

Analysis of Common Loop

Analysis of Common Loops: While Loop in Python

Loops are essential constructs in programming that allow us to repeat a block of code multiple times. Python offers several types of loops, and one of the most versatile is the **while** loop. The **while** loop continues to execute a block of code as long as a given condition remains true. In this article, we will analyze and demonstrate the use of the while loop for subtraction, multiplication, exponentiation, and nested loops in Python.



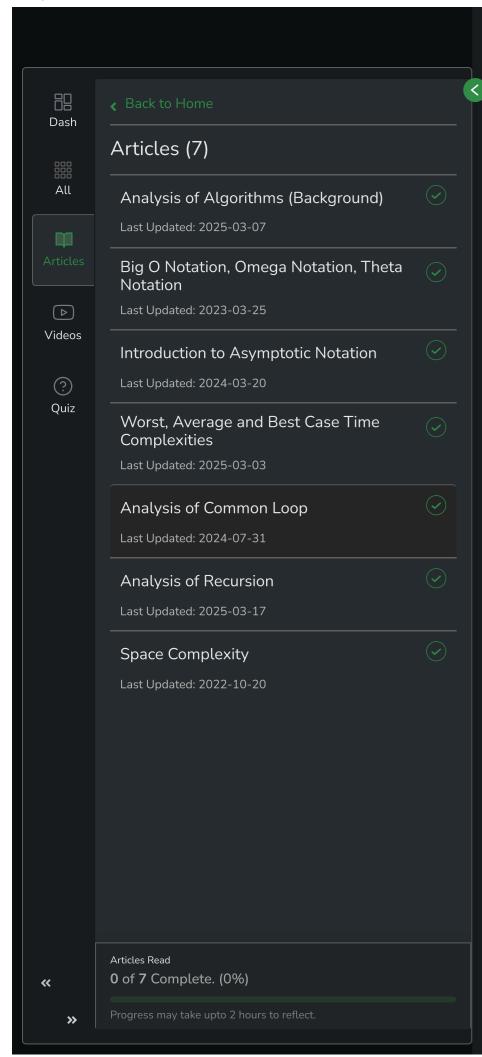
Addition using a While Loop

Let's begin by examining how to utilize a while loop for addition. The following code demonstrates a simple implementation of adding two numbers using a while loop:

```
Python
        def add(a, b):
result = a
               while b > 0:
                    result += 1
                    print(result)
                    b -= 1
       10 \quad a = 10
       11 \quad b = 7
       12 add(a,b)
```

Output

- 11
- 12
- 13
- 14
- 15



16

17

In the code above, we define a function called **add** that takes two numbers, **a** and **b**, as parameters. We initialize the variable **result** with the value of **a**. The **while** loop executes **b** times, incrementing the **result** by 1 on each iteration while decrementing **b** by 1. Finally, the function returns the added value.

Time complexity analysis:

Addition using a While Loop:

- Time Complexity: O(b)
- Similar to multiplication, the **while** loop executes **b** times, where **b** is the value being added. Thus, the time complexity is linear and proportional to **b**.

Subtraction using a While Loop

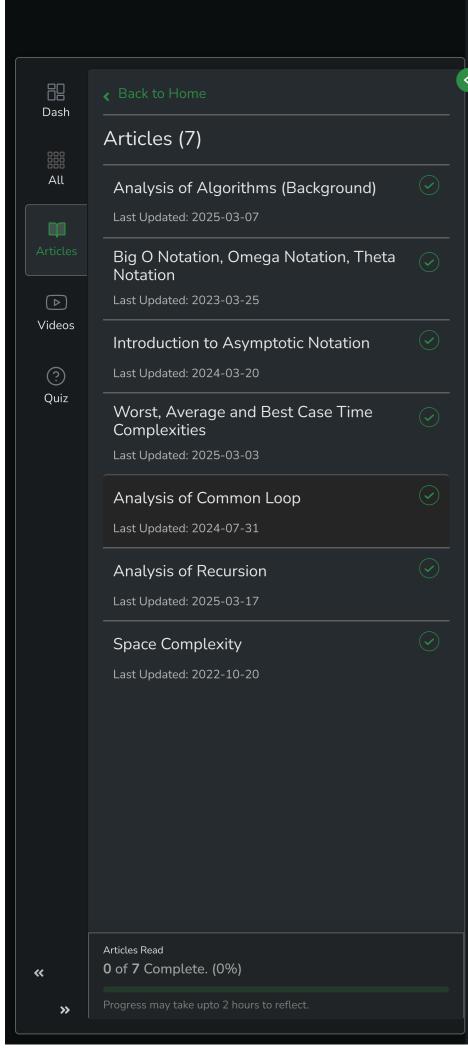
Let's start by examining how to use a while loop for subtraction. The following code demonstrates a simple implementation of subtracting two numbers using a **while** loop:

```
0
          def subtract(a, b):
              result = a
while result >= b:
                print(result)
                result -= b
      10 a = 15
      11 \quad b = 7
      12 subtract(a, b)
```

Output

15 8

In the code above, we define a function called **subtract** that takes two numbers, **a** and **b**, as parameters. We initialize the variable result with the value of a. The while loop continues subtracting b from result until result becomes less than b. Finally, the function returns the subtracted value.

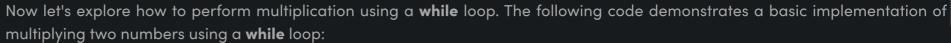


Time complexity analysis:

Subtraction using a While Loop:

- Time Complexity: O(a / b)
- In the worst case scenario, the number of iterations required to subtract **b** from **a** using a **while** loop is **a** / **b**. This assumes that **a** is significantly larger than **b**. The time complexity is linear and depends on the magnitude of **a** and **b**.

Multiplication using a While Loop





Output

2 4

8

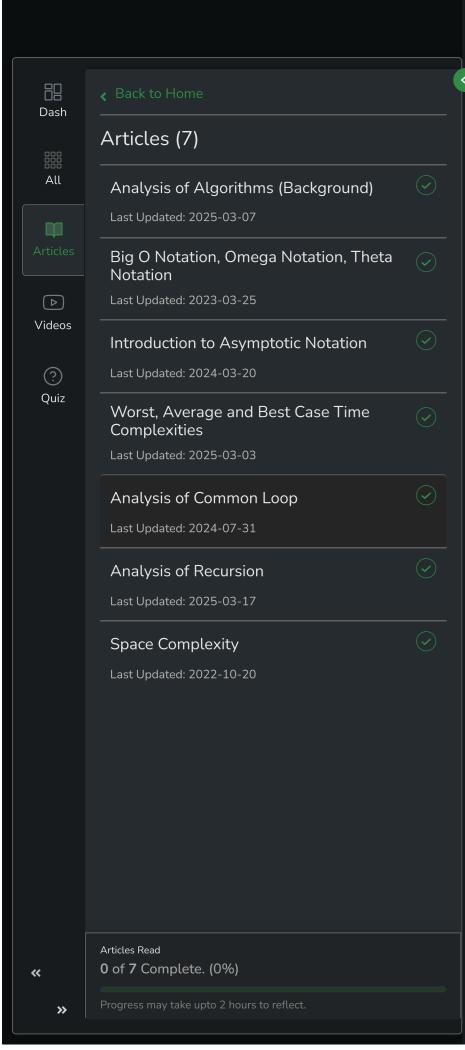
16

In the code above, we define a function called **multiply** that takes two numbers, n and 1, as parameters. We initialize the variable **result** with 0. The **while** loop executes upto result value not greater than n, multiply result with c Finally, the function prints the value of multiplication.

Time complexity analysis:

Multiplication using a While Loop:

- 1. Let's say the number of iteration be K so the value be: 1,C,C^2,C^3,C^K-1
- 2. C^K-1< n



- 3. **K<logn+1**
- 4. Growth of order is O(logn)
- Time Complexity: O(logn)

Exponentiation using a While Loop

Next, let's see how to perform exponentiation using a **while** loop. The following code demonstrates a basic implementation of calculating the exponentiation of a number using a **while** loop:

Output

2

4

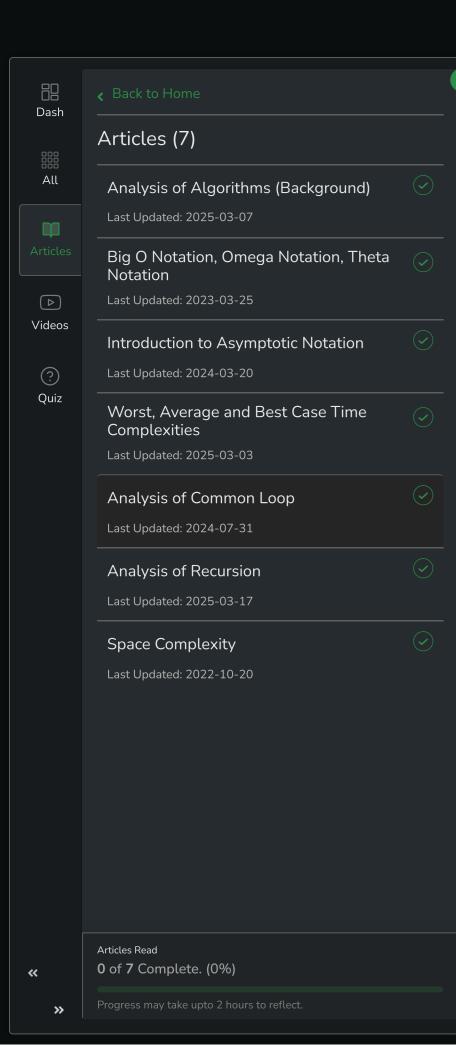
16

In the code above, we define a function called **exponentfun** that takes two numbers, **base** and **exponent**, as parameters. We initialize the variable **result** with 2. The **while** loop executes **exponent** times, multiplying the **result** by **base** on each iteration while result is not greater than n or exponent, Finally, the function returns the exponentiated value.

Time Complexity Analysis:

Exponentiation using a While Loop:

- 1. Time Complexity: O(loglogn)
- 2. if this function iterates K time then values comes up to be 2,2^C,(2^C)^C ... ((2^C)^2...))^C...
- 3. **2^CK-1<n**
- 4. C^K-1<log2n
- 5. k<loglogn+1



- 6. T=O(loglogn)
- Time Complexity: O(loglogn)

Nested While Loops

Python allows the usage of nested loops, including nested **while** loops. A nested loop is a loop within another loop. This type of loop structure can be useful for handling complex scenarios that require multiple iterations. Here's an example of a nested **while** loop:

```
n = int(input())

i = 0

while i <= 5:

j = 1

while j <= n

j += 1

i += 1
```

Some constant work is done in the nested loop.

```
j = 1
while j <= n:
j += 1
```

- 1. Time complexity of given nested loop O(N)
- 2. In case of nested loop we do multiply time complexity of of both loops
- 3. T=O(n)*O(N)=O(N^2)

Nested While Loops:

Time Complexity: O(N^2)

Conclusion

The **while** loop is a powerful construct in Python that allows us to repeat a block of code as long as a specific condition remains true. In this article, we explored the use of **while** loops for subtraction, multiplication, exponentiation, and nested loops. By leveraging the flexibility of the **while** loop, you can implement various algorithms and handle different scenarios efficiently. Keep in mind that while using **while** loops, it's essential to ensure the condition eventually becomes false to avoid infinite loops.

