Name: **Kushal Dilip Kothari**
Net ID: **kk5268**
N No: **N15066497**

PROGRAMMING LANGUAGES: Assignment 3 - Tail Recursion

Recursion: A recursive function is any function that calls an instance of itself.

Tail Recursion: In tail recursion, no other operation is needed after the successful execution of a recursive function call. The function directly returns the result of a recursive call without performing any operations on it.

Non-tail Recursion: In non-tail recursion, some operations must be performed after successfully executing a recursive function. The function never directly returns the result of a recursive call. It performs some operations on the returned value of the recursive call to achieve the desired output.

Tail recursion is considered better than non-tail recursion because tail recursive functions can be optimized by modern compilers.While non-tail recursion fully utilizes the stack frame and uses the value returned from the recursive call. JVM must retain all the stack frames, no matter how many they are, to compute the end result correctly. This leads to memory overuse and sometimes results in errors.

NON-TAIL RECURSION:

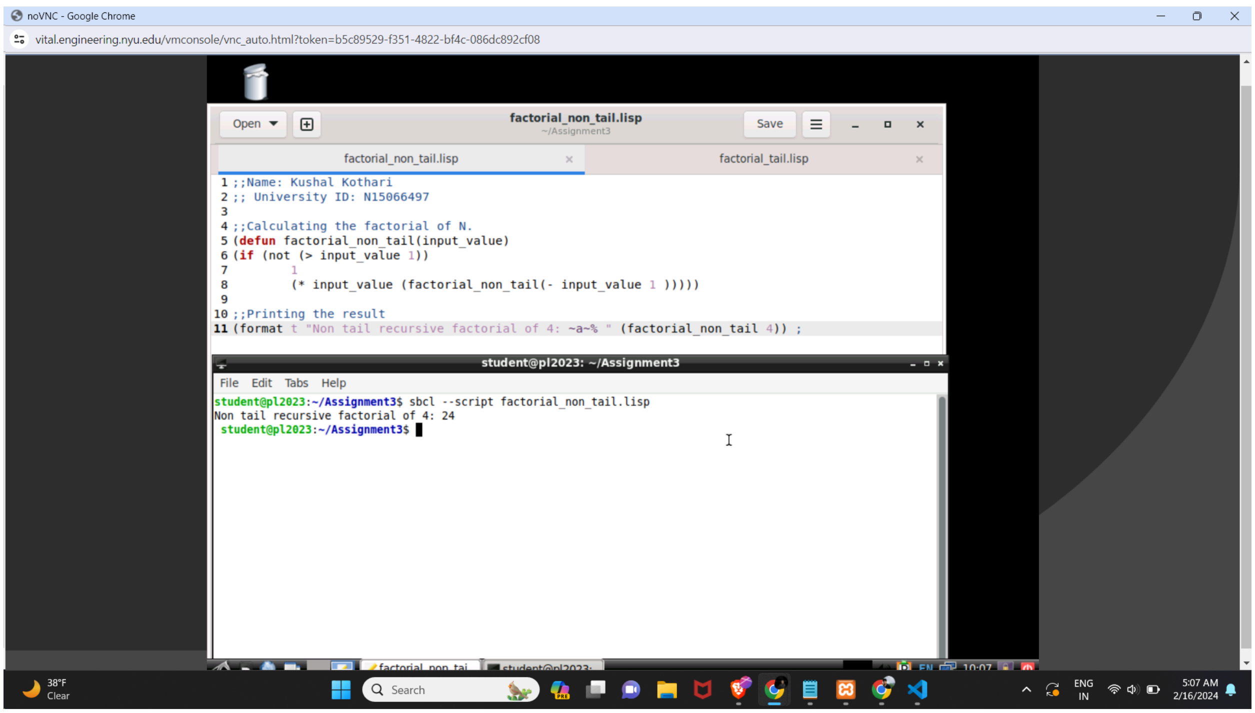
<u>Code with small value as input</u>

```
;; Name: Kushal Kothari
;; University ID: N15066497

;; Calculating the factorial of N.
(defun factorial_non_tail(input_value)
  (if (not (> input_value 1))
      1
      (* input_value (factorial_non_tail(- input_value 1)))))

;; Printing the result
(format t "Non tail recursive factorial of 4: ~a~%" (factorial_non_tail 4))
```

Approach:

The provided Common Lisp code defines a function factorial_non_tail that calculates the factorial of an integer using a non-tail recursive approach. If the input is greater than 1, the function recursively multiplies the input by the factorial of the next lower integer until it reaches 1. The result for the factorial of 4 is then printed to the console. Non-tail recursion is not memory efficient for large numbers due to the accumulation of deferred operations on the call stack.The factorial_non_tail function checks if input_value is not greater than 1 using the condition (not (> input_value 1)), which is equivalent to checking if input_value is less than or equal to 1.

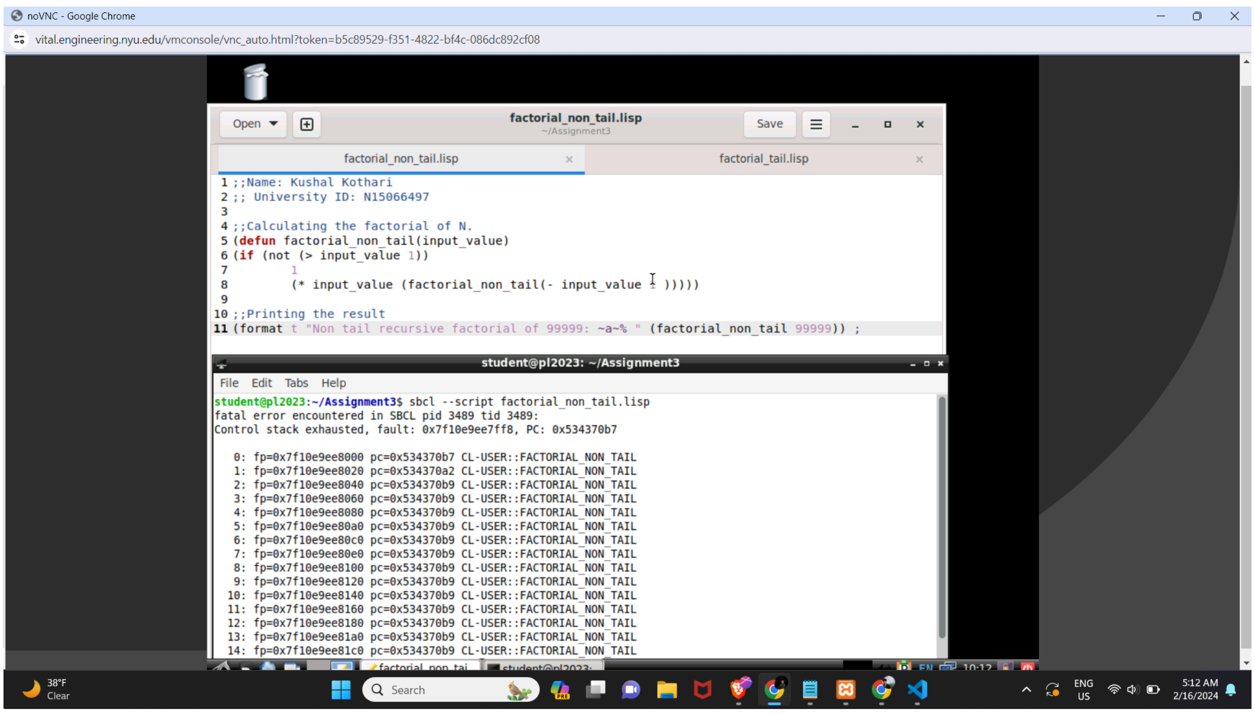
OUTPUT WHEN INPUT IS SMALL



Code with largevalue as input

```lisp
;; Name: Kushal Kothari
;; University ID: N15066497

;; Calculating the factorial of N.
(defun factorial_non_tail(input_value)
  (if (not (> input_value 1))
      1
      (* input_value (factorial_non_tail(- input_value 1)))))

;; Printing the result
(format t "Non tail recursive factorial of 99999: ~a~%" (factorial_non_tail 99999))
```

OUTPUT WHEN INPUT IS LARGE

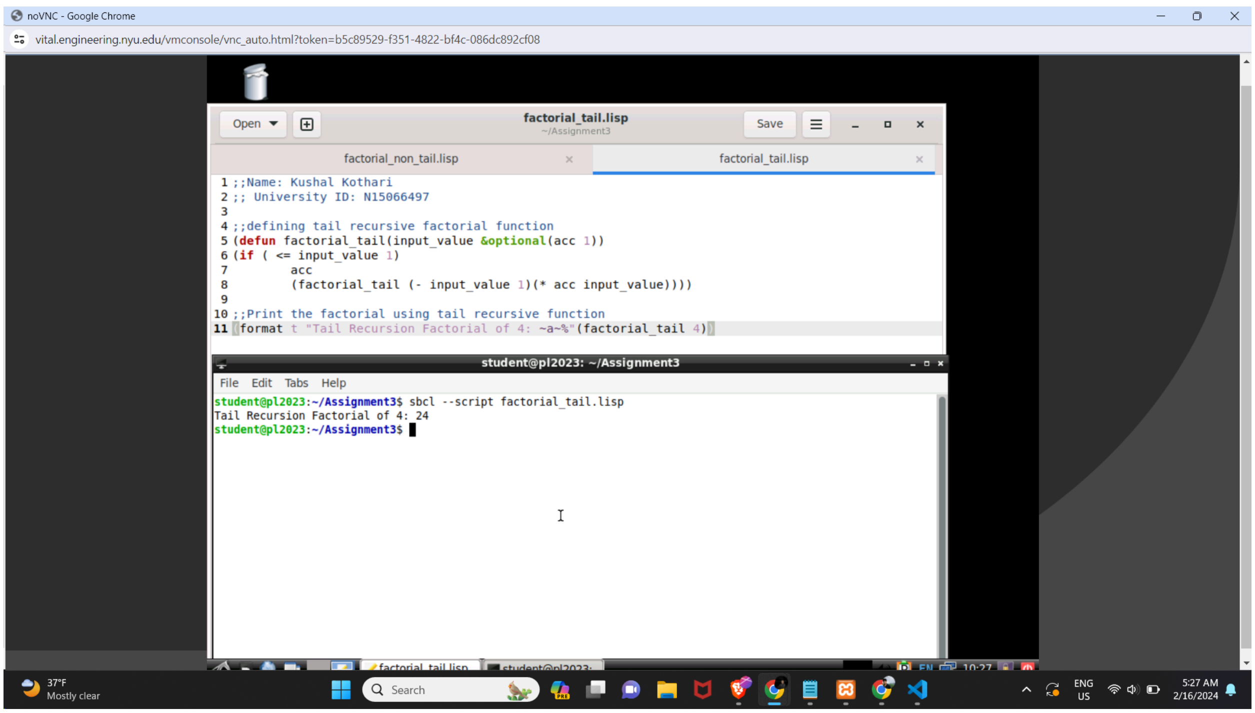


The screenshot displays a Common Lisp programming environment where an error is thrown during the execution of a non-tail recursive factorial function. The function is attempting to calculate the factorial of 99,999, a very large number that leads to an extensive number of recursive calls. The error message "Control stack exhausted" indicates that the program has run out of stack space, a common issue with recursive functions that do not employ tail recursion optimization. This demonstrates the limitations of non-tail recursive functions in handling large recursive depths due to their heavy stack usage.

TAIL RECURSION:

Code:

```
;; Name: Kushal Kothari
;; University ID: N15066497

;; defining tail recursive factorial function
(defun factorial_tail(input_value &optional (acc 1))
  (if (<= input_value 1)
      acc
      (factorial_tail (- input_value 1) (* acc input_value))))

;; Print the factorial using tail recursive function
(format t "Tail Recursion Factorial of 4: ~a~%" (factorial_tail 4))
```

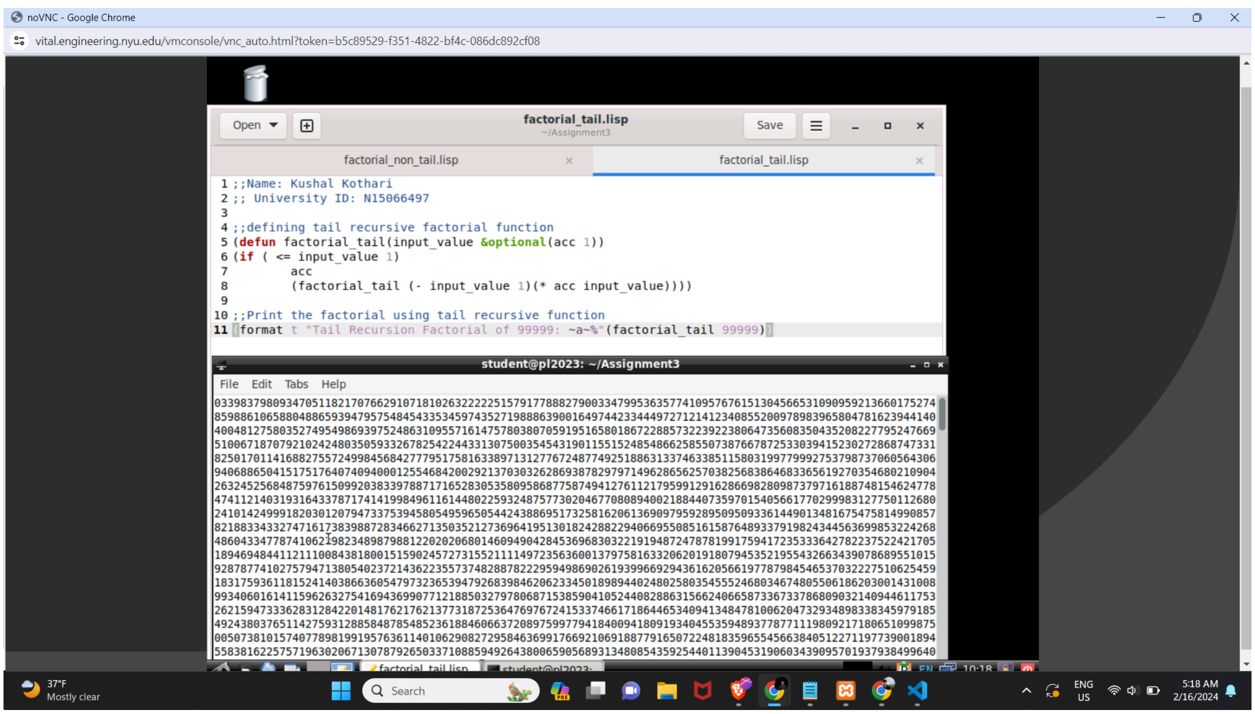


Code Explaination: This method optimizes the computation by using an accumulator to maintain state, allowing the function to handle large numbers without exceeding stack limits. It concludes by printing the factorial of 4.

FOR INPUT VALUE= 99999 i.e. large value

Code:

```
;; Name: Kushal Kothari
;; University ID: N15066497

;; defining tail recursive factorial function
(defun factorial_tail(input_value &optional (acc 1))
  (if (<= input_value 1)
      acc
      (factorial_tail (- input_value 1) (* acc input_value))))

;; Print the factorial using tail recursive function
(format t "Tail Recursion Factorial of 99999: ~a~%" (factorial_tail 99999))
```



This script showcases a well-structured Common Lisp routine that efficiently calculates the factorial of a very large number, specifically 99,999. Leveraging tail recursion, it elegantly avoids the pitfall of stack overflow, which often plagues similar computations in non-optimized scenarios. The optional accumulator parameter is a clever touch, serving as a running total that is neatly carried through each recursive step. The outcome is a testament to the language's capability to handle extensive iterative processes with ease.

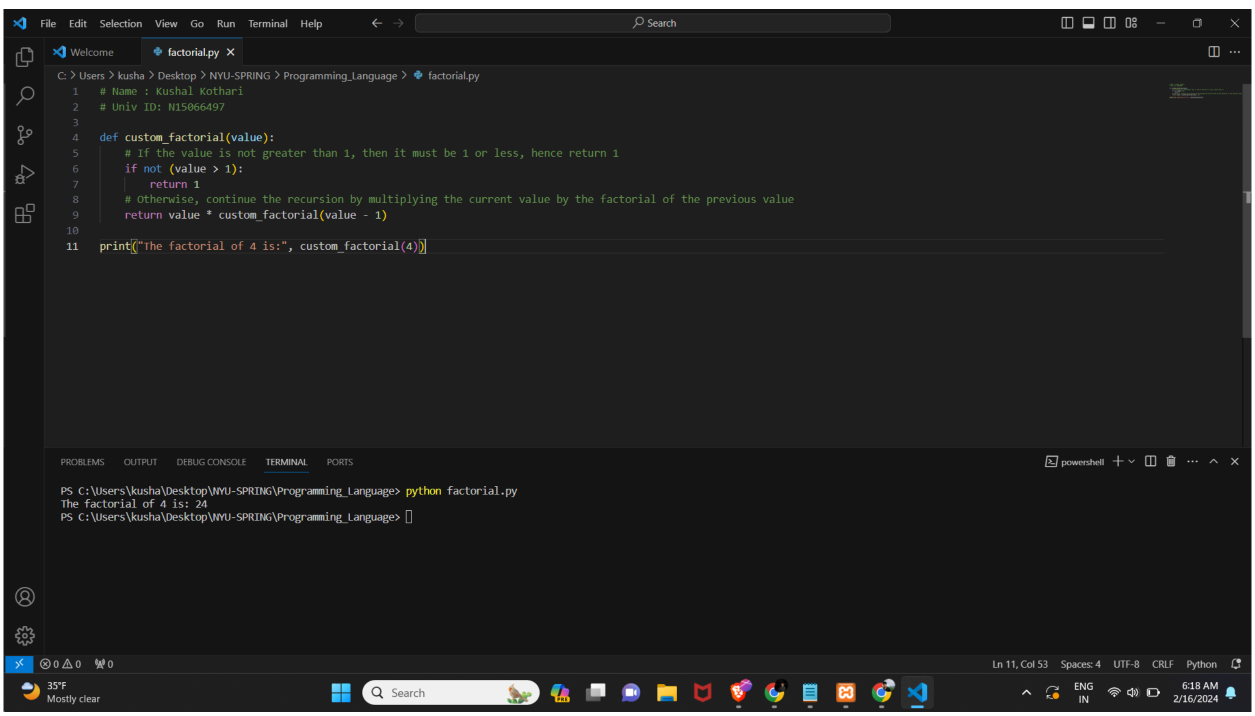
PYTHON:

NON TAIL RECURSION:

Code: small input

```python
# Name : Kushal Kothari
# Univ ID: N15066497

def custom_factorial(value):
    # If the value is not greater than 1, then it must be 1 or less, hence return 1
    if not (value > 1):
        return 1
    # Otherwise, continue the recursion by multiplying the current value by the factorial of the previous value
    return value * custom_factorial(value - 1)

print("The factorial of 4 is:", custom_factorial(4))
```

OUTPUT:

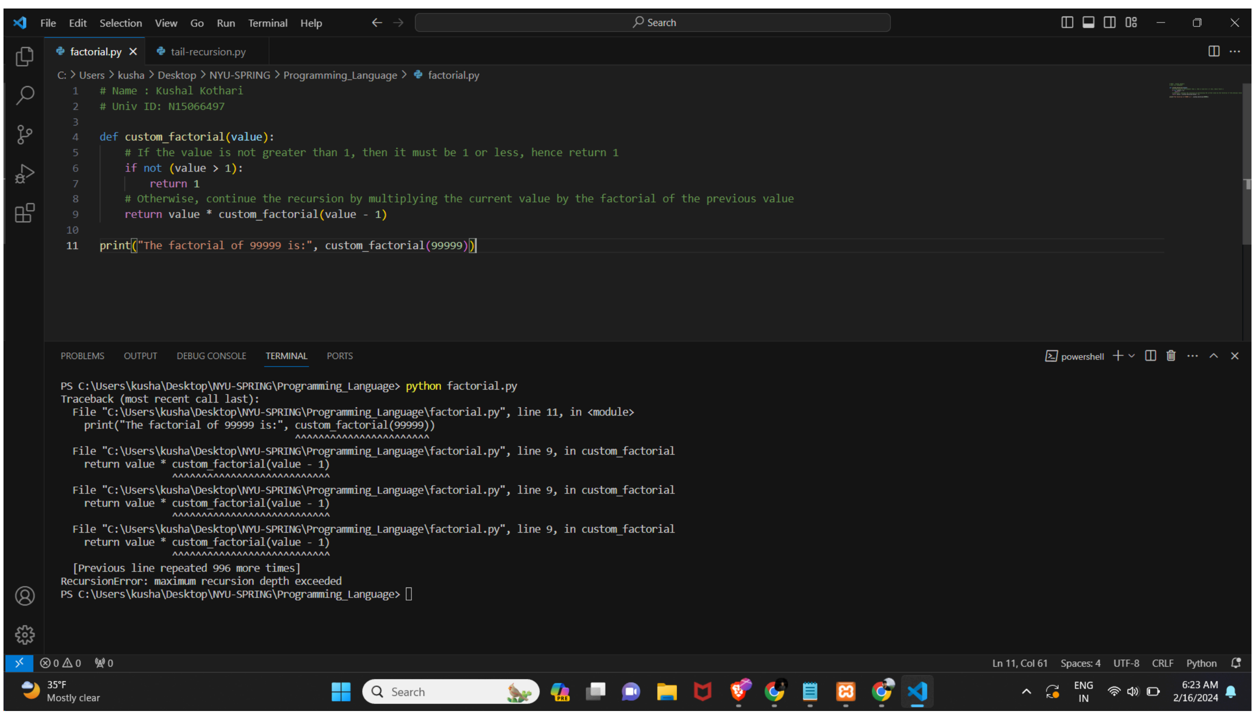


WITH LARGE VALUE as Input

CODE:

```
# Name : Kushal Kothari
# Univ ID: N15066497

def custom_factorial(value):
    # If the value is not greater than 1, then it must be 1 or less, hence
return 1
    if not (value > 1):
        return 1
    # Otherwise, continue the recursion by multiplying the current value
by the factorial of the previous value
    return value * custom_factorial(value - 1)

print("The factorial of 99999 is:", custom_factorial(99999))
```

OUTPUT:

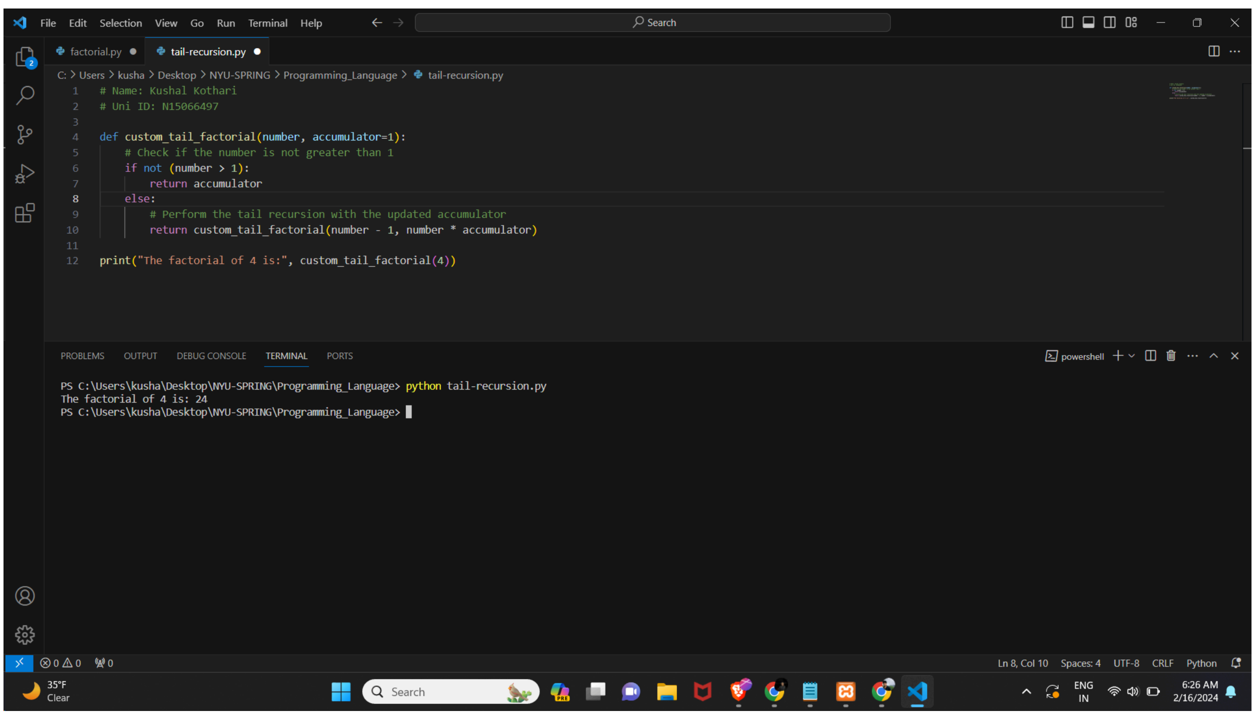


TAIL RECURSION:

Code:
With small value as input

```
# Name: Kushal Kothari
# Uni ID: N15066497

def custom_tail_factorial(number, accumulator=1):
    # Check if the number is not greater than 1
    if not (number > 1):
        return accumulator
    else:
        # Perform the tail recursion with the updated accumulator
        return custom_tail_factorial(number - 1, number * accumulator)

print("The factorial of 4 is:", custom_tail_factorial(4))
```

OUTPUT:

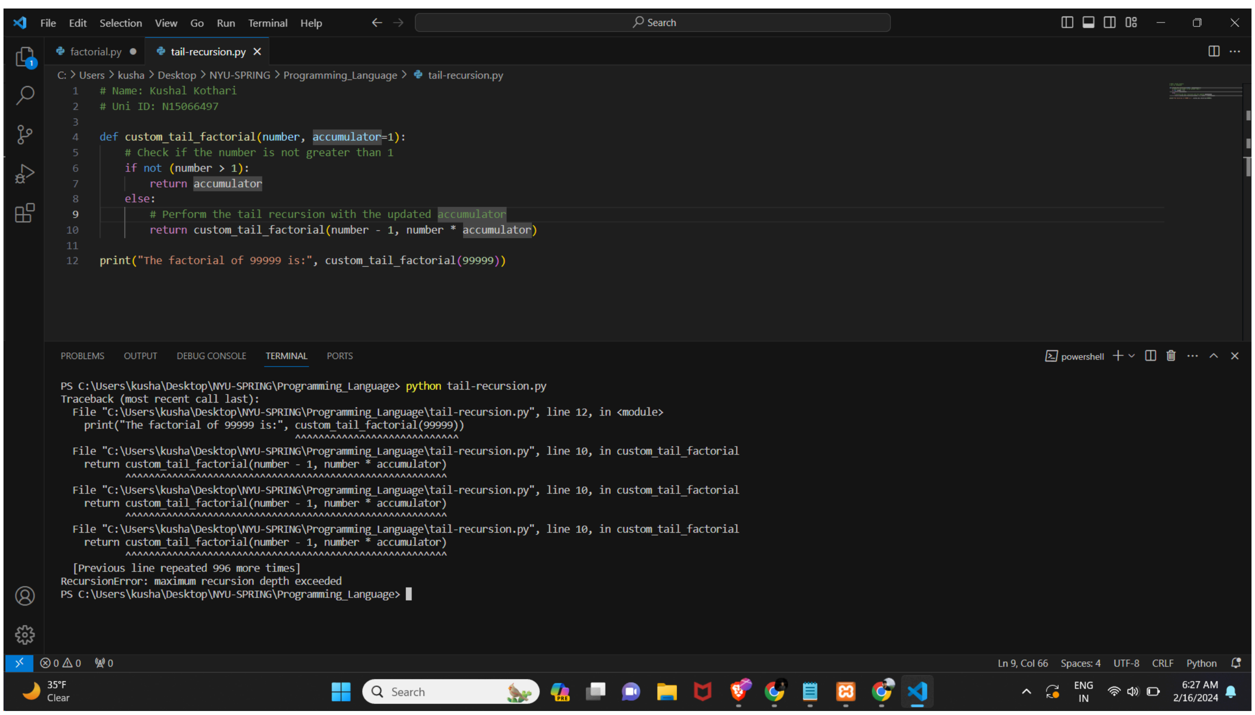


With large value as Input

Code:

```python
# Name: Kushal Kothari
# Uni ID: N15066497

def custom_tail_factorial(number, accumulator=1):
    # Check if the number is not greater than 1
    if not (number > 1):
        return accumulator
    else:
        # Perform the tail recursion with the updated accumulator
        return custom_tail_factorial(number - 1, number * accumulator)

print("The factorial of 99999 is:", custom_tail_factorial(99999))
```

OUTPUT:



CONCLUSION:

In this assignment, we explored recursion by implementing factorial functions in SBCL (Common Lisp) and Python, focusing on non-tail and tail recursion. Our findings revealed that Common Lisp (SBCL) excels in tail recursion optimization, allowing for efficient recursive calls without the risk of stack overflow, even with large numbers. Python, however, does not inherently support tail recursion optimization, making it prone to stack overflow errors in deep recursive calls.

This comparison underscores the importance of choosing the right programming language based on its strengths and limitations, particularly for tasks requiring extensive recursion. While Lisp provides built-in support for tail recursion, making it ideal for such scenarios, Python's approach necessitates careful planning and limitations awareness.