your commit message here"` - to commit changes
- `git push` - to push changes to the remote repository

Language Compilation Process

Computer programs are written in high-level programming languages (e.g., Python, Java, C++), which are human-readable but not directly executable by computers.

To run these programs, they need to be translated into a form that the computer's processor can understand, typically machine code or bytecode.

This translation process is known as compilation or interpretation, depending on the approach used.

Compilation

Compilation is the process of translating a high-level programming language into machine code, which is a low-level language that the computer's processor can directly execute.

Compiled programs typically run faster than interpreted programs because the translation to machine code is done ahead of time, and the resulting executable can run directly on the processor without the need for further translation at runtime.

Interpretation

Interpretation is the process of executing a high-level programming language directly, without first translating it into machine code.

An interpreter reads the source code, analyzes it, and immediately executes the corresponding instructions.

Interpreted programs are generally easier to develop and debug because the source code can be executed directly without the need for a separate compilation step.

However, interpreted programs typically run slower than compiled programs because the translation and execution happen simultaneously at runtime.

Compilation vs. Interpretation

S.NO.	Compiled Language	Interpreted Language
1	Compiled language follows at least two levels to get from source code to execution.	Interpreted language follows one step to get from source code to execution.
2	A compiled language is converted into machine code so that the processor can execute it.	An interpreted language is a language in which the implementations execute instructions directly without earlier compiling a program into machine language.
4	The compiled programs run faster than interpreted programs.	The interpreted programs run slower than the compiled program.
5	In a compiled language, the code can be executed by the CPU.	In Interpreted languages, the program cannot be compiled, it is interpreted.
6	This language delivers better performance.	This language delivers slower performance.

Compiled languages:

- C
- C++
- Rust
- Go

Interpreted languages:

- Python
- Ruby
- JavaScript
- PHP

Binary Representation

Computers, at their core, are built using electronic circuits.
They operate based on the presence or absence of electric signals (on and off), which are represented as 1s and 0s, respectively.

- These 1s and 0s are the fundamental units of information in the **binary number system**, which is the language that computers understand and process.

On the other hand, humans have traditionally used the **decimal number system (base 10)** for counting and arithmetic operations.

The decimal system is more intuitive for us because we have ten fingers, making it easier to conceptualize and work with numbers that range from 0 to 9.

Knowing how to convert between decimal and binary numbers is essential for solving certain type of computational problems.

Example

Suppose we want to represent the number "10" in both binary and decimal systems.

In the decimal system, we write it as "10," which is a single digit representing ten units.

However, in the binary system, "10" is represented as "1010," where each digit (0 or 1) is called a bit (binary digit).

Converting Decimal to Binary

Converting Binary to Decimal

Introductory Python

Before diving into writing actual code, it's essential to understand some fundamental programming concepts that are common across most programming languages:

- **Variables**: Variables are used to store data values, such as numbers, strings, or more complex data types, for later use or manipulation.

- **Data Types**: Programming languages support various data types, including integers, floating-point numbers, characters, strings, booleans, and more complex structures like arrays and objects.
- **Control Structures**: These structures allow programs to make decisions and execute different blocks of code based on certain conditions. Examples include if-else statements, loops (for, while), and switch statements.
- **Functions**: Functions are reusable blocks of code that perform specific tasks. They can accept input parameters and return output values, promoting code modularity and reusability.
- **Object-Oriented Programming (OOP)**: Many modern programming languages support object-oriented programming, which revolves around the concept of objects containing data (properties) and code (methods). OOP promotes code organization, reusability, and maintainability.

Namespace

A namespace is a mapping of names (identifiers) to objects (variables, functions, classes, etc.) in Python.

It helps to prevent naming conflicts by allowing the same name to be used for different objects in different namespaces.

For example you cannot use `for` as a variable name because it is a reserved keyword in Python that is used to define a loop.
Similarly, you cannot use `def` as a variable name because it is a reserved keyword used to define a function.

```
for = "Hello World" # SyntaxError: can't assign to keyword
print(for) # cannot execute
def = 0 # error as def word is reserved
```

Identifiers

An identifier is a name given to a variable, function, class, or any other object in Python. It is the rules in how you can name your variables, functions, classes, etc.

Identifiers must follow certain rules:

- They must start with a letter (a-z or A-Z) or an underscore (_).
- After the first character, they can contain letters, digits (0-9), and underscores.
- Identifiers are case-sensitive (e.g., myVar and myvar are different identifiers).
- Keywords (reserved words in Python) cannot be used as identifiers.

Variables

A variable is a named storage location in memory that holds a value. In Python, variables are created when you assign a value to them, and they can store different types of data, such as numbers, strings, lists, or objects. Variables are dynamically typed, meaning their type is determined by the value they hold.

```
x = 5       # Integer variable
name = "Alice"  # String variable
my_list = [1, 2, 3]  # List variable
```

Arithmetic Operators

Python provides several arithmetic operators for performing mathematical operations on numerical values.

Here are some commonly used arithmetic operators:

- `+` (Addition): Adds two values together.
- `-` (Subtraction): Subtracts one value from another.
- `*` (Multiplication): Multiplies two values.
- `/` (Division): Divides one value by another and returns a floating-point result.
- `//` (Floor Division): Divides one value by another and returns the integer part of the result (rounded down).
- `%` (Modulus): Returns the remainder when one value is divided by another.
- `**` (Exponentiation): Raises one value to the power of another.

```
# Example for arithmetic operators
a = 10
b = 3

result = a + b  # Addition: result = 13
result = a - b  # Subtraction: result = 7
result = a * b  # Multiplication: result = 30
result = a / b  # Division: result = 3.3333333333333335
result = a // b  # Floor Division: result = 3
result = a % b  # Modulus: result = 1
result = a ** b  # Exponentiation: result = 1000
```

These terms are fundamental concepts in Python programming and are essential for understanding how variables, identifiers, and arithmetic operations work in the language.

Logical & Conditional Expression

Conditional Expression

Conditional expressions in Python are used to execute different blocks of code based on certain conditions.

They are formed using conditional statements like if, elif (else if), and else.

`if` statement

The if statement executes a block of code if a specified condition is `True`

```
x = 10

if x > 5:
    print("x is greater than 5")  # This line will be executed
```

`if-else` statement

The if-else statement executes one block of code if a specified condition is `True`, and another block if the condition is `False`

```
x = 3

if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")  # This line will be executed
```

`if-elif-else` statement

The if-elif-else statement allows you to check multiple conditions and execute different blocks of code **based on the first condition that evaluates to True.**

```
x = 10
```

```
if x < 5:
    print("x is less than 5")
elif x < 10:
    print("x is less than 10")
else:
    print("x is greater than or equal to 10")  # This line will be executed
```

Nesting

Conditional expressions can be nested and combined with logical expressions to create more complex decision-making structures in your Python code.

```
age = 25
is_student = True

if age < 18:
    print("You are a minor")
elif age >= 18 and is_student:
    print("You are a student")
else:
    print("You are an adult")  # This line will be executed
```

Logical Operators

Logical expressions in Python are used to combine or negate conditions.

They are formed using logical operators and operands (which can be variables, values, or other expressions).

The result of a logical expression is either True or False.

Python provides the following logical operators:

`and`: Returns True if **both** operands are True, otherwise False. `or`: Returns True if **at least one** of the operands is True, otherwise False. `not`: Returns the **opposite boolean value** of the operand (True if the operand is False, and False if the operand is True).

Example:

```
x = 5
y = 10

# and operator
result = x < 10 and y > 5  # True
```

```
# or operator
result = x > 10 or y > 5  # True

# not operator
result = not (x == 5)  # False
```

More Examples

```
# ================
# Example 1
x = 5
y = 10

if x > 5 and y < 15:
    print("Both conditions are True")  # This line will be executed
else:
    print("At least one condition is False")

# ================
# Example 2 - Nested if-else
age = 25
is_student = True

if age < 18:
    print("You are a minor")
elif age >= 18 and is_student:
    print("You are a student")
else:
    print("You are an adult")  # This line will be executed

# ================
# Example 3 - If inside a if block
x = 5
y = 10

if x > 5:
    if y > 5:
        print("Both x and y are greater than 5")  # This line will be executed
    else:
        print("x is greater than 5, but y is not")
else:
    print("x is not greater than 5")

# ================
# More if inside if example
if x > 5:
    if y < 5:
        print("Both x and y are greater than 5")
    else:
        print("x is greater than 5, but y is not") # This line will be executed
else:
    print("x is not greater than 5")

# ================
# Example 4 - Using not operator
x = 5
y = 10
if not x == a:
    print("x is not equal to a")  # This line will be executed
else:
    print("x is equal to a")

# =================
# Example 5 - Using or operator
if x > 5 or y > 5:
    print("At least one of the conditions is True")  # This line will be executed
else:
    print("Both conditions are False")

# =================
# Example 6 - Using not operator
if not (x == 5 and y == 10):
    print("x is not equal to 5 and y is not equal to 10")  # This line will be executed
else:
    print("x is equal to 5 and y is equal to 10")

# =================
# Example 7 - Triple nested if statement
x = 5
y = 10

if x > 5:
    if y > 5:
        if x + y > 15:
            print("x, y, and their sum are greater than 5")
        else:
            print("x, y are greater than 5, but their sum is not greater than 15") # This line will be executed
    else:
        print("x is greater than 5, but y is not")
else:
    print("x is not greater than 5")
```

Literals & Abstract Data Types

Literals are the raw data values that are directly assigned to variables or constants in Python.

They are the building blocks of data manipulation in any programming language.

Python supports various types of literals:

Numeric Literals:

- **Integer Literals**: Whole numbers (positive or negative), without any fractional part. Examples: 42, -7, 0
- **Floating-Point Literals**: Numbers with a fractional part, represented using a decimal point. Examples: 3.14, -0.001, 6.625
- **Complex Literals**: Numbers with a real and imaginary part, represented using j or J as the imaginary unit. Examples: 3+4j, -5.6J

```
# Integer Literals
x = 42
y = -7
z = 0

# Floating-Point Literals
a = 3.14
b = -0.001
c = 6.625

# Complex Literals
d = 3 + 4j
e = -5.6J
```

String Literals

A string literal is a sequence of characters enclosed in single quotes ('), double quotes ("), or triple quotes (''' or """). Examples: 'Hello', "World", '''Multi-line string'''

```
# Single-quoted string
message = 'Hello, World!'

# Double-quoted string
name = "Alice"

# Triple-quoted string (multi-line)
poem = '''Roses are red,
Violets are blue,
Sugar is sweet,
And so are you.'''
```

Boolean Literals

Boolean literals represent the truth values `True` and `False`. They are used in logical operations and conditional statements.

```
# Boolean Literals
is_student = True
is_raining = False
```

Special Literals

None: Represents a null value or the absence of a value.

```
# None Literal
result = None
```

Abstract Data Types

Python provides several built-in data types that are used to store collections of values.

These are known as Abstract Data Types (ADTs) and include the following:

List A list is an ordered collection of items. Lists are mutable (changeable) and can contain elements of different data types.

Lists are defined using square brackets `[]`.

```
fruits = ['apple', 'banana', 'cherry']
numbers = [1, 2, 3, 4, 5]
mixed = [1, 'hello', 3.14, True]
```

Tuple

A tuple is an ordered, immutable (unchangeable) collection of items.

Tuples are defined using parentheses `()`.

```
point = (3, 4)
person = ('John', 30, 'Engineer')
coordinates = (10, 20, 30)
```

Set

A set is an unordered collection of unique elements.

Sets are defined using curly braces `{}`.

```
fruits = {'apple', 'banana', 'cherry'}
numbers = {1, 2, 3, 4, 5}
unique_chars = {'a', 'b', 'c', 'd'}
```

Dictionary

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

Dictionaries are defined using curly braces `{}`, with keys and values separated by colons.

```
person = {'name': 'John', 'age': 30, 'occupation': 'Engineer'}
person1 = {'name': 'Alice', 'age': 25, 'city': 'New York'}
product = {'id': 101, 'name': 'Laptop', 'price': 999.99}
student = {'name': 'Bob', 'grades': [85, 92, 78]}
```

Lists/Array Methods

There are various methods to change/edit values of a list as shown in the code snippets below

NOTE: The properties of Stacks and Queues can be achieved using list methods like `.pop()` `.remove()` `.insert()` etc.

```
# ==================
# Getting the length of the list
numbers = [10, 20, 30, 40]
# Note: The length is not 0 based (counting starts from 1 unlike indexing)
length = len(numbers)
print(length) # 4

# ==================
# Accessing Elements
# The index starts from 0 and not 1
numbers = [10, 20, 30, 40]
second_number = numbers[1]
print(second_number)  # Output: 20

fruits = ["apple", "banana", "cherry"]
last_fruit = fruits[2]  # Accessing the first element
print(last_fruit)  # Output: cherry

# Negative Indexing: Start from the last:
fruits = ["apple", "banana", "cherry", "orange"]
last_fruit = fruits[-1]  # Accessing the last element
print(last_fruit)  # Output: orange

# Accessing Slices: Same concept as String Slices
numbers = [1, 2, 3, 4, 5]
sublist = numbers[1:4]  # Accessing a sublist from index 1 to 4 (exclusive)
print(sublist)  # Output: [2, 3, 4]

# ==================
# Removing Elements
numbers = [10, 20, 30, 40]
del numbers[1]
```

```
print(numbers)  # Output: [10, 30, 40]

# Remove elements by value:
fruits = ["apple", "banana", "cherry", "banana"]
fruits.remove("banana")  # Removing the first occurrence of "banana"
print(fruits)  # Output: ["apple", "cherry", "banana"]

# ==================
# Modifying Elements
numbers = [10, 20, 30, 40]
numbers[2] = 55
print(numbers)  # Output: [10, 20, 55, 40]

# ==================
# Appending Elements
numbers = [10, 20, 30]
numbers.append(80)
print(numbers)  # Output: [10, 20, 30, 80]

fruits = ["apple", "banana"]
fruits.append("cherry")  # Adding an element to the end
print(fruits)  # Output: ["apple", "banana", "cherry"]

# Insert an element at a specific index:
numbers = [1, 3, 5]
numbers.insert(1, 2)  # Inserting 2 at index 1
print(numbers)  # Output: [1, 2, 3, 5]

# ==================
# Extending Elements
numbers = [1, 2, 3]
more_numbers = [4, 5, 6]
numbers.extend(more_numbers)
print(numbers)  # Output: [1, 2, 3, 4, 5, 6]

# using `in` operator and `not in` operator to check for elements in the array
# The concept here is as same as the operator used in strings
colors = ["red", "green", "blue"]
is_present = "green" in colors
print(is_present)  # Output: True

# =============
```

String Manipulation

Strings are sequences of character data. The string type in Python is called `str`

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
s = "I am a string"
t = 'I am a string too"

type(s) # <class str>
type(t) # <class str>
```

String Operators

You can perform concatenation and multiplication operations on string using the symbols `+` `*`:

```
# ================
# plus operator: String Concatenation
first_name = "Alice"
last_name = "Smith"
full_name = first_name + " " + last_name
print(full_name) # Alice Smith

s = "foo"
t = "bar"
u = "baz"

s + t # foobar
s + t + u # foobarbaz

s + u # foobaz

# ================
# Multiply Operator
s * 2 # foofoo
t * 3 # barbarbar
w = "hi."
w * 4 # hi.hi.hi.hi
```

The operator `in` and `not in` can be used to check if a string exist in another string

```
# ================
s = "foo"
t = "bar"
u = "baz"

r = 'o' in s # True
m = 'o' in t # False
r = 'o' not in s # False
r = 'z' not in 'abc' # True
```

String Manipulation

String methods such as `str.lower()` `str.upper()` can be used to change the string's case and slicing using `[]` can be used to extract particular string.

```
# =================
# String Methods
# .upper()
message = "Hello, world!"
uppercase_message = message.upper()
print(uppercase_message) # HELLO, WORLD!

# .lower()
message = "Hello, world!"
lower_case = message.upper()
print(lower_case) # hello, world!

# .capitalize()
name = "emma"
capitalized_name = name.capitalize()
print(capitalized_name) # Emma
```

For a full list of available string method refer to this article

String Slicing

Just like array indexing, you can access elements and slice elements from a string.

```
# ==================
# Basic Slicing
text = "Python Programming"
# Extracts characters from index 2 (inclusive) to index 7 (exclusive).
sliced_text = text[2:8]
print(sliced_text)  # Output: thon Pro

# ==================
# Slicing from beginning
text = "Hello World"
sliced_text = text[:5]
print(sliced_text)  # Output: Hello

# ==================
# Slicing from end:
text = "Welcome Back"
sliced_text = text[-4:]
print(sliced_text)  # Output: Back

# ==================
# Reversing a string:
text = "Never say never"
sliced_text = text[::-1]
print(sliced_text)  # Output: rven ays revaN
```

Loops

In computing, there will be tasks/operations where you will have to perform the same tasks again and again.

So how does a computer perform a task repeatedly without manually repeating the same code over and over again.

Loops, as the name suggests, allowed the Python to repeat a set of instructions multiple times, saving it from the tedious task of writing the same code again and again.

`for` loop

The `for` loop is used to iterate over a sequence (list, tuple, string) or other iterable objects.

In other words, it is used to iterate over a sequence of items, such as a list, a string, or a range of numbers.

```
# Iterating over a list
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    # the string syntax is used to print variables inside a string --> Called a f-string
    print(f"I love eating {fruit}s!")

# Output:
# I love eating apples!
# I love eating bananas!
# I love eating cherries!
```

In this example, the for loop iterates over each item in the fruits list, and for each iteration, the loop body (the indented code block) is executed with the current item bound to the variable fruit.

`while` loop

This loop is perfect for situations where the Python needed to repeat a set of instructions **until a certain condition is met**

```
# Counting down from 5
count = 5
while count > 0:
    print(f"Counting down... {count}")
    count -= 1
print("Liftoff!")

# Output:
# Counting down... 5
# Counting down... 4
# Counting down... 3
# Counting down... 2
# Counting down... 1
# Liftoff!
```

In this example, the while loop continues to execute the loop body as long as the condition count > 0 is true.

Inside the loop, the value of count is printed, and then it is decremented by 1. Once count becomes 0, the loop condition is no longer true, and the loop terminates, printing "Liftoff!" after the loop.

Nested Loops

Loops can be nested inside other loops. This means that you can put one loop inside the body of another loop.

```
# Printing a multiplication table
for i in range(1, 11): # i goes from 1 to 10
    for j in range(1, 11): # j goes from 1 to 10
        print(f"{i} x {j} = {i * j}")
    print("--------------------")

# Output:
# 1 x 1 = 1
# 1 x 2 = 2
# ...
# 1 x 10 = 10
# --------------------
# 2 x 1 = 2
# 2 x 2 = 4
# ...
# 2 x 10 = 20
# --------------------
# ...
```

Loop Control Statements

Loop control statements change the execution from its normal sequence --> `break` and `continue`.

In other words, they allow you to control how a loop executes a code.

`break` statement

The `break` statement is used to **exit** the loop prematurely, regardless of the loop condition.

```
# Breaking out of a loop
for i in range(1, 11):
    if i == 5:
        break
    print(i)

# Output:
# 1
# 2
# 3
# 4
```

In the above example:

Loop breaks and it does not print 5 --> The loop stops when i == 5 and exits out of the loop

- The same applies for while loops too

`continue` statement

The `continue` statement is used to skip the rest of the code inside a loop for the current iteration only.

In other words, it **skips** the rest of the code inside the loop for the current iteration and continues with the next iteration.

```
# Skipping an iteration
for i in range(1, 9):
    if i == 5:
        continue # Skip the rest of the code inside the loop if i is 5
    print(i)

# Output:
# 1
# 2
# 3
# 4
# 6
# 7
# 8
```

In the above example:

The value of `i` becomes 5, the `continue` statement is executed, and the rest of the code inside the loop is skipped.

The loop then continues with the next iteration.

- The same applies for while loops too

`for` loop examples

```
# ==================
# Iterating over a list
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
# Output:
# apple
# banana
# cherry

# ==================
# Iterating over a string
greeting = "Hello, World!"
for char in greeting:
    print(char)
# Output:
# H
# e
# l
# l
# o
# ,
#
# W
# o
# r
# l
# d
# !

# ==================
# Iterating over a range
for i in range(5): # i goes from 0 to 4
    print(i)
# Output:
# 0
```

```python
# 1
# 2
# 3
# 4

# ==================
# Iterating with index
colors = ['red', 'green', 'blue']
for i, color in enumerate(colors): # i is the index, color is the value --> This is how enumerate() works: It returns the index and the value
    print(f"{i}: {color}")
# Output:
# 0: red
# 1: green
# 2: blue

# ==================
# Iterating over a list with conditional
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for num in numbers:
    if num % 2 == 0: # Research the % operator
        print(num)
# Output:
# 2
# 4
# 6
# 8
# 10

# ==================
# Iterating over multiple sequences
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
for name, age in zip(names, ages): # zip() function is used to combine two or more sequences
    print(f"{name} is {age} years old.")
# Output:
# Alice is 25 years old.
# Bob is 30 years old.
# Charlie is 35 years old.

# ==================
# Nested for loops
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] # Array inside an array: Imagine it as a 3x3 matrix
# matrix[0] = [1, 2, 3]
# matrix[1] = [4, 5, 6]
# matrix[0][0] = 1
# matrix[0][1] = 2
# matrix[1][2] = 6
for row in matrix: # Loop through each row
    for element in row: # Loop through each element in the row
        print(element + " ") # Print the element and a space
    print()
# Output:
# 1 2 3
# 4 5 6
# 7 8 9
# =================
# For loop with else --> Special for loop
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
else:
    print("Loop completed")
# Output:
# 1
# 2
# 3
# 4
# 5
# Loop completed
```

while loop examples

```
# =================
# Basic while loop
count = 0
while count < 5:
    print(count)
    count += 1 # Increment the count by 1
    # if count == 5, the loop will stop

    # NOTE:  if count variable is not incremented, the loop will run forever
    # This is called an infinite loop: because count will always be 0 and count < 5 will always be true

# Output:
# 0
# 1
# 2
# 3
# 4

# =================
# Infinite loop (until break)
count = 0
while True: # Intentionally creating an infinite loop
    print(count)
    count += 1
    if count == 5: # Break out of the loop when count is 5:
        # This is the only way to exit the loop
        break
# Output:
# 0
# 1
# 2
# 3
# 4

# =================
# While loop with else: also a special while loop
```

```
num = 0
while num < 5:
    print(num)
    num += 1
else:
    print("Loop completed")
# Output:
# 0
# 1
# 2
# 3
# 4
# Loop completed

# ==================
# While loop with conditional
num = 10
while num > 0:
    if num % 2 == 0:
        print(num)
    num -= 1
# Output:
# 10
# 8
# 6
# 4
# 2

# ==================
# While loop with user input
user_input = ""
while user_input.lower() != "quit":
    # Keep on asking user for input until the user types 'quit'
    user_input = input("Enter 'quit' to exit: ")
print("Loop exited")
# Output:
# Enter 'quit' to exit: hello
# Enter 'quit' to exit: world
# Enter 'quit' to exit: quit
# Loop exited

# ==================
# Nested while loops
i = 1
while i <= 3:
    j = 1
    while j <= 3:
        print(f"({i}, {j})")
        j += 1
    i += 1
# Output:
# (1, 1)
# (1, 2)
# (1, 3)
# (2, 1)
# (2, 2)
# (2, 3)
# (3, 1)
# (3, 2)
# (3, 3)
```

Functions

In Python, a function is a block of reusable code that performs a specific task.

Functions are designed to take input(s) in the form of arguments, perform some operations or calculations on those inputs, and optionally return a result.

Functions are essential for modular programming, code reusability, and organizing code into logical units.

The objective behind function is that you define a function once and then you can call it multiple times without having to rewrite the code each time.

Python Code Examples of Functions:

```
# ========================================
# Defining a function
def greet():
    print("Hello!")

# If you do not write the code below, the function will not be executed (it is not called)
greet()  # Output: Hello!

# ========================================
# Function with arguments: You want to pass data while calling to the function
def greet(name): # The variable `name`'s value will be `Alice` when the function is called
    print(f"Hello, {name}!")

greet("Alice")  # Output: Hello, Alice!

# ========================================
# Function with return value
def add(a, b):
    sum = a + b
    return sum # the vale of `sum` is returned to whereever the function was called from

# The function returns a value which is captured/stored in an another variable
result = add(3, 5)
print(result)  # Output: 8

# ========================================
# Function with default arguments
# If you do not pass any value to the function, the default value `Human` will be used
# If you pass a value to the function, that value will be used
```

```
def greet_person(name="Human"):
    print(f"Hello, {name}!")

greet_person()  # Output: Hello, Human!
greet_person("Alice")  # Output: Hello, Alice!

# ========================================
# Function with keyword arguments
def greet_person(name="Human", age=0):
    print(f"Hello, {name}! You are {age} years old.")

greet_person()  # Output: Hello, Human! You are 0 years old.
greet_person(age=25, name="Alice")  # Output: Hello, Alice! You are 25 years old.
# In the code above with keyword arguments, the order of the arguments does not matter

# ========================================
# Function with positional arguments
def greet_person(name, age):
    print(f"Hello, {name}! You are {age} years old.")

# The order matters here
greet_person("Alice", 25)  # Output: Hello, Alice! You are 25 years old.

# ========================================
# Function with nested functions
# Python allows you to define functions inside other functions
def outer_function():
    def inner_function():
        print("Inside inner function")
    print("Inside outer function")
    inner_function()

outer_function()
# Output:
# Inside outer function
# Inside inner function
```

Function Call Stack

In the previous section, we learned about functions and how they work.

In this section, we will learn about the function call stack: what happens when multiple functions are called, and how the computer keeps track of them.

A function call stack is created when one function is called inside another function.

The function call stack behaves like a stack data structure.

Simple Case Example:

```python
def greet():
    print("Hello, World!")

def main():
    greet()
    print('This is the main function')

main()

# Output:
# Hello, World!
# This is the main function
```

In the above code, the `main()` function calls the `greet()` function.

When you run this code, the following happens:

- The interpreter starts executing the `main()` function.
- `main()` calls the `greet()` function.
  NOTE: The `greet()` function need to finish executing before the `main()` function can continue executing: Cannot print ("This is the main function") until `greet()` is done executing.
- A new stack frame for `greet()` is created and pushed onto the call stack.
- `greet()` executes and prints "Hello, World!".
- The stack frame for `greet()` is popped off the call stack, and execution returns to `main()`.
- `main()` completes execution, and its stack frame is popped off the call stack.

Another Example:

```python
def greet(name):
    print(f"Hello, {name}!")
    print('greet done')

def welcome(age, occupation):
    greet(f"{occupation} of age {age}")
    print('welcome done')

def main():
    welcome(30, "Developer")
    print('main done')

main()

# Output:
# Hello, Developer of age 30!
# greet done
# welcome done
# main done
# No call stack: Stops Executing
```

When you run this code, the following happens:\

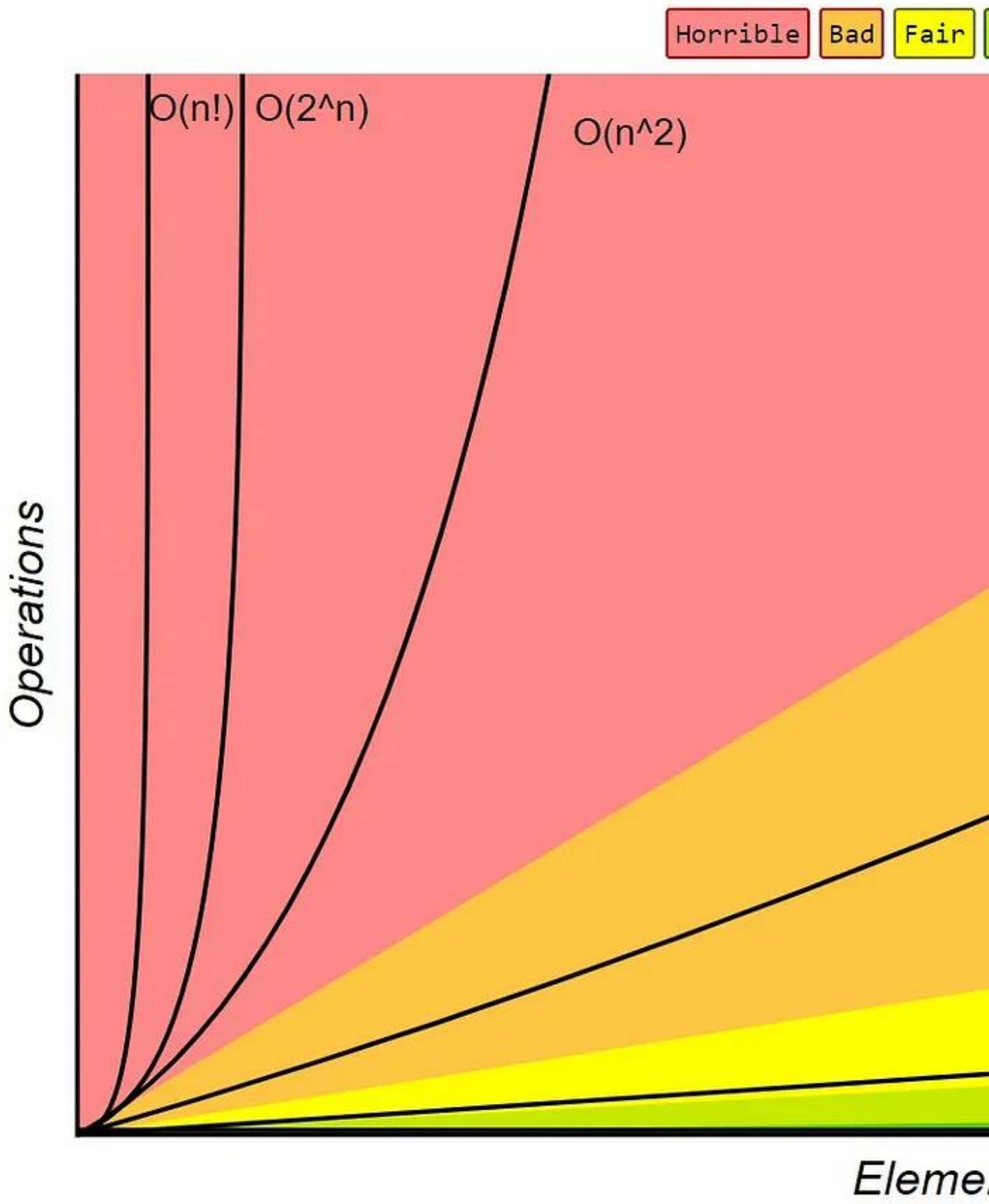
- The interpreter starts executing the `main()` function.
- `main()` calls the `welcome()` function with arguments 30 and "Developer".
- A new stack frame for `welcome()` is created and pushed onto the call stack.
- Inside `welcome()`, the `greet()` function is called with the argument "Developer of age 30".
- A new stack frame for `greet()` is created and pushed onto the call stack.
- `greet()`executes and prints "Hello, Developer of age 30!".
- The stack frame for `greet()` is popped off the call stack, and execution returns to `welcome()`.
- `welcome()` completes execution, and its stack frame is popped off the call stack.
- `main()` completes execution, and its stack frame is popped off the call stack.

Space & Time Complexity; Asymptotic Notation

Space and time complexity are concepts used to measure the efficiency of an algorithm or a program.

They help us understand how an algorithm or program behaves **as the input size grows**, allowing us to make informed decisions about which approach is better suited for a particular problem.

Big-O Compl



Time Complexity

Time complexity refers to the amount of time an algorithm or program takes to complete its execution relative to the input size.

It is typically expressed using **Big O notation**, which describes the worst-case scenario or the upper bound of the algorithm's running time.

Big O notation is a mathematical way of expressing the growth rate of an algorithm's running time relative to the input size.

- In other words, it describes how long the algorithm will take to run as the input size grows (as the input size tends to infinity).

The following table shows the most common time complexities in order of increasing efficiency:

Notation	Name	Description
O(1)	Constant	The algorithm's running time is constant and does not depend on the input size - The time required to run an algorithm is same no matter the input (be it input size of 10 or 1 million)
O(log n)	Logarithmic	The algorithm's running time grows logarithmically with the input size.
O(n)	Linear	The algorithm's running time grows linearly with the input size.
O(n log n)	Linearithmic	The algorithm's running time grows in proportion to n log n, where n is the input size.
O(n^2)	Quadratic	The algorithm's running time grows quadratically with the input size.
O(2^n)	Exponential	The algorithm's running time doubles with each addition to the input size.
O(n!)	Factorial	The algorithm's running time grows factorially with the input size.

Space Complexity

Space complexity refers to the amount of memory an algorithm or program uses relative to the input size.

- In other words, it describes how much space/memory of a computer will be used by the algorithm as the input size grows.

Similar to time complexity, space complexity is also expressed using Big O notation, which describes the worst-case scenario or the upper bound of the algorithm's memory usage.

The following table shows the most common space complexities in order of increasing efficiency:

Notation	Name	Description
O(1)	Constant	The algorithm uses a constant amount of memory regardless of the input size - The space required to run an algorithm is same no matter the input (be it input size of 10 or 1 million).
O(log n)	Logarithmic	The algorithm's memory usage grows logarithmically with the input size.
O(n)	Linear	The algorithm's memory usage grows linearly with the input size.
O(n log n)	Linearithmic	The algorithm's memory usage grows in proportion to n log n, where n is the input size.
O(n^2)	Quadratic	The algorithm's memory usage grows quadratically with the input size.
O(2^n)	Exponential	The algorithm's memory usage doubles with each addition to the input size.
O(n!)	Factorial	The algorithm's memory usage grows factorially with the input size.

Code Examples

Let's look at some code examples to understand how time and space complexity are calculated.

Space-Time Example

Consider the following Python function that calculates the sum of the first `n` numbers:

```
def sum_of_n_numbers(n):
    total = 0
    for i in range(1, n+1):
        total += i
    return total
```

The time complexity of this function is `O(n)` because the loop runs `n` times, where `n` is the input size.

- The loop runs `n` times, so the time complexity is `O(n)`.
- The space complexity of this function is `O(1)` because it uses a constant amount of memory regardless of the input size - It uses the same amount of variable.
- **NOTE**: If you were using an array and adding elements to it, the space complexity would be `O(n)` because the array would grow in proportion to the input size.

More Examples:

```
# ==============================
# Time complexity: O(n): Linear
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

# ==============================
# Time complexity: O(1): Constant
# Space complexity: O(1): Constant
def access_element(arr, index):
    if index >= len(arr):
        return "Index out of range"
    return arr[index]

# ==============================
# Time complexity: O(n^2): Quadratic
# Space complexity: O(1): Constant
def nested_loops(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            print(arr[i], arr[j])

# ==============================
# Time complexity: O(log n): Logarithmic
# Space complexity: O(1): Constant
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
```

```
        low = mid + 1
    else:
        high = mid - 1
return -1

# ==============================
# Time complexity: O(n log n): Linearithmic
# Space complexity: O(n): Linear
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)
# In the above example the merge function is not shown, but it is a helper function that merges two sorted arrays.
# The space complexity of merge sort is O(n) because it uses additional memory (new `left` and `right` arrays whose values gets added ) to store the sorted subarrays.
```

Exercises

```
# ==============================
def sum_of_squares(n):
    total = 0
    for i in range(1, n+1):
        total += i**2
    return total

# Time: O(n)
# Space: O(1)

# ==============================
input = [1,2,3]
for i in range(1, len(input)):
    print(i)

# Time: O(n)
# Space: O(1)

# ==============================
arr = []
input = [1,2,3]
for i in range(1, len(input)):
    arr.append(i)
    print(i)

# Time: O(n)
# Space: O(n)

# ==============================
input = [1,2,3]
for i in range(1, len(input)):
    break
    print(i)

# Time: O(1)
# Space: O(1)

# ==============================
input = [1,2,3]
for i in range (1, len(input)):
    for j in range(1, 10):
        print(j)

# Time: O(n^2)
# Space: O(1)

# ==============================
input = [1,2,3]
a = []
for i in range (1, len(input)):
    for j in range(1, len(input)):
        a.append(j)
        print(j)

# Time: O(n^2)
# Space: O(n^2) # because the amount of values stored in the array `a` grows quadratically with the input size. (for
input size 2 -> 4 elements will be added to the array a)

# ==============================
input = [1,2,3]
a = []
for i in range (1, len(input)):
    for j in range(1, len(input)):
        break
        a.append(j)
        print(j)

# Time: O(n) # the inside for loop is not executed because of the for loop
# Space: O(1) # The append function is never executed

# ==============================
input = [1,2,3]
a = []
for i in range (1, len(input)):
    for j in range(1, len(input)):
        if j == 2:
            continue
        print(j)

# Time: O(n^2) # Actually it's (n^2 - 1) because of the continue statement
    # but as n tends to infinity, the -1 becomes insignificant, therefore the time complexity is O(n^2)
# Space: O(1)
```

Searching Algorithms

Searching algorithms are crucial in computing because they help businesses and organizations efficiently manage and retrieve information from large datasets.

Imagine you run a large retail company with thousands of products in your inventory. Your customers want to quickly find and purchase the products they need.

If you didn't have an efficient way to search through your inventory, it would be incredibly time-consuming and frustrating for your customers to locate the items they're looking for, leading to potential lost sales and dissatisfied customers.

Searching algorithms provide a systematic and efficient way to locate specific data or information within a large collection, similar to how you might use an index or table of contents in a book to quickly find a particular topic or chapter.

The two most common searching algorithms are:

1. Linear Search
2. Binary Search

Linear Search

Time Complexity: O(n) - Linear
Space Complexity: O(1) - Constant

It works by sequentially checking each element in the list until the desired element is found or the end of the list is reached.

Real life example:
You can think of linear search as similar to how you might search for a specific book in a library by checking each bookshelf one by one from the start until you find the book you're looking for.

NOTE: The input list does not need to be sorted for linear search to work.

```
def linear_search(arr, target):
    # The triple quote is a python string to specify what a function does
    """
    Performs a linear search on the given list 'arr' to find the 'target' element.
    Returns the index of the target element if found, otherwise returns -1.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i # Return the index of the target element:
            # Just like break statement, return statement will immediately exit the function itself
    return -1

# Usage of linear search
my_list = [5, 2, 9, 1, 7]
target = 9

index = linear_search(my_list, target)
if index == -1:
    print(f"Target {target} not found in the list")
else:
    print(f"Target {target} found at index {index}")
```

Binary Search

Time Complexity: O(log n) - Logarithmic
Space Complexity: O(1) - Constant

The binary search algorithm is an efficient way to find a target element in a **sorted list** or array.

It works by **repeatedly dividing the search interval in half** until the target element is found or it is determined that the element is not present in the list.

Real life example:\

Imagine you have a large dictionary (a book containing words and their definitions in alphabetical order), and you want to find the definition of a specific word. Instead of going through the dictionary page by page from the beginning (linear search), you can utilize the binary search approach.

Steps to perform binary search for the example above:

- Open the dictionary in the middle.
- Compare the word you're looking for with the word on the currently opened page.
- If the word you're looking for comes before the word on the current page, you know the word must be in the first half of the dictionary, so you can discard the second half.

- If the word you're looking for comes after the word on the current page, you know the word must be in the second half of the dictionary, so you can discard the first half.
- Repeat steps 1-4, but this time, open the dictionary in the middle of the remaining half.

```
def binary_search(arr, target):
    """
    Performs a binary search on the sorted list 'arr' to find the 'target' element.
    Returns the index of the target element if found, otherwise returns -1.
    """
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1

my_list = [1, 3, 5, 7, 9]
target = 7

index = binary_search(my_list, target)
if index != -1:
    print(f"Target {target} found at index {index}")
else:
    print(f"Target {target} not found in the list")
```

Visualization:

```
Initial list: [1, 3, 5, 7, 9]
               0  1  2  3  4
               low       high

Iteration 1:
mid = (0 + 4) // 2 = 2 (index of 5)
5 < 7, so we update low = mid + 1 = 3

List: [1, 3, 5, 7, 9]
          low    high

Iteration 2:
mid = (3 + 4) // 2 = 3 (index of 7)
7 == 7, target found at index 3

# ================================================
```

Visualization:

```
Initial list: [1, 3, 5, 7, 9]

                  mid
                   |
                   v
            [1, 3, 5, 7, 9]
             ^           ^
             |           |
            low         high


After Iteration 1:
            [1, 3, 5, 7, 9]
                     ^  ^
                     |  |
                    low high

After Iteration 2 (Target Found):
            [1, 3, 5, 7, 9]
                     ^
                     |
                    mid
```

Sorting Algorithms

Imagine you own an e-commerce company that sells a wide range of products. Your online store has a vast inventory, and customers can search for products based on various criteria, such as price, popularity, or ratings.

Without an efficient way to sort the products based on these criteria, it would be challenging for your customers to find what they're looking for, leading to a poor user experience and potentially lost sales.

Bubble Sort

Time Complexity: O(n^2) in the worst case, O(n) in the best case
Space Complexity: O(1)

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

Let's use a real-life example to understand the intuition behind bubble sort:

Imagine you have a group of friends standing in a line, but they are not arranged in order of height. You want to rearrange them from shortest to tallest (or vice versa).

Here's how you could use the bubble sort approach:

- Start from the beginning of the line.
- Compare the heights of the first two people. If the person on the right is shorter than the person on the left, swap their positions.
- Move to the next pair of adjacent people and repeat step 2.
- Continue this process, moving along the line and swapping adjacent people if they are in the wrong order.
- After reaching the end of the line, you will have bubbled up (or down) the tallest (or shortest) person to the correct position.
- Repeat steps 2-5 for the remaining unsorted portion of the line until the entire line is sorted.

Visualization:

```
# Imagine the following heights represent your friend's height in a line:
Initial line: [68, 62, 72, 65, 70, 61]

Iteration 1:
[62, 68, 65, 70, 61, 72]  (Swapped 68 and 62, 72 bubbled to the end)
  ^   ^

Iteration 2:
[62, 65, 68, 61, 70, 72]  (Swapped 68 and 65, 70 bubbled up)
      ^   ^

Iteration 3:
[62, 65, 61, 68, 70, 72]  (Swapped 68 and 61)
          ^   ^

Iteration 4:
[62, 61, 65, 68, 70, 72]  (Swapped 65 and 61)
      ^   ^

Iteration 5:
[61, 62, 65, 68, 70, 72]  (Swapped 62 and 61)
  ^   ^

Sorted line: [61, 62, 65, 68, 70, 72]
```

Python Implementation:

```
def bubble_sort(arr):
    """
    Sorts the given list 'arr' in ascending order using the bubble sort algorithm.
    """
    n = len(arr)

    # Iterate through the list n-1 times
    for i in range(n-1):
        # Last i elements are already sorted
        for j in range(n-i-1):
            # Traverse the list from 0 to n-i-1
            # Swap if the element is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]

    return arr

my_list = [64, 34, 25, 12, 22, 11, 90]

sorted_list = bubble_sort(my_list)
print("Sorted list:", sorted_list)

# Output: Sorted list: [11, 12, 22, 25, 34, 64, 90]
```

Video:
NOTE: No need to watch the Java Implementation

Insertion Sort

Time Complexity: O(n^2) in the worst case, O(n) in the best case
Space Complexity: O(1)

The insertion sort algorithm is a simple and intuitive sorting algorithm that works by building a sorted array (or list) one element at a time.

Let's use a real-life example to understand the intuition behind insertion sort:

Imagine you are a card player, and you have a hand of cards that is initially unsorted. You want to sort your cards in ascending order (for example, from Ace to King)

Here's how you could use the insertion sort approach:

- Start by considering the first card as your sorted "hand" (a single card is already sorted).
- Take the next card from the unsorted portion of your hand.

- Insert the new card into the correct position in your sorted "hand" by shifting the cards that are greater than the new card to the right.
- Repeat steps 2 and 3 for each remaining card in the unsorted portion until all cards are inserted into the sorted "hand".

Visualization:

```
Initial hand (unsorted): [5, 2, 9, 1, 7]

Iteration 1:
Sorted hand: [5]
Unsorted cards: [2, 9, 1, 7]
Insert 2 into the sorted hand:
Sorted hand: [2, 5]
Unsorted cards: [9, 1, 7]

Iteration 2:
Sorted hand: [2, 5]
Unsorted cards: [9, 1, 7]
Insert 9 into the sorted hand:
Sorted hand: [2, 5, 9]
Unsorted cards: [1, 7]

Iteration 3:
Sorted hand: [2, 5, 9]
Unsorted cards: [1, 7]
Insert 1 into the sorted hand:
Sorted hand: [1, 2, 5, 9]
Unsorted cards: [7]

Iteration 4:
Sorted hand: [1, 2, 5, 9]
Unsorted cards: [7]
Insert 7 into the sorted hand:
Sorted hand: [1, 2, 5, 7, 9]
Unsorted cards: []

Final sorted hand: [1, 2, 5, 7, 9]
```

Video:

NOTE: No need to watch the Java Implementation

Python Implementation:

```python
def insertion_sort(arr):
    """
    Sorts the given list 'arr' in ascending order using the insertion sort algorithm.
    """
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Shift elements greater than 'key' to the right
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        # Insert 'key' at the correct position
        arr[j + 1] = key

    return arr

my_list = [5, 2, 9, 1, 7]

sorted_list = insertion_sort(my_list)
print("Sorted list:", sorted_list)

# Output: Sorted list: [1, 2, 5, 7, 9]
```