## Question 1 - An important feature of Object Oriented Programming is using codes as objects. Describe an example of how objects are used in Python programming.

Object-Oriented Programming (OOP) is a paradigm that uses "objects" to design and manage software. These objects are instances of classes, which are blueprints defining the properties and behaviors of the objects.

If we are developing a software system for a bank, the customers will have accounts, and they will be able to manage their bank accounts.

In an OOP approach, we will create a class for the bank accounts and it will have certain properties and methods that reflect the functions of an actual bank account. If we name the class BankAccount then each of the accounts will be an object of the class. Using objects help us utilize the 4 pillars of OOP, inheritance, abstraction, encapsulation and polumorphism.

## Question 2

**1. A program generates all the combinations of elements of a set and writes the output to a file at a rate of 1100 combinations per second. How long will it take to generate all the combinations of a set with 4 distinct elements?**

First of all 4 distinct elements can generate 2^4 = 16 numbers of elements. And if we can write the numbers to a file at the rate of 1100 combinations per second, then we will need 16/1100 = 0.0145 seconds.

**2. What is the Big-O time complexity of the given arithmetic function? Show working.**

Big-O time complexity of the given arithmetic function,

Given Function,

def Arithmetic(n):

    a = 0

    for i in range(1, n//2):

        for j in range(1, i//4):

            a += n\*\*i - j

    return a

Analysis

* a = 0 is a constant-time operation, **O(1)**.
* The first for loop runs from **1 to n//2**, so it executes **O(n/2)** times. In Big-O notation, we simplify constants, so this is **O(n)**.
* The second for loop runs from **1 to i//4**. The number of iterations depends on the current value of  **i**. Specifically, it runs **O(i/4)** times. Again, simplifying constants, this is **O(i)**.
* **a += n\*\*i - j** is a constant-time operation, so **O(1)** is the time complexity for each iteration of the inner loop.

Combining the Loops, to find the overall time complexity, we need to consider the total number of iterations of the inner loop across all iterations of the outer loop.

- The outer loop runs **O(n)** times.

- The inner loop, for a given **i**, runs **O(i)** times.

We sum the number of iterations of the inner loop over all iterations of the outer loop:

This is an arithmetic series. The sum of the first k integers is given by:

In our case, k = n/2, so,

Simplifying the constant factor, the overall time complexity is:

## Question 3

**1. Write pseudocode or code for an algorithm to find the product of a sequence, P, of n integers.**

Pseudocode to find the product of a sequence of **n** integers:

function calculateProductOfSequence(P, n)

product ← 1

for i from 1 to n do

product ← product \* P[i]

end for

return product

end function

**2. What are the time complexity and space complexity of your algorithm? Note that this will depend on your code.**

Time Complexity

* The initialization of product variable is a constant-time operation, **O(1)**.
* The loop runs **n** times, where **n** is the number of integers in the sequence **P**. Each iteration of the loop involves a multiplication operation, which is **O(1)** in terms of time complexity.

So, the overall time complexity is linear, **O(n)**.

Space Complexity

We use a constant amount of additional space for the **product** variable and the loop control variables. The space used does not depend on the size of the input **n**.

So, the overall space complexity is constant, **O(1)**.

## Question 4

**1. Write a recursive function to generate the nth Fibonacci number.**

This function that uses recursion to generate the nth Fibonacci number:

def fibonacci(n):

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    else:

        return fibonacci(n-1) + fibonacci(n-2)

**2. What are the time complexity and space complexity of this function?**

Time Complexity:

The time complexity of this recursive function is O(2n). Because this function makes two recursive calls if the base cases are not found. This makes the calls grow exponentially.

Space Complexity:

The space complexity is O(n). In the worst case scenario, the maximum depth of the recursion would be n, that is why the space complexity would be O(n).

**3. Is this method more efficient than the iterative method?**

Iterative implementation:

def fibonacci\_iterative(n):

    if n <= 0:

        return 0

    elif n == 1:

        return 1

    else:

        a = 0

        b = 1

        for i in range(2, n + 1):

            a = b

            b = a + b

        return b

Time Complexity:

The iterative implementation is linear and uses only one loop. So, the time complexity is O(n).

Space Complexity:

This method uses a fixed number of variables, and updates them in each step of the loop which makes the space complexity constant. So the space complexity is O(1).

From the above comparison we can determine that the iterative method (using a loop) is much faster than the recursive method because it doesn't repeat calculations, and it also uses less memory in comparison to the recursive method.

So, in my opinion the iterative method would be the more efficient one than the other.

## Question 5

**What is the complexity of the queue and dequeue operations on a queue? Please create a data structure for a queue with different complexity.**

Complexity of queue and deque operations on a queue implemented in C/C++:

For adding a new item to the queue we would use the Linked List data structure which would add a new item to the end of the list and take constant time. So the time complexity will be O(1) for enqueue.

For removing an item from the front of the linked list would also take constant time because we are just simply removing the link between the first two elements. So the time complexity will be O(1) for deque.

As an alternate data structure to implement queue we could use the Array data structure. When using an array data struture to add a new item to the queue would take O(n) time because we need to increase the size of the array an then add the new item at the very end. So the time complexity would be O(n).

And to remove an element from the queue would also take linear time because we will have shift the later element after removing the first element. So the deque operation would also take O(n) time compleity.

## Question 6

**1. Write pseudocode for calculating the Greatest Common Divisor (GCD) of two numbers.**

Pseudocode for calculating the Greatest Common Divisor (GCD) of two numbers,

function calculateGCD(a, b)

result ← min(a, b)

while result > 0 do

if a mod result = 0 and b mod result = 0 then

break

else

result ← result – 1

end if

end while

return result

end function

**2. What is the time complexity of this calculation in Big-O notation?**

Time complexity determination:

- result = min(a, b) takes constant time

- Then the while loop runs for min(a, b) times in the worst case because if in the wrost case we might need to check each number in the range of the smaller number. So, the time complexity is O(min(a, b))

- If the condition is not met, result = result – 1 takes constant time.

So, in the worst case, the time complexity is O(min(a, b)).

**3. Show that the output of this algorithm is indeed the Greatest Common Divisor.**

The GCD (Greatest Common Divisor) of two numbers is the largest number that divides both of the numbers. This algorithm is based on the following mathematical property of the GCD:

Let's assume that the inputs of the algorithm are a = 48 and b = 18.

- Declare a variable result and initialize it to min(a, b) = min(48, 18) = 18

- For result = 18 and result > 0, do

- As 48 mod 18 = 12 and 18 mod 18 = 0, go to the next iteration of the loop with result = 17

- For result = 17 and result > 0, do

- As 48 mod 17 = 14 and 18 mod 17 = 1, go to the next iteration of the loop with result = 16

- For result = 16 and result > 0, do

- As 48 mod 16 = 0 and 18 mod 16 = 2, go to the next iteration of the loop with result = 15

- For result = 15 and result > 0, do

- As 48 mod 15 = 3 and 18 mod 15 = 3, go to the next iteration of the loop with result = 14

- For result = 14 and result > 0, do

- As 48 mod 14 = 6 and 18 mod 14 = 4, go to the next iteration of the loop with result = 13

- For result = 13 and result > 0, do

- As 48 mod 13 = 9 and 18 mod 13 = 5, go to the next iteration of the loop with result = 12

- For result = 12 and result > 0, do

- As 48 mod 12 = 0 and 18 mod 12 = 6, go to the next iteration of the loop with result = 11

- For result = 11 and result > 0, do

- As 48 mod 11 = 4 and 18 mod 11 = 7, go to the next iteration of the loop with result = 10

- For result = 10 and result > 0, do

- As 48 mod 10 = 8 and 18 mod 10 = 8, go to the next iteration of the loop with result = 9

- For result = 9 and result > 0, do

- As 48 mod 9 = 3 and 18 mod 9 = 0, go to the next iteration of the loop with result = 8

- For result = 8 and result > 0, do

- As 48 mod 8 = 0 and 18 mod 8 = 2, go to the next iteration of the loop with result = 7

- For result = 7 and result > 0, do

- As 48 mod 7 = 6 and 18 mod 7 = 4, go to the next iteration of the loop with result = 6

- For result = 6 and result > 0, do

- As 48 mod 6 = 0 and 18 mod 6 = 0, the condition is satisfied, break the loop

- The loop exits with result = 6

- Return result = 6

Thus, the Greatest Common Divisor (GCD) of 48 and 18 is indeed 6, which matches the output of the algorithm.