Chapter 10: Recursion

Contents

Syllabus Learning Outcomes

2.2 Programming Elements and Constructs

Use programming language elements and constructs to write recursive and non-recursive programs to solve a variety of problems.

- 2.2.5 Understand the concept of recursion.
- 2.2.6 Trace the steps and list the results of recursive and non-recursive programs.
- 2.2.7 Understand the use of stacks in recursive programming.

1 Trace tables

Using trace tables is a technique used to test an algorithm and predict step by step how the computer will run the algorithm.

Trace tables are tables that consist of columns. Each column can represent a variable, a condition, or an output. Not every variable, condition or output in an algorithm needs a column in a trace table.

The purpose of the table is that we can run through an algorithm and simulate what a computer would do if the program were to execute. We complete the table to show how the variables change, what the conditions would resolve to, and/or what outputs would be displayed.

There are two main reasons that trace tables are used. The first is to determine what an algorithm does by running through it to see what happens as the algorithm runs. The second is to test the logic of an algorithm if there are errors that are not easily spotted.

1.1 Examples

A trace table can be used to track the values of variables as they change from line to line while the program is running.

Algorithm

```
1  number = 3
2  PRINT number
3  FOR i from 1 to 3:
4      number = number + 5
5      PRINT number
6  PRINT " ? "
```

Trace Table

Line	number	i	OUTPUT
1	3		
2			3
3		1	
4	8		
5			8
3		2	
4	13		
5			13
3		3	
4	18		
5			18
6			?

"Line" may not always be a column in a trace table. The above trace table can be drawn with columns consisting only the variables and output.

number	i	OUTPUT
3		3
8	1	8
13	2	13
18	3	18
		?

The following examples show some basic algorithms and their trace tables.

Example 1

The purpose of the algorithm in this example is to swap the values of x and y.

Algorithm

```
1 x = 3
2 y = 4
3
4 temp = x
5 x = y
6 y = temp
```

Trace table

x	y	temp
3	4	3
4	3	

Example 2

This algorithm includes a conditional statement - IF/ELSE.

Algorithm

```
1   code = 61
2   IF code MOD 2 == 0 THEN
      code = code * 3
3   ELSE
      code = code * 2
4   ENDIF
5   OUTPUT code
```

Trace table

code	code MOD 2 == 0	OUTPUT
61	False	
122		122

Example 3

The algorithm below contains 2 variables (`num`, `count`), 1 condition (`num < 500`) and 1 output (`OUTPUT num`). The variable `count` will keep track of how many times the while loop has repeated. The trace table shows how this algorithm would run if the user entered an input of 16.

Algorithm

```
1 num ← USERINPUT
2 count ← 0
3
4 WHILE num < 500 THEN
5     num ← num * 2
6     count ← count + 1
7 ENDWHILE
8
9 OUTPUT num
10 OUTPUT count
```

Trace table

num	count	num < 500	OUTPUT
16	0	True	
32	1	True	
64	2	True	
128	3	True	
256	4	True	
512	5	False	
			512
			5

1.2 Exercises

Complete the trace tables for the following algorithms.

Algorithm 1

```
1 code = ""
2 code = code + "7"
3 code = "4" + code
4 code = code + "3"
5 OUTPUT(code)
```

Trace table

num	OUTPUT

Algorithm 2

```
number = 5
factorial = number

WHILE number>2
    number = number - 1
    factorial = factorial * number
END WHILE

OUTPUT(factorial)
```

Trace table

number	factorial	number > 2	OUTPUT

Algorithm 3

```
code = ""

FOR i FROM 1 TO 3
   digit = i * 2
   code = code + STR(digit)
ENDFOR

OUTPUT(code)
```

Trace table

i	digit	code	OUTPUT

2 Recursion

The process in which a function calls itself directly or indirectly is known as recursion. The corresponding function is called a recursive function.

Generally, a recursive function has these features:

1. It is defined in terms of itself.
2. It calls itself with one or more similar but smaller problems.
3. It can repeat itself until a base case or terminating case is reached.
   A base case or terminating condition is a condition that determines when the recursion should stop. It is the smaller problem that is simple enough to be solved easily and does not require further recursion.

Any iteration can possibly be rewritten into a recursive function and vice versa.

2.1 Recursion without return value

Let's look at a countdown() function using while loop.

It takes in a parameter n, which is the starting number to count down to 1.

```
1  def countdown(n):
2      while n > 0:
3          print(n)
4          n = n - 1
5      print('Done!')
6
7  # main
8  countdown(3)
```

```
3
2
1
Done!
```

Can this function be converted into a recursive function? YES!

How do we convert the countdown function into a recursive function?

Analysis steps:

- What are the common actions in each loop? (Hint: check the loop statements)
- What is the base case (or terminating case)?
- What does it do when it is the base/terminating case?

Analysis result:

- In each iteration, it prints out current `n` value.
- The base case is when `n = 0`.
- If base case is `True`, it prints `Done`.

Thus,

- In the recursive function, it checks for the base case.
- Once the base case is true, it prints Done, else it prints current `n` value and recurses (calls the same function again)

Let's convert the countdown() function into a recursive function countdown_r().

```
1  def countdown_r(n):
2      if n == 0:
3          print("Done")
4      else:
5          print(n)
6          countdown_r(n-1)
7
8  # main
9  countdown_r(3)
```

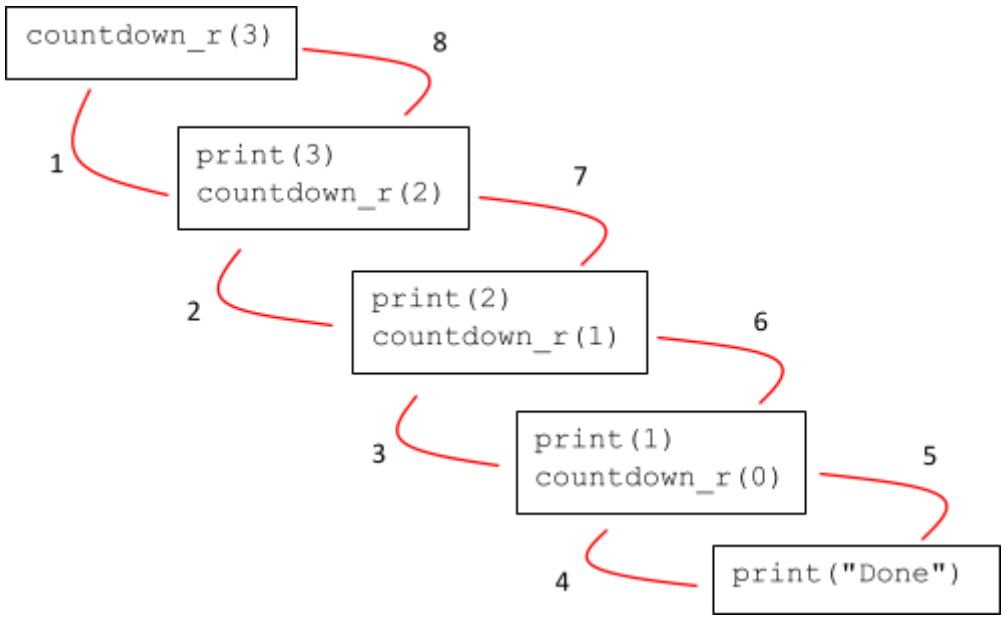
```
3
2
1
Done
```

How do you visualise the above recursive function execution?

1. Trace table

Call number	Procedure call	n	n == 0	OUTPUT
1	countdown_r(3)	3	False	3
2	countdown_r(2)	2	False	2
3	countdown_r(1)	1	False	1
4	countdown_r(0)	0	True	"Done"

2. Trace tree



Let's modify the countdown() function into a recursive function count up function countup_r().

```
1  def countup_r(n):
2      if n == 0:
3          print("Done")
4      else:
5          countup_r(n-1)
6          print(n)
7
8
9  # main
10 countup_r(3)
```

```
Done
1
2
3
```

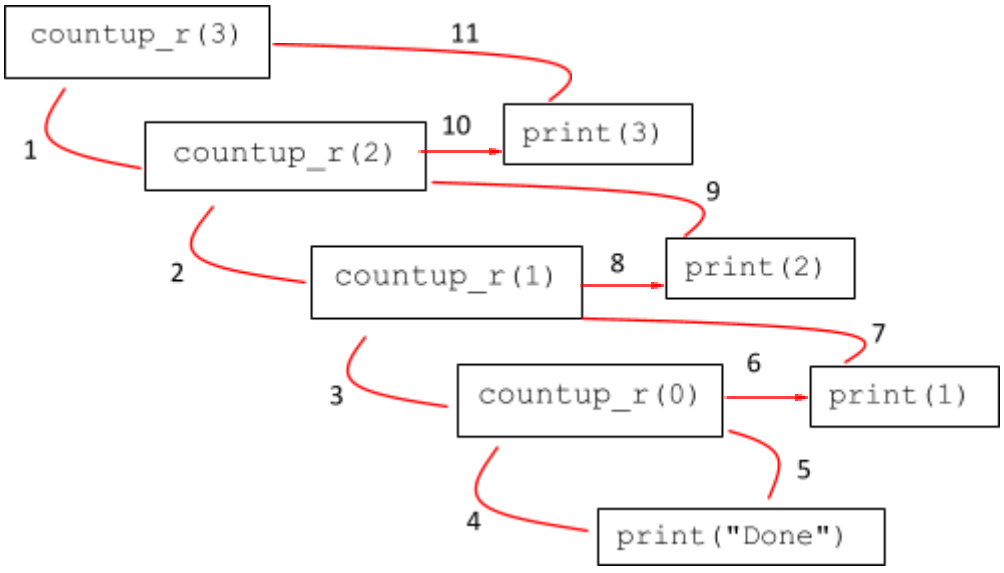
By switching the code on line 5 and 6, we can totally change a count down to a count up as the order of execution has been switched. Instead of printing `n` followed by calling the function, the program recursively calls the function until it reaches the base case then it starts the printing.

Let's visualise the execution order of the `countup_r` function.

1. **Trace table**

Call number	Procedure call	n	n == 0	OUTPUT
1	countup_r(3)	3	False	
2	countup_r(2)	2	False	
3	countup_r(1)	1	False	
4	countup_r(0)	0	True	"Done"
3	countup_r(1)	1	False	1
2	countup_r(2)	2	False	2
1	countup_r(3)	3	False	3

2. **Trace tree**



2.2 Recursion with return value

In the above count-down and count-up examples, the recursion function does not return any value. It is common for recursion functions to return value back to the calling function.

Let's look at a recursion function with return value.

Factorial

The factorial value of an explicit number n is typically represented as `n!`, and $F_n = n * F_{n-1}$.

```
n! = n * (n-1) * (n-2) * . . . * 1

n! = n * (n-1)!
```

We can implement an iterative function `factorial_i(n)` to calculate the factorial value of an integer `n`.

```
def factorial_i(n):
    factorial = 1
    while n > 0:
        factorial *= n
        n -= 1

    return factorial

print(factorial_i(4))
```

```
24
```

How do we convert the iterative factorial function into a recursive function?

Analysis steps:

- What are the common actions in each iteration? (Hint: check the loop statements)
- What is the base case (or terminating case)?
- What does it do when it is the base/terminating case?

Analysis result:

- It sets `factorial = factorial * n` and repeats with `n-1`
- The base case is when `n = 1`
- If base case is `True`, program will execute while loop a last time and exit the loop. After which, it returns `factorial`.

Let's now implement the `factorial_r(n)` recursive function to compute `factorial(4)`.

```
def factorial_r(n):
    if n == 1:
        return 1
    else:
        return n * factorial_r(n-1)

print(factorial_r(4))
```

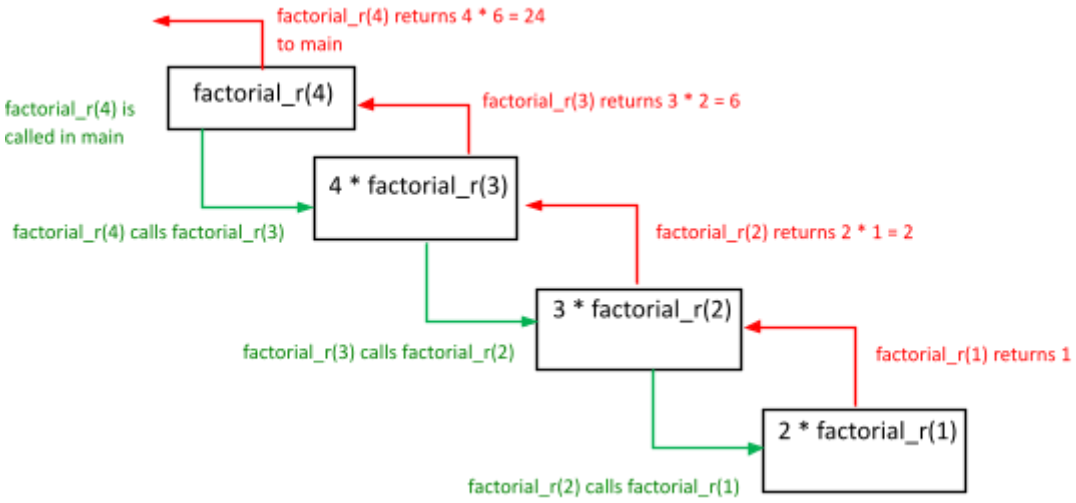
```
24
```

Let's visualise the execution order of the `factorial_r(n)` recursive function using a trace table and a trace tree.

1. **Trace table**

Call number	Procedure call	n	n == 1	Return value
1	factorial_r(4)	4	False	
2	factorial_r(3)	3	False	
3	factorial_r(2)	2	False	
4	factorial_r(1)	1	True	1
3	factorial_r(2)	2	False	2
2	factorial_r(3)	3	False	6
1	factorial_r(4)	4	False	24

2. **Trace tree**



2.3 Using stacks to store return addresses

When a function is called, a certain amount of memory is set aside for that function to use for purposes such as storing local variables. This memory, called a **stack frame**, is also used by the computer to store information about the function such as the **function's address in memory** as well as **the contents of the local variables.** This allows the program to return to the proper place after a function call. For example, if you write a function that calls printf(), you would like control to return to your function after printf() completes. This is made possible by the frame.

When a function calls another function, the caller's stack space is kept intact, and a new space (stack frame) is created to handle the new function call.

When a function finishes its work and returns to its caller, its associated space is released.

Illustration of stack using factorial_r(4):

```
1  def factorial_r(n):
2      if n == 1:
3          return 1
4      else:
5          return n * factorial_r(n-1)
6
7  print(factorial_r(4))
```

```
24
```

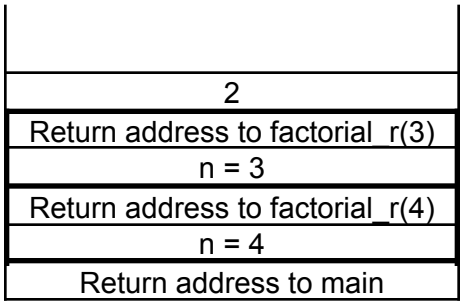
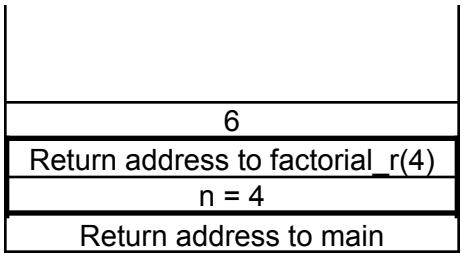
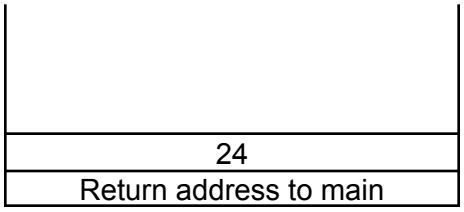
What happens when the program above is executed?

The first program statement to be executed is line 7. The actual parameter n takes in the value of 4. The function call causes the return address to be put on the stack. Program execution jumps to line 1 to execute the function `factorial_r(4)`.

The execution of `factorial_r(4)` begins. Since it is not base case, line 5 will be executed, `n` is decremented by 1 and becomes 3. The function call, `factorial_r(3)`, causes the return address of `factorial_r(4)` at line 5 to be stored on the stack, together with the current contents of the local variables. The locations used to store these values are referred to as a **stack frame**. Each recursive call will add another stack frame to the stack until the base case is reached.

When the base case `factorial_r(1)` is reached, 1 is returned by pushing it onto the stack. This value is popped off the stack and returned to the caller of this function. With each return from a function call, the corresponding stack frame is taken off and the values of the local variables are restored. Eventually, control is returned to main at line 7.

Description	Stack
1st call factorial_r(4) is made by the main program. Return address to main is pushed into stack.	Return address to main
2nd call factorial_r(3) is made by factorial_r(4). Return address to factorial_r(4) and content of n is pushed into stack.	Return address to factorial_r(4) n = 4 Return address to main
3rd call factorial_r(2) is made by factorial_r(3). Return address to factorial_r(3) and content of n is pushed into stack.	Return address to factorial_r(3) n = 3 Return address to factorial_r(4) n = 4 Return address to main
4th call factorial_r(1) is made by factorial_r(2). Return address to factorial_r(2) and content of n is pushed into stack.	Return address to factorial_r(2) n = 2 Return address to factorial_r(3) n = 3 Return address to factorial_r(4) n = 4 Return address to main
Base case reached. Push 1 onto stack.	1 Return address to factorial_r(2) n = 2 Return address to factorial_r(3) n = 3 Return address to factorial_r(4) n = 4 Return address to main

Pop 1 and stack frame. Return to factorial_r(2). Push new value of 1*2 onto stack.	
Pop 2 and stack frame. Return to factorial_r(3). Push new value of 2*3 onto stack.	
Pop 6 and stack frame. Return to factorial_r(4). Push new value of 6*4 onto stack. Return to main	

2.3 Benefits and drawbacks of recursion

Recursion solutions are often more elegant and use less program code than iterative solutions. A complex task can be broken down into simpler sub-problems using recursion where iterative solutions may be difficult to program. When designing a solution to a mathematical problem that is recursive by nature, recursive solution is easier to implement than an iterative one.

However, repeated recursive calls can carry large amounts of memory usage and processor time from the multiple function calls, each time storing of return addresses and copies of local or temporary variables. If the recursion continues too long, computer may run out of memory and the program will crash.