Recursion

Definition of a recursive subroutine

A function is recursive if it is defined in terms of itself **and** it calls itself.

A recursive function has **three** essential characteristics:

- A stopping condition or **base case** which when met means that the function will not call itself and will start to 'unwind'.
- A condition that deals with input values other than the base case condition, this is where the function calls itself. This is the **recursive case**.
- The stopping condition must be reached after a finite number of calls, i.e the function can't endlessly call itself. Python has a limit of 1000 calls.

Algorithms can be written using either recursion or iteration. Recursive routines are often much shorter, but **more difficult to trace through**. If a recursive function is called a very large number of times before the stopping condition is reached, the program may crash with a "Stack overflow' error. An iterative routine, on the other hand, has no limit to the number of times it may be called.

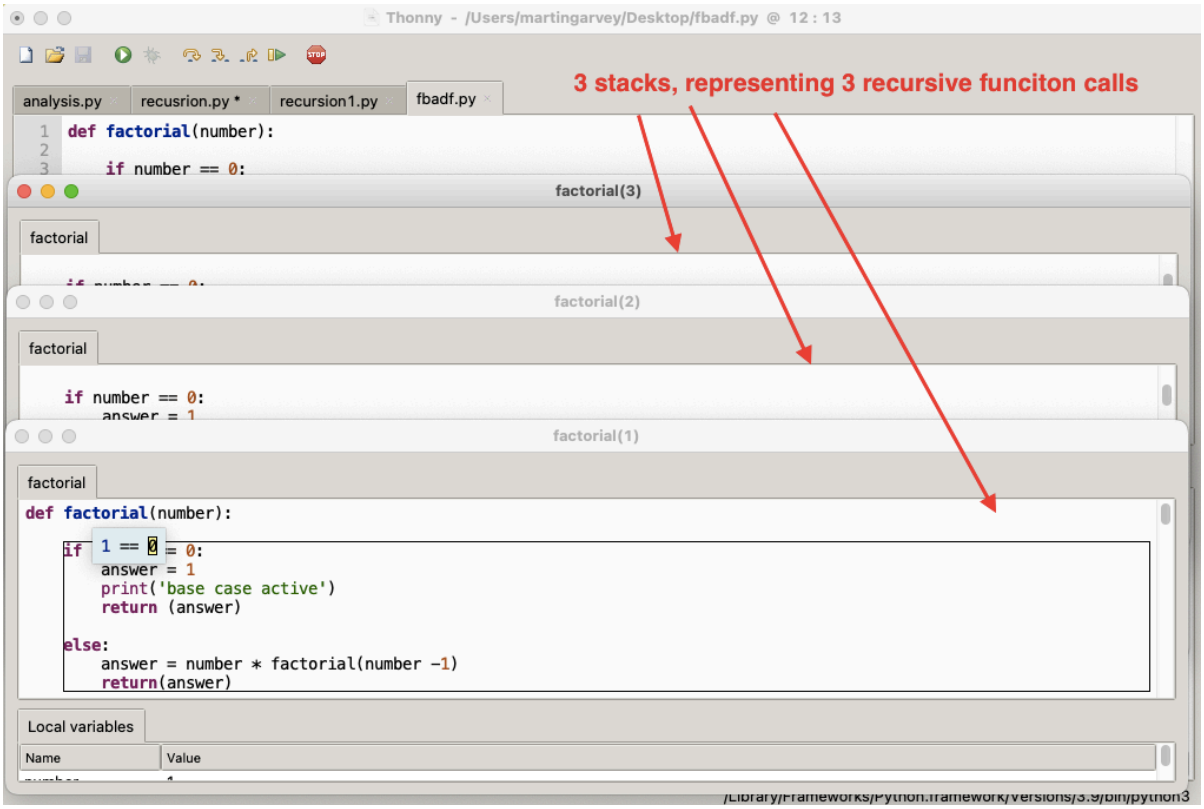
Example

A simple example of a recursive routine is the calculation of a factorial, n!. A function to calculate a factorial for any positive whole number n! is recursive. The definition for the function uses:

- a base case of 0! = 1
- a general case of n! = n * (n–1)!

```
1  def factorial(number):
2      # Base Case
3      if number == 0:
4          answer = 1
5          print('Here in the base case.')
6          return answer
7
8     # Recursive Case
9      else:
10         answer = number* factorial(number - 1)
11         print(f'Value returned from previous call stack {answer}')
12         return answer
13
14
15 factorial(3)
```

Nothing will be printed until the routine has stopped calling Itself. As soon as the base case is reached, in this case n = 0, the variable answer is set equal to 1, the return statement at this point is passed back to the previous call to *factorial.*



Each function call open until it unwinds. (Stacks)

With recursive functions, the statements after the recursive function call are not executed until the base case is reached; this is called winding. After the base case is reached it can be used to end the recursive process, the function now begins unwinding. By unwinding I mean, each return call is returning data back to its original call. In the image above you can see each windows is called factorial(3), facotrial(2), factorial(1) etc, these are all the active recursive calls. So factorial(1) is going to return the variable back to the its call in factorial(2) etc.

In order to understand how the winding and unwinding processes in recursion work, we can use a trace table for a specific example: 3!

Call number	Function call	number	answer	RETURN	
1	Factorial (3)	3	3 * factorial (2)		winding
2	Factorial (2)	2	2 * factorial (1)		winding
3	Factorial (1)	1	1 * factorial (0)		winding
4	Factorial (0)	0	1	1	base case
3 continued	Factorial (1)	1	1 * 1	1	unwinding
2 continued	Factorial (2)	2	2 * 1	2	unwinding
1 continued	Factorial (3)	3	3 * 2	6	unwinding

Recursive algorithms

Binary search algorithm

```
#defining a function to execute binary search on any given sorted list L.
#start is the lowest index of the list being checked at any given time.
#end is the highest index of the list being checked at any given time.
#item is the item to be searched in the list.

def binary_search(L, start, end, item):
    if end >= start:
        middle = (start + end) // 2

        if L[middle] == item:  # <---- Base Case
            return middle #middle element is the item to be located
        #if middle item is greater than the item to be searched, left side of the list will be searched

        elif L[middle] > item:      # <---- Recursive Case
            #starting index will be same but ending index will be the middle of the main list
            # i.e. left half of the list is given in function.
            return binary_search(L, start, middle - 1, item)

        else:  #<---- Also a recursive case
            #if middle item is smaller than the item to be searched, new starting index
            # will be middle of the list i.e. right half of the list.
            return binary_search(L, middle + 1, end, item)
    else:
        #if element is not present in the list
        return -1

#Drivers code
my_list = [ 2, 4, 6, 9, 12, 16, 18, 19, 20, 21, 22 ]
element_to_search = 6

index_of_element = binary_search(my_list, 0, len(my_list)-1, element_to_search)  #<-- original call

if index_of_element != -1:
    print("Element searched is found at the index ", str(index_of_element), "of given list")
else:
    print("Element searched is not found in the given list!")
```

Recursion occurs each time the midpoint value (L[middle]) is not equal to '*item'*. The base case is activated when they do equal, otherwise the end and start variables will eventually become equal which will return(-1) indicating that the element is not in the list. Try working it out with a pen and paper.

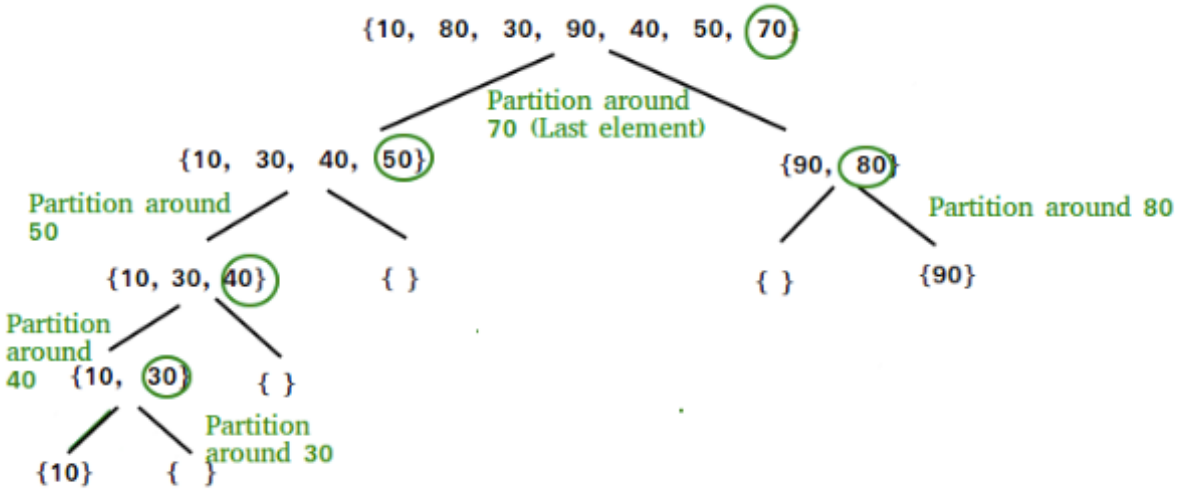
Quicksort sorting algorithm

```
1  def quick_sort(L):
2
3      left_list = []
4      middle_list = []
5      right_list = []
6
7     # Base case
8      if len(L) <=1:
9          #print(f'Base case {L}')
10         return(L)
11
12 # Set pivot to the last element in the list
13     pivot = L[-1]
14
15 # Iterate through all elements (keys) in L
16     for key in L:
17         if key < pivot:
18             left_list.append(key)
19         elif key == pivot:
20             middle_list.append(key)
21         else:
22             right_list.append(key)
23     print(left_list)
24     print(middle_list)
25     print(right_list)
26     input()
27     # Repeat the quicksort on the sub-lists and combine the results return quick_sort(left_list) + middle_list + quick_sort(right_list)
28     return quick_sort(left_list) + middle_list + quick_sort(right_list)
29
30 L = [88, 46, 25, 11, 18, 12, 22]
31 print("INPUT (initial list): ", L)
32 print("OUTPUT (sorted list): ", quick_sort(L))
```

The quick sort algorithm shown above is recursive, it partitions the given list around a picked pivot value.

- Pick a “pivot” item. (Line 13)
- Partition the other items by adding them to a “less than pivot” sublist, or “greater than pivot” sublist. (Left_list, right_list), (Line 18 and 22)
- The pivot goes between the two lists (middle_list, line 20)
- Repeat the quicksort on the sublists, until you get to a sublist of size 1 (which is sorted). (Line 28)
- Combine the lists — the entire list will be sorted

The pivot point in a quick sort algorithm is the point around which comparisons are made, similar to the midpoint value in a binary search. However, the pivot point can be any value you want it to be. In the example above the pivot point is set as the last item in the list, line 13. Each time recursion occurs, the pivot point will be the end item in the list being sorted.



Algorithm Complexity for Quicksort

Selection of the pivot value is the main factor affecting Quicksort algorithms. For example n the worst case scenario, if you pick the first/last element as the pivot point and it is already the smallest/largest item in the list then your ‘smaller/larger than the pivot value’ list will still contain all the other elements in the list rather than being split. That means we are only creating a subset of one item smaller each time, which gives us O(N^2) behavior in the worst case.