**Isaac Computer Science Student Activity Booklet** 

Recursive Programming Solutions

**Activity 1:** Visualise **Python Tutor:** <http://tiny.cc/37czyz>

* **Run** this code in the Python Tutor visualiser.

def sum\_to\_n(n):

if n == 1:

return 1

else:

return n + sum\_to\_n(n-1)

sum\_to\_n(3)

* Step through the code by pressing the *Next* button
* Click *Edit this code* to change the code
* Use the *Generate permanent link* button to create a link for your own code.
* What does this program do?  
  This program calculates the sum of all integers from 1 to n using recursion. The function sum\_to\_n(n) calls itself with n-1 until it reaches the base case of n == 1, at which point it returns 1. The recursive calls then accumulate the sum as they return back up the call stack. For example, calling sum\_to\_n(3) returns 3 + 2 + 1, which equals 6.
* How can visualising code execution help in debugging?  
  Visualising code execution helps in debugging by allowing you to see the step-by-step flow of the program. This includes tracking the values of variables, understanding the control flow, and seeing how recursive calls unfold and return. It makes it easier to identify logical errors, understand how data is being manipulated, and pinpoint where the program deviates from the expected behaviour. Tools like Python Tutor provide a clear, visual representation of the code’s execution, which can be especially helpful for complex recursive functions.

**Activity 2:** Stack Overflow **Trinket:** <https://trinket.io/python3/f5edfef789>

* Create a recursive function that takes an integer argument, prints it out, increments it by 1 and then calls itself, passing in the new value.

def recursive\_function(n):

n += 1

print(n)

recursive\_function(n) # general case

* How many calls to itself does it manage before we get a ‘stack overflow’?
* To overcome this issue, we need to introduce what is known as a ‘base case’. Use the recursive structure to write a base case for your function.
* **Hint:** The known condition should be one less than the recursive depth allowed by your machine.

**if** (condition for which answer is known):

statement # base case

**else:** recursive function call # general case

def recursive\_function(n):

if n >= 993: # base case

print(n)

else:

print(n)

n += 1

recursive\_function(n) # general case

**Activity 3:** Factorial **Trinket:** <https://trinket.io/python3/229debde2f>

* Given the following recursive function to calculate the factorial of a number:

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5))

* Trace the execution of the factorial function call factorial(5) step-by-step and fill in the table below to show the values of x, whether the base case condition is met (if n = 0) and the value returned at each level of recursion.

|  |  |  |  |
| --- | --- | --- | --- |
| Level | n | if n = 0: | Return |
| 0 | 5 | F | 120 (5 \* 24) |
| 1 | 4 | F | 24 (4 \* 6) |
| 2 | 3 | F | 6 (3 \* 2) |
| 3 | 2 | F | 2 (2 \*1) |
| 4 | 1 | F | 1 (1 \* 1) |
| 5 | 0 | T | 1 |

* Start at Level 0 with n = 5 and trace the recursive calls until the base case is reached.
* For each level, note whether the condition if X = 0 is true (T) or false (F).
* Calculate the value returned at each level after the base case is reached and the recursion starts to unwind.
* Fill in the final returned value at each level in the “Return” column.

**Activity 4:** Tail Recursion **Trinket:** <https://trinket.io/python/dec0c23d73>

* Given the following recursive function to calculate the factorial of a number using tail recursion:

def factorial(n, a=1):

if n == 0:

return a

else:

return factorial(n - 1, n \* a)

print(factorial(5))

* Trace the execution of the tail recursive function call factorial(5) step-by-step and fill in the table below to show the values of n, whether the base case condition is met (if n == 0) and the value returned at each level of recursion.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Level | n | a | if n = 0: | Return |
| 0 | 5 | 1 | F |  |
| 1 | 4 | 5 ( 5 \* 1) | F |  |
| 2 | 3 | 20 (4 \* 5) | F |  |
| 3 | 2 | 60 (3 \* 20) | F |  |
| 4 | 1 | 120 ( 2 \* 60) | F |  |
| 5 | 0 | 120 (1 \* 120) | T | 120 |

* Start at Level 0 with n = 5 and a = 1 and trace the recursive calls until the base case is reached.
* For each level, note whether the condition if n == 0 is true (T) or false (F).
* Calculate the value returned at each level after the base case is reached and the recursion starts to unwind.
* Fill in the final returned value at each level in the “Return” column.

**Activity 5:** Recursion vs Iteration

**Trinket:** <https://trinket.io/python3/f4a209d80c>

* The algorithm represented using the recursive function below calculates the factorial of a number. Your task is to develop an equivalent iterative function (naive\_factorial) to achieve the same result.

def recursive\_factorial(n):

if n == 0: # base case

return 1

else: # general case

return n \* recursive\_factorial(n - 1)

What you need to do:

* **Task 1:** Write the Python program for the naive\_factorial function described above using the pseudocode below:

FUNCTION naive\_factorial(n):

result ← 1

WHILE n > 0 DO:

result ← result \* n

n ← n - 1

END WHILE

RETURN result

END FUNCTION

* **Task 2**: Test the program by showing the result of entering 5 and 7. Use the following code to compare both functions:

# Testing recursive factorial

print("Recursive Factorial of 5:", recursive\_factorial(5))

print("Recursive Factorial of 7:", recursive\_factorial(7))

# Testing naive factorial

print("Naive Factorial of 5:", naive\_factorial(5))

print("Naive Factorial of 7:", naive\_factorial(7))

* **Task 3:** Compare the result of the naive\_factorial function with the recursive\_factorial function for the inputs 5 and 7. Ensure both functions produce the same output.

|  |
| --- |
| def naive\_factorial(n):  result = 1  while n > 0:  result \*= n  n -= 1  return result |

**Activity 6:** Which is best?

**Trinket:** <https://trinket.io/python3/9aa4a3cc29>

* Compare the time efficiency of the naive/iterative and the elegant/recursive factorial algorithm.
* **Hint:** Use the following test to measure the efficiency of each algorithm.

import time

start = time.time()

for count in range(10000):

x = 1 + 1 # The code you want to test

end = time.time()

print(end - start)

* Which algorithm was most time efficient?

import time

start = time.time()

for count in range (10000):

naive\_factorial(4)

end = time.time()

print(end - start)

start = time.time()

for count in range (10000):

recursive\_factorial(4)

end = time.time()

print(end - start)