Recursion

Elshad Karimov

June 30, 2021

This is a personal learning note for Udemy course: The Complete Data Structures and Algorithms Course in Python – Elshad Karimov, Software Developer.

1 Basic Knowledge

1.1 What is Recursion

Recursion = a way of solving a problem by having a function calling itself.

- Performing the same operation multiple times with different inputs.
- In every step we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion, otherwise infinite loop will occuri.

```

def openRussianDoll(doll):
  if doll == 1:
    print("All dolls are opened")
  else:
    openRussianDoll(doll-1)

```

Listing 1: 1st Standard Example

1.2 Why Recursion?

- Recursive thinking is really important in programming and it helps you break down big problems into smaller ones and easier to use. when to choose recursion?
    1. If you can divine the problem into similar sub problems.
    2. Design an algorithm to compute nth...
    3. Write code to list the n...
    4. Implement a method to compute all.
    5. Practice.
- The prominenent usage of recursion in date structures like treees and graphs.
  So when you are dealing with trees, the recursion becomes almost mandatory to use.
- Interviews.
- It is used in many algorithms. (Divide and conquer, greedy and dynamic programming)

1.3 How Recursion Works?

- 1. A emthod calls it self.
- 2. Exit from infinite loop.

```
def recursionMethod(parameters):
  if exit from condition satisfied:
    return some value
  else:
    recursionMethod(modified parameters)
```

Listing 2: 2nd Standard Example

```
def firstMethod():
  secondMethod()
  print("I am the first method")

def secondMethod():
  thirdMethod()
  print("I am the second method")

def thirdMethod():
  fourthMethod()
  print("I am the third method")

def fourthMethod()
  print("I am the fourth method")
```

Listing 3: 1st Good Example

```
def recursiveMethod(n):
  if n<1:
    print("n is less than 1")
  else:
    recursiveMethod(n-1)
    print(n)
```

Listing 4: 2nd Good Example

Stack memory is controlled by system to call the recursive method.

1.4 Recursive VS Iterative Solutions

Recursive method example:

```
def powerOfTwo(n):
  if n == 0:
    return 1
  else:
    power = powerOfTwo(n-1)
    return power*2
```

Listing 5: Recursive Method Example

Iterative method example:

```
def powerOfTwoIt(n):
  i = 0
  power = 1
  while i<n :
    power = power*2
    i = i+1
  return power
```

Listing 6: Iterative Method Example

Points	**Recursion**	**Iteration**	
Space Efficient?	No	Yes	No Stack Memory Require in Case of Iteration
Time Efficient?	No	Yes	In Case of Recursion Needs More Time for Pop and Push Elements to Stack Memory which Makes Recursion Less Time Efficient
Easy to Code?	Yes	No	We use Recursion Especially in the Cases We Know Threat a Problem can be Divided into Similar Sub-problems.

1.5 When to Use/Avoid Recursion?

1.5.1 When to use it?

- When we can easily breakdown a problem into similar sub-problems.
- When we are fine with extra overhead (both time and space) that comes with it.

- When we need a quick working solution instead of efficient one. (Solving mathematical problems like: Factorial or Fibonacci)
- It is very useful when we traverse a Tree.
- When we use memorization in recursion.
  - This means that if you memorize the result by saving the value of each calculation for further use in the recursive call, you can in factor reduce the time complexity.

1.5.2 When avoid it?

- If time and space complexity matters for us.
- Recursion uses more memory. If we use embedded memory. For example an application that takes more memory int the phone is not efficient.
  - If you are developing a mobile application, which should run on low memory devices as well.
- Recursion can be slow
  - If you are developing a real-time application like air-bag in the car system.

1.6 How to Write Recursion in 3 Steps

1.6.1 Example of Factorial

- It is the product of all positive integers less than or equal to n.
- Denoted by n!
- Only positive numbers
- 0!=1.

Example 1: $4! = 4 \times 3 \times 2 \times 1 = 24$.

Example 2: $10! = 3,628,800$.

Its general term formula is $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 = n \times (n-1)!$

- **Step 1: Recursive Case - the Flow**
  - Get the general term formula, such as $n! = n \times (n-1) \times (n-2) \ldots \times 2 \times 1 \rightarrow n! = n \times (n-1)!$
  - $(n-1)! = (n-1) \times (n-1-1) \times (n-1-2) \times \ldots \times 2 \times 1 = (n-1) \times (n-2) \times (n-3) \times \ldots \times 2 \times 1$
- **Step 2: Base Case - the Stopping Criterion**
  - $0! = 1$
  - $1! = 1$
- **Step 3: Unintentional Case - the Constraint**
  - factorial(-1) ??
  - factorial(1.5) ??

  Please see the python file recursion.py.

1.6.2 Example of Fibonacci Number - Recursion

Fibonacci sequence is a sequence of numbers in which each number is the sum of the two preceding ones and the sequence starts from 0 and 1. 152 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

- **Step 1: Recursive Case - the Flow** $5 = 3 + 2$, and $f(n) = f(n-1) + f(n+2)$
- **Step 2: Base Case - the Stopping Criterion**
  - 0 and 1
- **Step 3: Unintentional Case - the Constraint**

2 Interview Questions

2.1 Question: How to find the sum of digits of a positive integer number using recursion?

- **Step 1: Recursive Case - the Flow** So the general term formula is $f(n) = n\%10 + f(n/10)$.

Table 1: Table Example of the flow

Number	Quotient	Remainder
10	10/10 = 1	Remainder = 0
54	54/10 = 5	Remainder = 4
112	112/10 = 11	Remainder = 2
11	11/10 = 1	Remainder =1

- **Step 2: Base case - the stopping criterion**
  - n = 0
- **Step 3: Unintentional Case - the Constraint**
  - sumofDigits(-11) ??
  - sumofDigits(1.5) ??

```python
def digits_sum(n):
  assert n>=0 and int(n)==n, "the input number should be the non negative number"
  if n < 10:
    return n
  else:
    return n%10+digits_sum(int(n/10))

print(digits_sum(1))
```

Listing 7: Solution for Sum of Digits in Recursion

2.2 How to calculate power of a number using recursion?

- **Step 1: Recursive Case - the Flow** $x^n = x \times x \times \ldots\ldots x$ and $2^4 = 2 \times 2 \times 2 \times 2$

2.3 How to find GCD (Greatest Common Divisor) of two numbers using recursion?

- **Step 1: Recursive Case - the Flow** GCD is the largest positive integer that divides the numbers without a remainder. For example, if $gcd(8,12)$, it should equal 4. If we want to have the GCD of $gcd(48,18)$, the **Euclidean Algorithm** for GDC calculation is:
  - 1. $48/18 = 2$, remainder 12.
  - 2. $18/12 = 1$, remainder 6.

    - 3. 12/6 = 2. remainder 0.
    - 4. Additionally, $gcd(b,a) = gcd(b, a\%b)$ and $gcd(a,0) = a$.

- **Step 2: Base Case - the Stopping Criterion**
    - b=0.
- **Step 3: Unintentional Case - the constraint**
    - Positive Integers
    - Convert Negative Numbers to Positive

```python
def gcd(q1, q2):
  assert int(q1) == q1 and int(q2) == q2, 'The numbers must be integer only!'
  if q1 < 0:
    q1 = -1*q1
  if q2 < 0:
    q2 = -1*q2
  if q1%q2 == 0:
    return q2
  else:
    return gcd(q2, q1%q2)

print(gcd(32, 16))
```

Listing 8: Solution for GCD

2.4 How to Convert a Number from Decimal to Binary Using Recursion?

- **Step 1: Recursive Case - the Flow**
    - 1. Divide the number by 2.
    - 2. Get the integer quotient for the next iteration
    - 3. Get the remainder for the binary digit
    - 4. Repeat the steps until the quotient is equal to 0

Table 2: Example for 10 Converting to Binary Number

Division by 2	**Quotient**	**Remainder**	**Flow**
10/2	5	0	$0 + 10 \times (101) = 0 + 10 \times f(10 - 10\%2)$
5/2	2	1	$f(10 - 10\%2) = 1 + 10 \times f\Big((10 - 10\%2) - (10 - 10\%2)\%2\Big)$
2/2	1	0	$f\Big((10 - 10\%2) - (10 - 10\%2)\Big) = 0 + 10 \times f(\ldots)$
1/2	0	1	$1/2 = 0, \quad 1\%2$

```python
def d2b(n):
  if n<0:
    n = n*-1
  if n//2 == 0:
    return n%2
  else:
    return n%2+10*(d2b(n//2))

print(d2b(13))
```

Listing 9: Solution for Decimal Converting to Binary

3 Coding Exercise

- **Power** Write a function called power which accepts a base and an exponent. The function should return the power to the exponent. This function should mimic the functionality of math.pow()-do not worry about negative bases and exponents.

```python
def power(base, exponent):
    if exponent == 0:
        return 1
    return base * power(base, exponent-1)

print(power(2, 9))
```

Listing 10: Solution for productOfArray

- **productOfArray** Write a function called **productOfArray** which takes in an array of numbers and returns the product of them all.

```python
def productOfArray(arr):
    if len(arr) == 0:
        return 1
    else:
        return arr.pop()*productOfArray(arr)

print(productOfArray([1,2,3]))
```

Listing 11: Solution for productOfArray

- **recursiveRange** Write a function called **recursiveRange** which accepts a number and adds up all numbers from 0 to the number passed to the function.

```python
def recursiveRange(num):
    assert num >= 0, "the input number should be bigger than 0"
    if num == 0:
        return 0
    else:
        return num + recursiveRange(num-1)

print(recursiveRange(10))
```

Listing 12: Solution for recursiveRange

- **Fibonacci** Write a recursive function called **fib** which accepts a number and returns the n^{th} number in the Fibonacci sequence. Recall that the Fibonacci sequence is the sequence of whole numbers 0,1,2,3,5,8,... which starts with 0 and 1, and where every number thereafter is equal to the sum of the previous two numbers.

```python
def fib(num):
    assert num >=0, "num should be non negative"
    if num == 0:
        return 0
    if num == 1:
        return 1
    return fib(num-1)+fib(num-2)

print(fib(4))
```

Listing 13: Solution for fib

- **Reverse** Write a recursive function called **reverse** which accepts a string and returns a new string in reverse.

```python
def reverse(strng):
    if len(strng)==1:
        return strng
    else:
        return strng[len(strng)-1]+reverse(strng[0:(len(strng)-1)])
```

Listing 14: Solution for Reverse

Note: when strng[0:3] means strng[0]+strng[1]+strng[2].

- **isPalindrome** Write a recursive function called isPalindrome which returns true if the string passed to it is a palindrome (reads the same forward and backward). Otherwise it returns false.

```
def isPalindrome(strng):
    if len(strng) == 0:
        return True
    if strng[0] != strng[len(strng)-1]:
        return False
    return isPalindrome(strng[1:len(strng)-1])
```

Listing 15: Solution for isPalindrome

- **someRecursive** Write a recursive function called **someRecursive** which accepts an array and a callback. The function returns true if a single value in the array returns true when passed to the callback. Otherwise it returns false.

```
def isOdd(num):
    if num%2==0:
        return False
    else:
        return True

def someRecursive(arr, cb):
    if cb(arr.pop()) == True:
        return True
    if len(arr) == 0:
        return False
    return someRecursive(arr, cb)
```

Listing 16: Solving for someRecursive

- ⋆⋆ **flatten** Write a recursive function called **flatten** which accepts an array of arrays and returns a new array with values flattened.

Example

```
flatten([1,2,3,[4,5]]) # [1,2,3,4,5]
flatten([1,[2,[3,4],[[5]]]]) # [1,2,3,4,5]
flatten([[1],[2],[3]]) # [1,2,3]
flatten([[[[1], [[[2]]], [[[[[[[3]]]]]]]]]]) # [1,2,3]
```

Solution

```
def flatten(arr):
    line_list = []
    for item in arr:
        if type(item) is list:
            line_list.extend(flatten(item))
        else:
            line_list.append(item)
    return line_list
```

Note: python extend() and append() are different. The similar of these two method is to add the received cell at the tail of the list. But the extend() method can only accept the list and keep the item of the list into the original list position. However, the append() method can accept any type of data structure and simply add the item at the tail of the list.