National University of Sciences & Technology (NUST)
School of Natural Sciences (SNS)
Department of Mathematics

CS250: Data Structures and Algorithm

Class: BS2k21_Mathematics – Fall 2023

Lab 09:

Recursion

Date: 14th November, 2023

Time: 10:00 am - 01:00 pm

Instructor: Fauzia Ehsan

Submission Deadline: 21st November, 2023 (11:59 P.M)

National University of Sciences & Technology (NUST)
School of Natural Sciences (SNS)
Department of Mathematics

Lab 09: Recursion

"A mirror mirroring a mirror"

– **Douglas R. Hofstadter, I Am a Strange Loop**

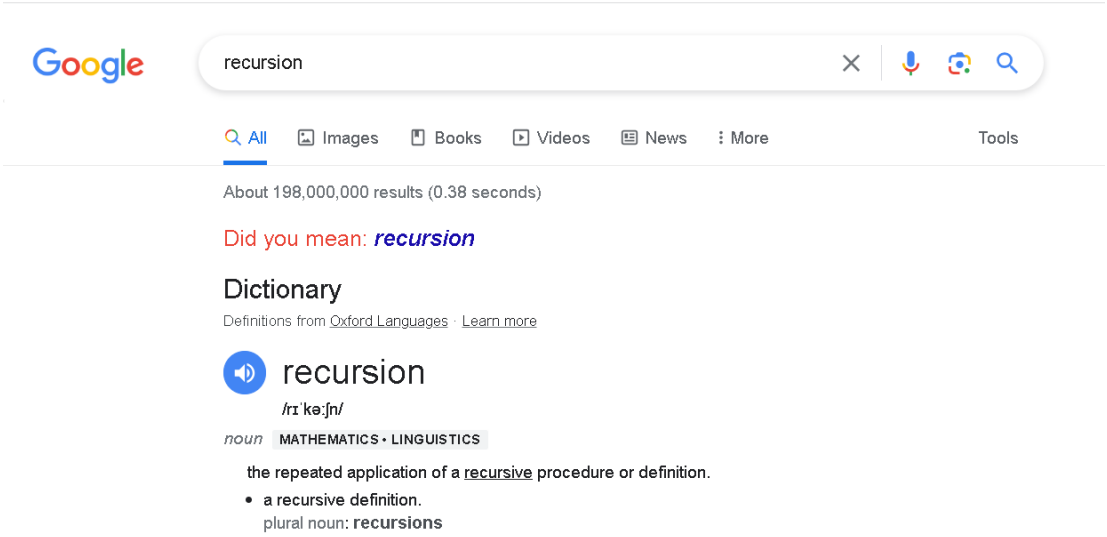


Figure 1. Google Search for recursion

Introduction

You have learned about while loops, Python lists and how to manipulate the elements in a list using for-loops and list comprehensions. In this lab you will learn about recursion. Recursion can be used to iterate through complicated data structures and to replicate the functionality offered by while loops and for loops. We will look at some examples of recursive functions and the type of problems recursion can help solve.

Tools/Software Requirement
Python 3/ PyCharm IDE / MS Visual Studio

Contents

1 A Simple Recursive Function

How does Python evaluate this function?

```
def count(n):
    print(n)
    count(n + 1)
count(0)
```

This function never stops running. It follows this procedure:

1. To count from `n` to infinity, print `n`.
2. Continue counting from `n + 1` to infinity.

Notice the second step is self-referential. This instruction was translated into line 3 of the **`count()`** function. This line is an example of a recursive call where the function calls itself. In this lab, we will explore these ***recursive functions***.

2 Termination

How do you make a recursive function stop running? Don't allow it to call itself every time. For example, this function only calls itself when its argument `problems` is greater than 1.

```
def how_to_do_homework(problems):
    print(f"How to do {problems} homework problems")
    if problems <= 1:
        print("Do the problem and then you're done.")
    else:
        print("Do the first problem and then do {problems} problems")
        problems -= 1  # get rid of one problem
        how_to_do_homework(problems) # then do the rest of the problems
```

If we call the function with the argument `problems` set to 5, Python will print this output:

```
1 How do you do 5 homework problems?
2 Do the first one and then do 4 problems.
3 How do you do 4 homework problems?
4 Do the first one and then do 3 problems.
5 How do you do 3 homework problems?
6 Do the first one and then do 2 problems.
7 How do you do 2 homework problems?
8 Do the first one and then do 1 problems.
9 How do you do 1 homework problems?
10 Do the problem and then you're done.
```

3 Examples with One Recursive Call

3.1 Recursion in Mathematics

Recursive calls are not just for iterating through data, they can also be used for computations of arbitrary length. A common example of a recursive function is the factorial. The factorial of a number is the product of all numbers from 1 to that number, for example

$$\begin{aligned}
&\text{factorial of } 1 = 1\\
&\text{factorial of } 2 = 1 \cdot 2 = 2\\
&\text{factorial of } 3 = 1 \cdot 2 \cdot 3 = 6\\
&\text{factorial of } 4 = 1 \cdot 2 \cdot 3 \cdot 4 = 24\\
&\text{factorial of } 50 = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdots\cdots 49 \cdot 50\\
&\qquad = 30414093201713378043612608166064768844377641568960512000000000000
\end{aligned}$$

The code is very close to a mathematical definition of the factorial:

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

In order to compute the factorial of an integer, you must compute the factorial of a smaller integer. Some examples. Note, unlike many other programming languages Python supports huge integers.

```
>>> factorial(1)
1
>>> factorial(4)
24
>>> factorial(50)
30414093201713378043612608166064768844377641568960512000000000000
```

Lab Activity: expo.py . Write a recursive function that takes in a base and an exponent and returns the value of that base taken to that exponent. You are not allowed to use ** or any built-in exponentiation functions.

3.2 Iterating through Lists

Here is an example function that adds 1 to every element in a list. The function first checks if the list is non-empty. If so, it adds one to the first element and appends that element to the rest of the list. The rest of the list is passed as an argument to the recursive call.

This code is an ***unrealistic*** way of traversing a list. It would be better to use a list comprehension or a for-loop.

```
def add(elements):
    if len(elements) == 0:
        return []
    first_element = elements[0] + 1 # add 1 to the first element
    rest_of_list = elements[1:] # shrink the list
    # work on the rest of the list
    return [first_element] + add(rest_of_list)
```

```
print(add([1,2,3])) #calling the above function
```

What happens when we call this function? Here is a step-by-step example of running `add([1, 2, 3])`.

Input of the first function call: `[1, 2, 3]`

First the function checks if the list is non-empty, then it adds 1 the first element and calls itself with a shorter input.

Input of recursive call 1: *[2, 3]*

This next function call also checks that the list is non-empty, it adds 1 to the first element and calls itself with an even shorter list.

Input of recursive call 2: *[3]*

Same thing happens with this input.

Input of recursive call 3: *[]*

There are no elements to add 1 to.

Output of recursive call 3: *[]*

When this function receives the empty list, it returns an empty list.

Output of recursive call 2: `[3 + 1] + []`

Function call 3 gets a list and adds an element to it.

Output of recursive call 1: `[2 + 1] + [3 + 1] + []`

Function call 2 also gets a list and adds an element to it.

Output of the first function call: `[3 + 1] + [2 + 1] + [3 + 1] + [] = [4, 3, 2]`

Finally, function call 1 gets the last two elements, adds the very first element after adding 1 and then returns final result.

When a function makes a recursive call by calling itself with a smaller input, it is requesting that the same thing be done to that smaller input. In this case, the first function call to `add` added 1 to the first value of the list and then passed off the rest of the work to function call 2. Function call 2 added 1 to the second element of the list, and passed the rest of the work on to function call 3. This happened until there were no more numbers in the list.

4 Examples with Multiple Recursive Calls

4.1 Iterating through Nested Lists

Recursion is especially useful when dealing with deeply nested data structures. These include trees, graphs, and other data structures. We will focus on nested lists. Consider this list:

`[1, 2, [3, 4, [5, 6]], [7]]`

What if we want to add 1 to every number in this list?

A useful code fragment when dealing with such a list is **type(element) == list**. This checks if the value of **element** is a Python list. We can use it to decide whether to add 1 to an element of a nested list or to add 1 to all of its sub-elements.

```
def add1(nested_list):
    new_list = []
    for element in
        nested_list: if
        type(element) == list:
            # if the element is a list
            # add 1 to each of its elements
            new_list.append(add1(element))
        else:
            # otherwise, just add 1 to the element
            new_list.append(element + 1)
    return new_list
```

We get these results:

```
>>> add1([1, 2, 3])
[2, 3, 4]
>>> add1([1, 2, [3, 4, [5, 6]], [7]])
[2, 3, [4, 5, [6, 7]], [8]]
```

Here is a description of what the function `add1` did in the second example.

1. The function starts iterating through the given list.

2. **`element = 1`**

   The first element is an integer, so

   `new_list = [1 + 1]`

3. **`element = 2`**

   `new_list = [2, 3]`

4. **`element = [3, 4, [5]]`** We need to make a recursive call to add 1 to every element of this sub-list. Essentially, the list looks like this now:

   `new_list = [2, 3, add1([3, 4, [5]])]`

   The recursive call to `add1` makes its own recursive call in order to compute the last element of its return value: **`[4, 5, add1([5])]`.**

   The end result is **`[4, 5, [6]]`.** So, after two recursive calls, we have:

   `new_list = [2, 3, [4, 5, [6]]]`

5. **`element = [7]`**

   This is a list, so a recursive call is made. A for loop iterates through the one-element list. In this recursive call **`element = 7`**, so 8 is added to the new list and the recursive call to the function returns **`[8]`.**

   `new_list = [2, 3, [4, 5, [6]], [8]]`

6. **`add1`** has finished iterating through the list.

We can also think of this function like this: add one to every number in the list, add one to every element in all sub-lists.

First function call `add1([1, 2, [3, 4, [5, 6]], [7]])`

Recursive call 1 `[2, 3, add1([3, 4, [5, 6]]), add1([7])]`

Recursive call 2 `[2, 3, [4, 5, add1([5, 6])], [8]]`

End result `[2, 3, [4, 5, [6, 7]], [8]]`

Here is another function that works on nested lists. It counts the number of integers or other "non-lists" in a nested list. For example,

```
>>>   elements_in([])
0
>>>   elements_in([[[[]]]])
0
>>>   elements_in([100, 200])
2
>>>   elements_in([1, 2, [3, 4, [5, 6]], [7]])
7
```

This is the code for this recursive function. Recursive calls are made to count the number of elements in each sub-list. Try to run this code.

```
def elements_in(nested_list):
    count = 0
    for element in nested_list:
        if type(element) == list:
        # count the number of elements in a sub-list
          count += elements_in(element)
        else:
            # otherwise, just add 1 to the count
            count += 1
    return count
```

See **Task02**

5 Overview

- A typical recursive function calls itself one or more times.
- A recursive function that finishes has a condition that causes it to **stop** calling itself.
- Whenever a problem needs small parts of its own solution in order to be solved, a recursive function may make a good solution.

Recursion is used to process nested data structures, sort lists of data, to read and run computer programs, solve mathematical problems, and to solve many other types of problems. Anything you can do with a while loop or a for loop can be done using a recursive function.

Lab Tasks

Task 01. (Greatest Common Divisor)

The greatest common divisor of integers `x` and `y` is the largest integer that evenly divides into both `x` and `y`. Write and test a recursive function `gcd` that returns the greatest common divisor of `x` and `y`. The `gcd` of `x` and `y` is defined recursively as follows:
If `y` is equal to `0`, then `gcd(x, y)` is `x`; otherwise,
`gcd(x, y)` is `gcd(y, x % y)`, where % is the remainder operator.

Task 02. `nested.py`.

A nested list is a list that contains one or more lists as elements. For example, the list `[1,2,3,[50,60,70],[[8888],999],10]` contains 6 elements.

- The first three elements are integers.
- The fourth element is a list: `[50,60,70]`.
- The fifth element is also a list. This list has two elements: another list `([[8888],999])` and `10`.

The following functions should have **recursive** definitions.

1. Write a function named `element_of` that returns `True` if the first argument is within any of the sub-lists of the nested list and `False` otherwise.

2. Write a function named `filter_by_depth` that takes two arguments: an integer representing **depth** and **a nested list**. It should remove all sub-lists that are more than **depth** deep.

Hint: use the **type** function to check if an element is a list.
Test Cases:

```
>>> nested.element_of(5, [1,2,3,4,5,6,7])
True
>>> nested.element_of(7, [1,2,[3,4,[5,6]],[7]])
True
>>> nested.element_of(77, [1,2,[3,4,[5,6]],[7]])
False
>>> nested.filter_by_depth(0, [1,2,3])
[]
>>> nested.filter_by_depth(1, [1,2,3])
[1,2,3]
>>> nested.filter_by_depth(5, [1,2,3])
[1,2,3]
>>> nested.filter_by_depth(2, [1,2,[3,4,[5,6]],[7]])
[1,2,[3,4],[7]]
```

Task 3. Peasants.py

a. Russian peasant multiplication is a method of multiplying integers in which you keep halving one factor while doubling the other factor until one of the factors is 1. For example:

8 × 38
= 4 × 76
= 2 × 152
= 1 × 304

Hence, **8 × 38 = 304.**

This, of course, becomes a bit more involved if the integer that you keep halving is **not a power of two**. For example, imagine 38 is the number that keeps getting halved:

8 × 38
= 16 × 19
= 32 × 9 + 16 19//2 = 9 with remainder 1
= 64 × 4 + 16 + 32 9//2 = 4 with remainder 1
= 128 × 2 + 16 + 32
= 256 × 1 + 16 + 32
= 256 + 16 + 32
= 304

This is all based on the following recursive definition of multiplication:

$$x * y = \begin{cases} \frac{x}{2} * (2 * y), & \textbf{if } x \textbf{ is even} \\ \frac{x-1}{2} * (2 * y) + y, & \textbf{if } x \textbf{ is odd} \end{cases}$$

Write a non-recursive function that implements Russian peasant multiplication using a **`while`** loop. Call this function **`multiply()`.**

b. The **Russian peasant method** can also be applied to exponentiation (also only for integers):

$$x^y = \begin{cases} (x^2)^{y/2}, & \textbf{if } y \textbf{ is even} \\ x(x^2)^{(y-1)/2}, & \textbf{if } y \textbf{ is odd} \end{cases}$$

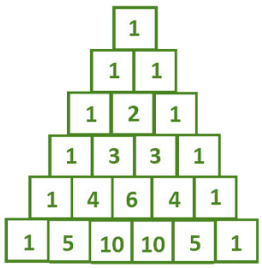
This method is also called exponentiation by squaring or the square-and-multiply method. Write a recursive function called **`expo()`** that implements this.

Task 04. pascal.py

In Pascal's triangle, each number is the sum of the two numbers directly above it. The left and right ends of the triangle always consists of 1s. This recursive rule produces the following triangle:

Row 0
Row 1
Row 2
Row 3
Row 4
Row 5
Row 6
Row 7
Row 8
Row 9

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5   10  10  5   1
        1   6   15  20  15  6   1
      1   7   21  35  35  21  7   1
    1   8   28  56  70  56  28  8   1
  1   9   36  84  126 126 84  36  9   1
```



Write a recursive function **pascal(n, k)** that finds the kth value of the nth row by using the sum of the numbers directly above it. We start counting at 0 for both n and k.
Since we know that the left and right end of the triangle are all 1s, we know that for every row n:

```
pascal(n, 0) = 1          left end
pascal(n, n) = 1          right end
```

If for example I want to know the 2nd entry of the 4th row:

pascal(4, 2) = pascal(3, 1) + pascal(3, 2)
= pascal(2, 0) + pascal(2, 1) + pascal(2, 1) + pascal(2, 2)
= 1 + pascal(1, 0) + pascal(1, 1) + pascal(1, 0) + pascal(1, 1) + 1
= 1 + 1 + 1 + 1 + 1 + 1
= 6

Happy Coding!

Deliverables

Comment your program heavily. Intelligent comments and a clean, readable formatting of your code account for 20% of your grade.
You should submit your codes and report as a compressed zip file. It should contain all files used in the exercises for this lab.
The submitted file should be named cs250_firstname_lastname_lab08.zip