

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:

Code with small value as input

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

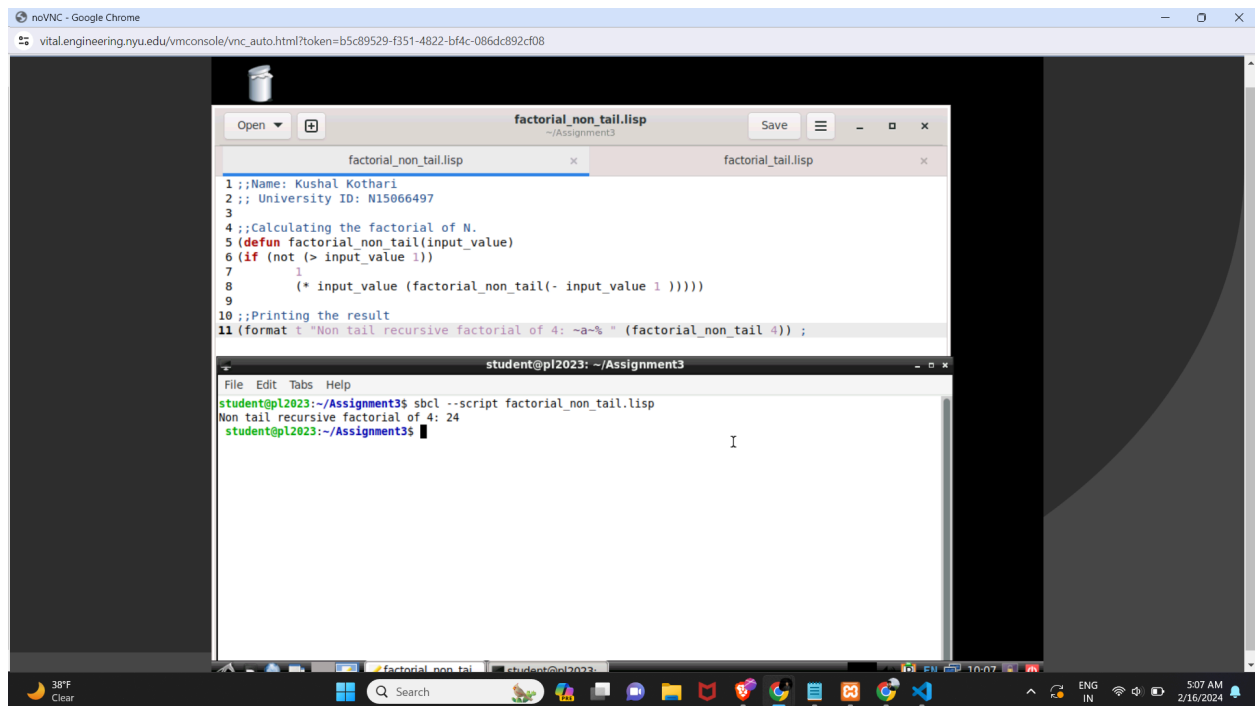
;; Calculating the factorial of N.
(defun factorial_non_tail(input_value)
  (if (not (> input_value 1))
      1
      (* input_value (factorial_non_tail(- input_value 1)))))

;; 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



The screenshot shows a noVNC browser window displaying a Common Lisp script in a file named `factorial_non_tail.lisp`. The script includes comments and a function definition for calculating the factorial of a number using a non-tail recursive approach. The output of the script is displayed in a terminal window below the code editor.

```
1 ;;Name: Kushal Kothari
2 ;; University ID: N15066497
3
4 ;;Calculating the factorial of N.
5 (defun factorial_non_tail(input_value)
6 (if (not (> input_value 1))
7     1
8     (* input_value (factorial_non_tail(- input_value 1 )))))
9
10 ;;Printing the result
11 (format t "Non tail recursive factorial of 4: ~a~%" (factorial_non_tail 4)) ;
```

The terminal output shows the execution of the script using `sbcl --script factorial_non_tail.lisp`, resulting in the message: "Non tail recursive factorial of 4: 24".

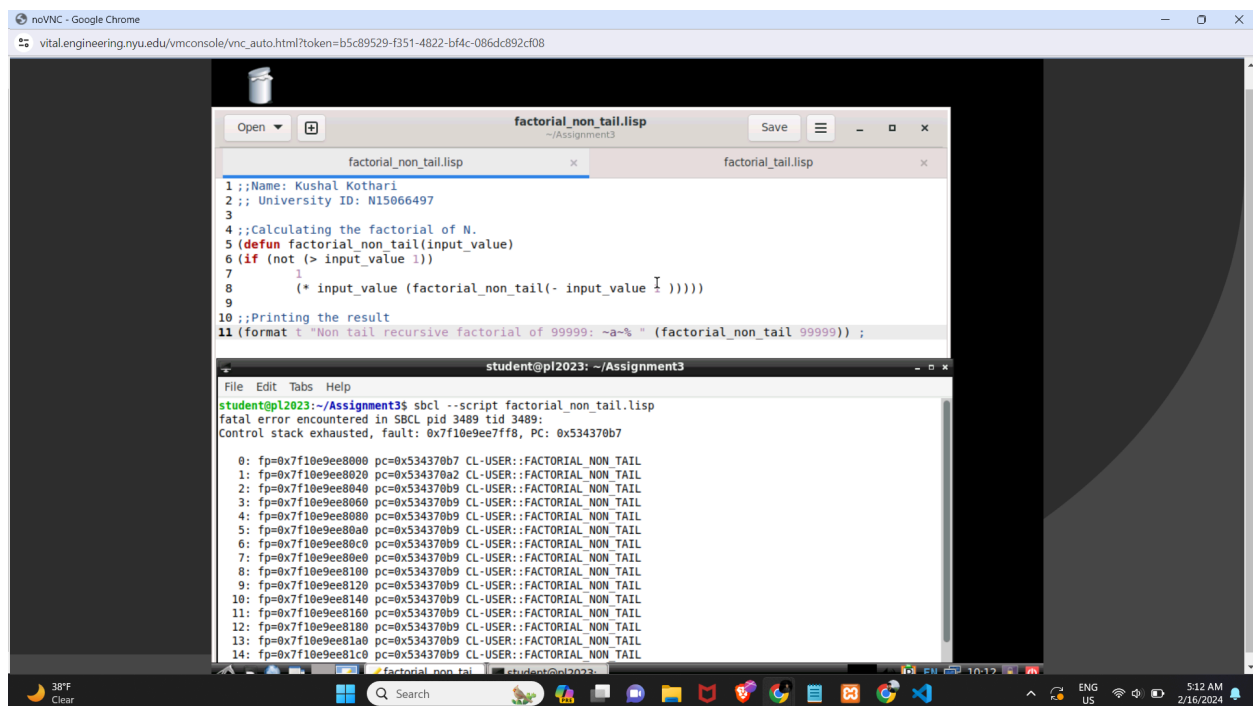
Code with largevalue as input

```
;; 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.

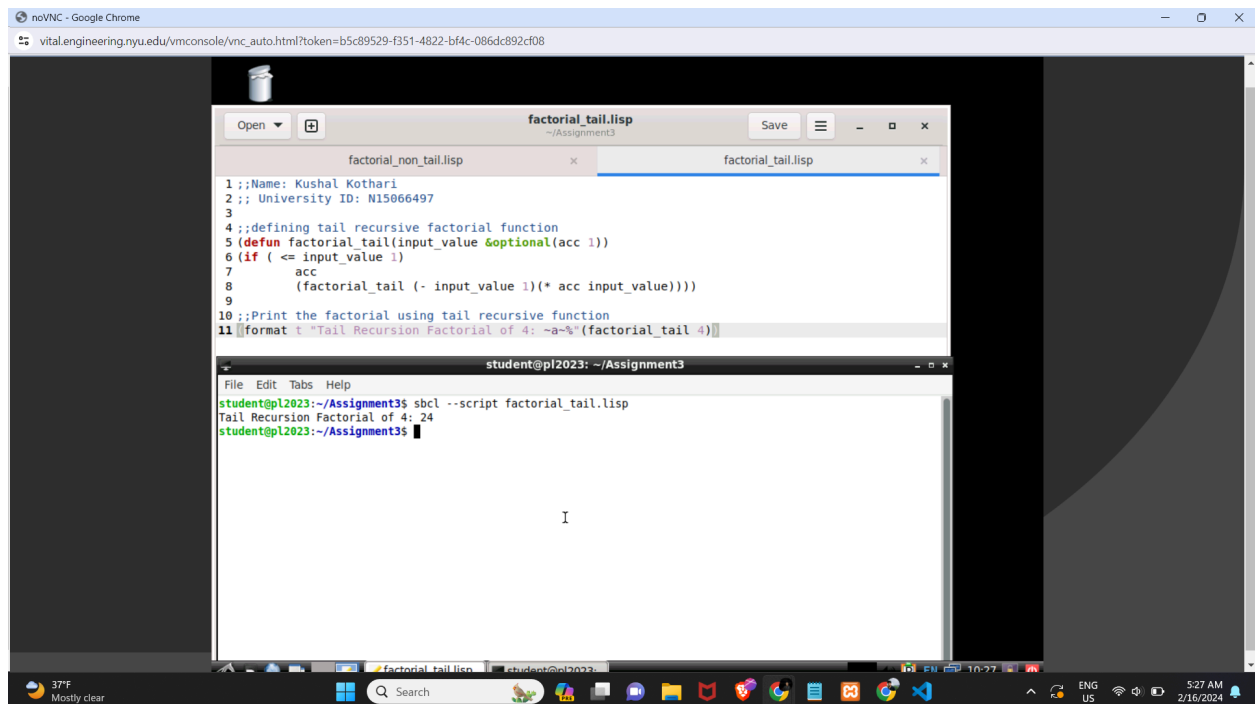
TAIL RECURSION:

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))
```



The screenshot shows a terminal window titled "student@pl2023: ~/Assignment3". The terminal displays the following commands and output:

```
student@pl2023:~/Assignment3$ sbcl --script factorial_tail.lisp
Tail Recursion Factorial of 4: 24
student@pl2023:~/Assignment3$
```

The background of the terminal window shows the source code of the `factorial_tail.lisp` file, which is identical to the code provided in the previous block.

Code Explanation: 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))
```

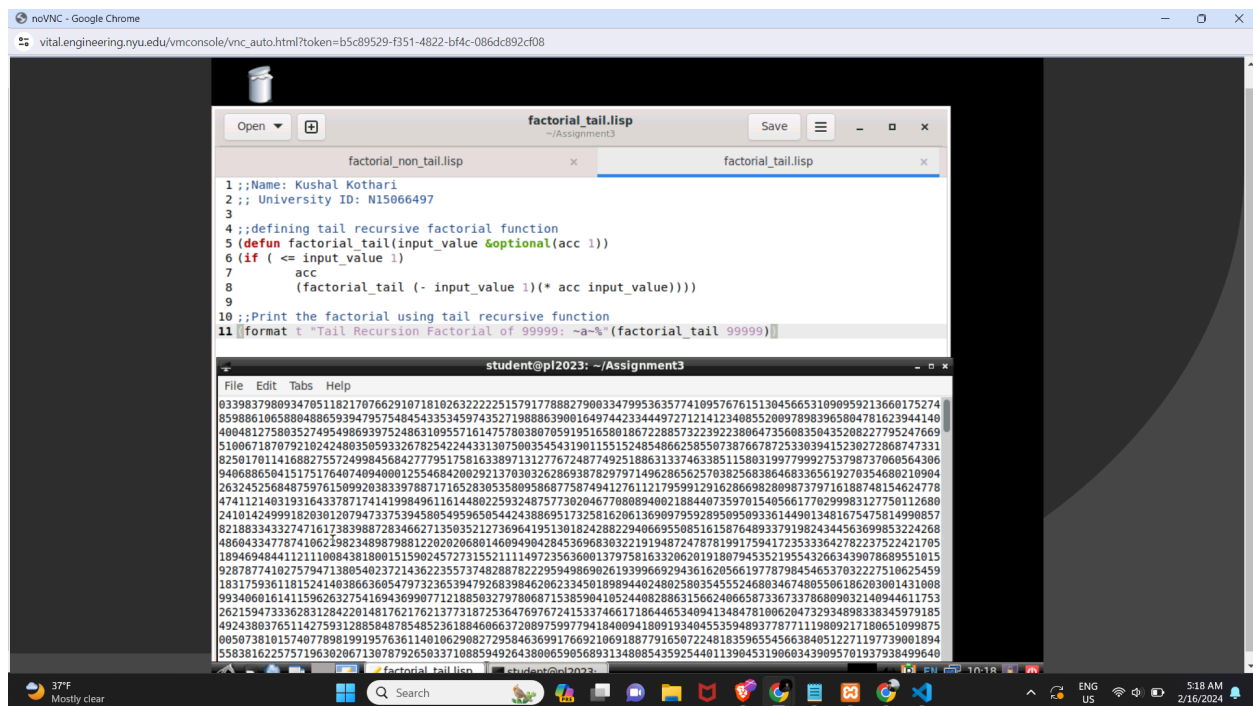
```
  (if (<= input_value 1)
```

```
      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))
```



The screenshot shows a web browser window with a URL from vital.engineering.nyu.edu. It displays a Common Lisp script in a text editor and its output in a terminal window. The script defines a tail-recursive factorial function and prints the result for 99,999. The terminal output shows a long string of digits representing the factorial of 99,999.

```
factorial_tail.lisp
1 ;; Name: Kushal Kothari
2 ;; University ID: N15066497
3
4 ;; defining tail recursive factorial function
5 (defun factorial_tail(input_value &optional (acc 1))
6   (if (<= input_value 1)
7       acc
8       (factorial_tail (- input_value 1) (* acc input_value))))
9
10 ;; Print the factorial using tail recursive function
11 (format t "Tail Recursion Factorial of 99999: ~a~%" (factorial_tail 99999))
```

```
student@pl2023: ~/Assignment3
83398379809347051182170766291071810263222251579177888279003347995363577410957676151304566531090959213660175274
8598061065808408659394795754845433545974352719880639001649744233444972712141234805200978983965004781623944140
400481275803527495498693975248631095571614757803807059195165801867228857322392238064735608350435208227795247669
51006718707921024248035059332678254224331307500354543190115515248548662585507387667872533039415230272868747331
82501701141688275572499845684277951758163389713127767248774925188631337463385115803199779992753798737060564306
94068865041517517640740940001255468420029213703032628693878297971496286562570382568386468336519270354680210904
263245256848759761509920383397887171652830535809586877587494127611217959912916286698280987379716188748154624778
474112140319316433787174141998496116144802259324875773020467708089400218844073597015405661770299983127750112680
241014249991820301207947337394580549596505442438069517325816206136909795928950953361449013481675475014990857
82188343327471617583988728346627135035212736964195130182428822940669550851615876489337919824344563699853224268
4860434774781062198234898798812202020680146094904284536968303221919487247878199175941723533642782237522421705
1894694844121110084381800151590245727315521114972356360013797581633206201918079453521955432663439078689551015
928787741027579471380540237214362235573748288782229594906902619399669294361620656197787984546537032227510625459
18317593611815241403866360547973236539479268398462062334501898944024802580354552468034674805506186203001431008
993406016141159626327541694369907712188503279780687153859041052440828863156624066587336733786809032140944611753
262159473336283128422014817621762137731872536476976724153374661718644053409413484781006204732934898338345979185
49243083765114275931208584878548523618846066372089759977941840094180913404553594893778771119809217180651099875
005073810517407789819919576361140106290827295846369917669210691887791650722481835965545663840512271197739001894
558381622575719630206713078792650337108859492643800659056893134808543592544011390453190603439095701937938499640
```

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:

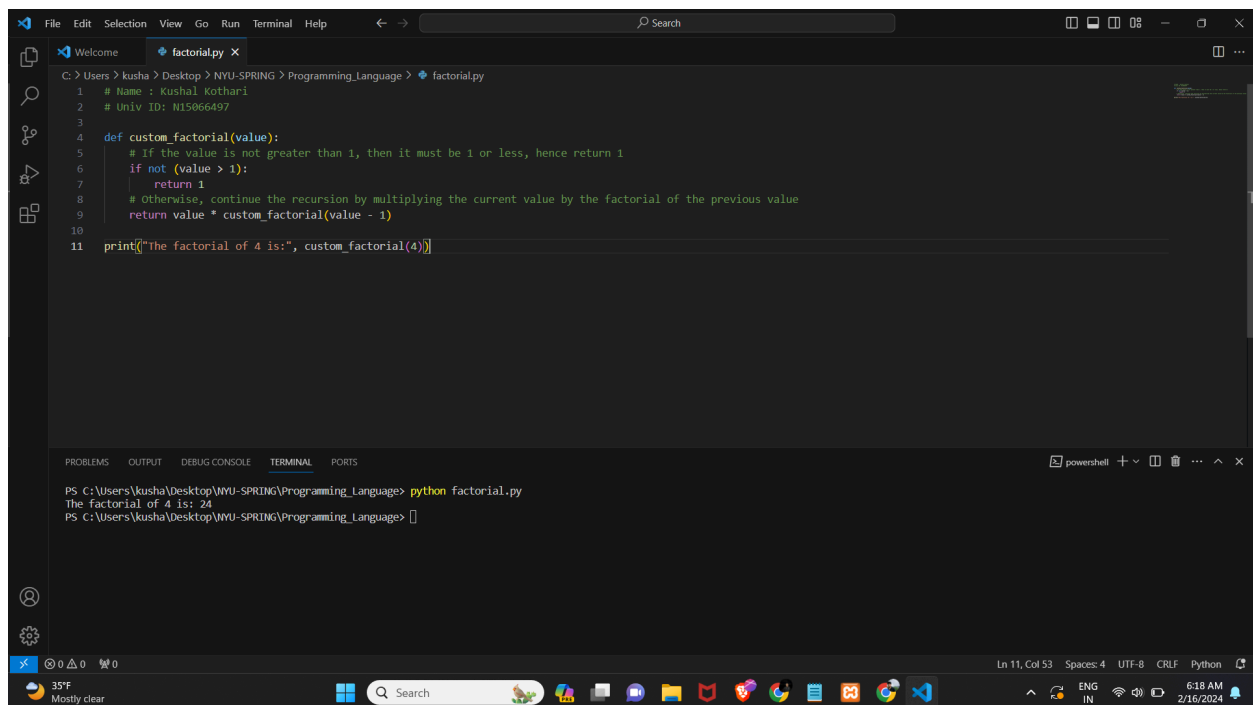
Code: small input

```
# 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

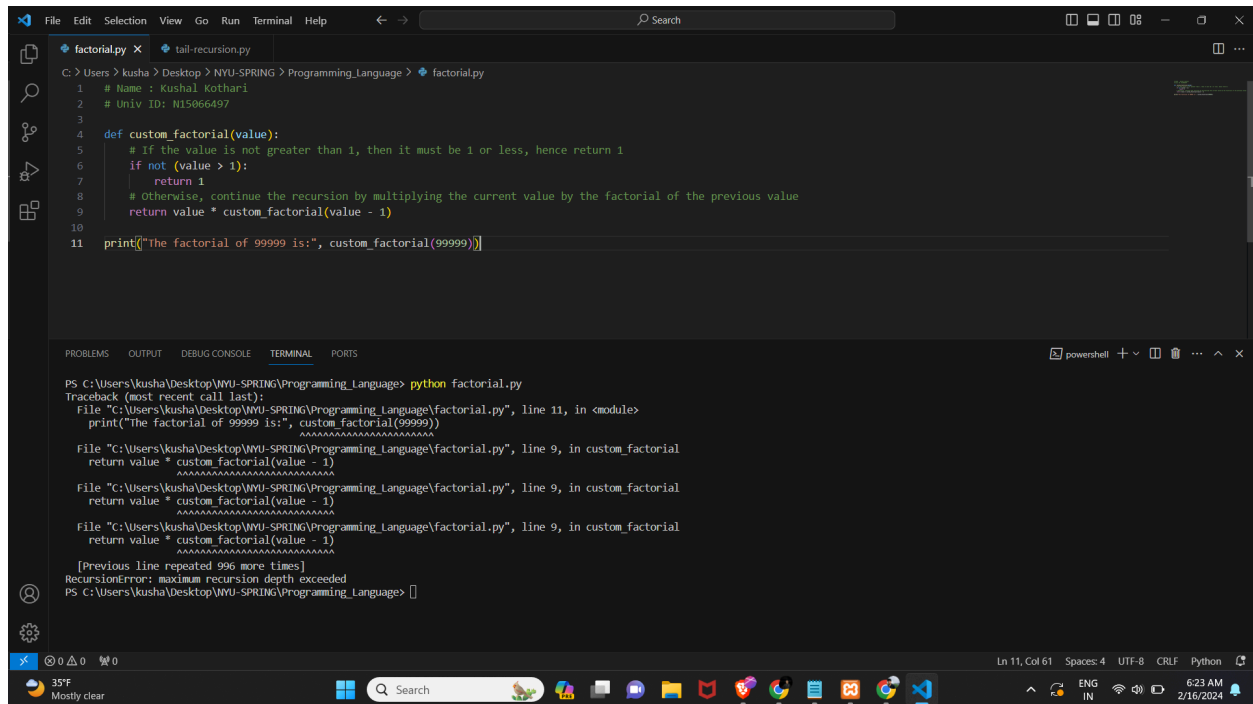
CODE:

```
# 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:



```
factorial.py x  tail-recursion.py
C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language> python factorial.py
1 # Name : Kushal Kothari
2 # Univ ID: N15066497
3
4 def custom_factorial(value):
5     # If the value is not greater than 1, then it must be 1 or less, hence return 1
6     if not (value > 1):
7         return 1
8     # Otherwise, continue the recursion by multiplying the current value by the factorial of the previous value
9     return value * custom_factorial(value - 1)
10
11 print("The factorial of 99999 is:", custom_factorial(99999))

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language> python factorial.py
Traceback (most recent call last):
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\factorial.py", line 11, in <module>
    print("The factorial of 99999 is:", custom_factorial(99999))
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\factorial.py", line 9, in custom_factorial
    return value * custom_factorial(value - 1)
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\factorial.py", line 9, in custom_factorial
    return value * custom_factorial(value - 1)
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\factorial.py", line 9, in custom_factorial
    return value * custom_factorial(value - 1)
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
PS C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language>
```

TAIL RECURSION:

Code:

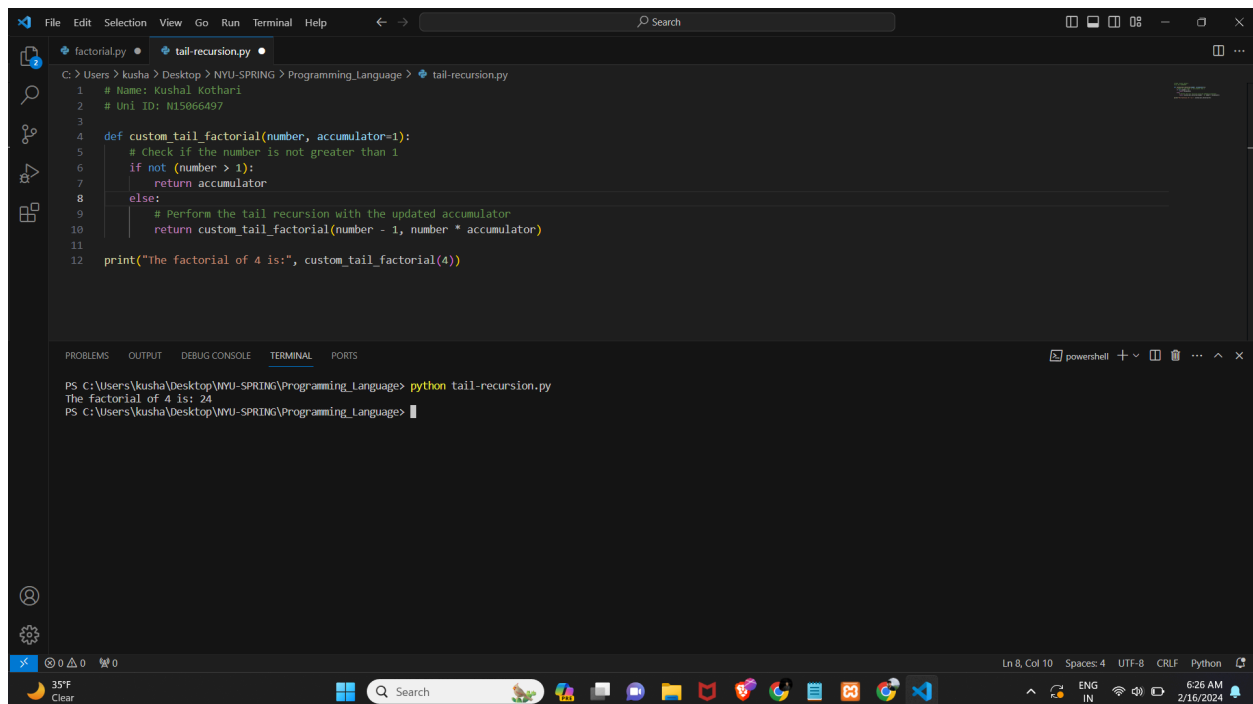
With small value as input

```
# 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:



The screenshot shows a code editor with a file named 'tail-recursion.py' open. The code is a Python function 'custom_tail_factorial' that calculates the factorial of a number using tail recursion. It includes comments and a print statement to display the result for the input 4. Below the code editor, a terminal window shows the command 'python tail-recursion.py' being executed, resulting in the output 'The factorial of 4 is: 24'.

```
File Edit Selection View Go Run Terminal Help
tail-recursion.py
C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language> tail-recursion.py
1 # Name: Kushal Kothari
2 # Uni ID: N15066497
3
4 def custom_tail_factorial(number, accumulator=1):
5     # Check if the number is not greater than 1
6     if not (number > 1):
7         return accumulator
8     else:
9         # Perform the tail recursion with the updated accumulator
10        return custom_tail_factorial(number - 1, number * accumulator)
11
12 print("The factorial of 4 is:", custom_tail_factorial(4))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language> python tail-recursion.py
The factorial of 4 is: 24
PS C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language>
```


With large value as Input

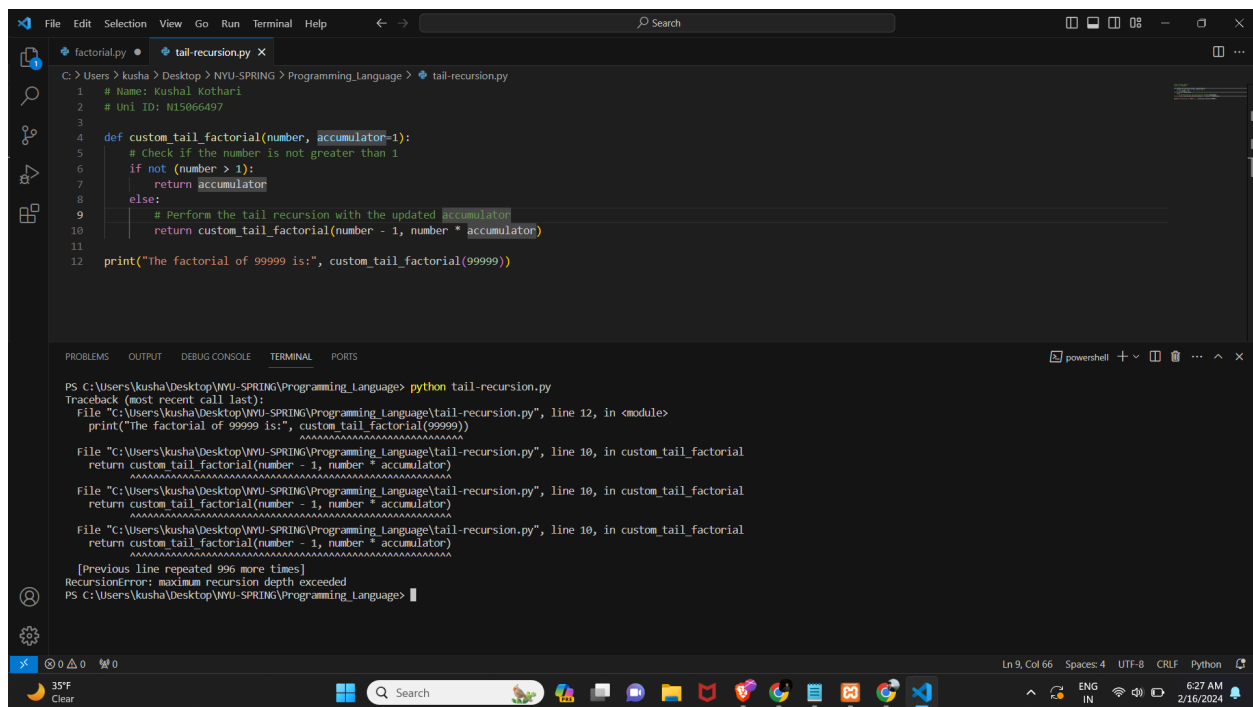
Code:

```
# 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:



```
File C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\python tail-recursion.py
Traceback (most recent call last):
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\tail-recursion.py", line 12, in <module>
    print("The factorial of 99999 is:", custom_tail_factorial(99999))
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\tail-recursion.py", line 10, in custom_tail_factorial
    return custom_tail_factorial(number - 1, number * accumulator)
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\tail-recursion.py", line 10, in custom_tail_factorial
    return custom_tail_factorial(number - 1, number * accumulator)
  File "C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language\tail-recursion.py", line 10, in custom_tail_factorial
    return custom_tail_factorial(number - 1, number * accumulator)
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
PS C:\Users\kusha\Desktop\WVU-SPRING\Programming_Language>
```

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.