Find Uppercase Letter in String

In this lesson, you will learn how to find the uppercase letter in a string using both an iterative and recursive approach in Python.

We'll cover the following

- Iterative Approach
- Recursive Approach

In this lesson, given a string, we develop an algorithm to return the first occurring uppercase letter. We will solve this problem using an iterative and recursive approach:

For instance, for the strings:

```
str_1 = "lucidProgramming"
str_2 = "LucidProgramming"
str_3 = "lucidprogramming"
```

The algorithm should return `P` `L`, and output a message indicating that no capital letter was found for `str_1`, `str_2`, and `str_3`, respectively.

Iterative Approach

Let's have a look at the code in Python, which uses the iterative approach:

```
def find_uppercase_iterative(input_str):
    for i in range(len(input_str)):
        if input_str[i].isupper():
                return input_str[i]
    return "No uppercase character found"
```

```
 5     return "No uppercase character found"
```

find_uppercase_iterative(input_str)

The `for` loop on **line 2** runs for all the characters present in `input_str`. By using the built-in function `isupper()`, every character of `input_str`, i.e., `input_str[i]` is checked if it is uppercase or not. If the condition on **line 3** evaluates to `True` for some `input_str[i]`, then that character is returned from the function on **line 4**. However, if the condition does not evaluate to `True` in any iteration of the `for` loop, `"No uppercase character found"` is returned from the function on **line 5** to indicate that there was no uppercase in `input_str`.

Recursive Approach

The iterative approach was very straightforward. Let's look at the recursive approach in the snippet below:

```
 1  def find_uppercase_recursive(input_str, idx=0):
 2    if input_str[idx].isupper():
 3      return input_str[idx]
 4    if idx == len(input_str) - 1:
 5      return "No uppercase character found"
 6    return find_uppercase_recursive(input_str, idx+1
```

find_uppercase_recursive(input_str, idx=0)

`find_uppercase_recursive()` takes in `input_str` and `idx` as input parameters. To provide some starting point, the second parameter is written as `idx = 0` which will set `idx` to `0` if no second parameter is provided when the function is called.

The base case is present on **line 2** which returns `input_str[i]` if it is an uppercase. On the other hand, if we reach somewhere in the recursive calls where `idx` is equal to `len(input_str) - 1`, i.e., we have reached the end of the string but didn't find any character which was uppercase. Therefore, we return `"No uppercase character found"` to indicate so. However, if both the conditions on **line 2** and **line 4** are not `True`, we make a recursive call on **line 6** and pass `input_str` and `idx + 1` so that the next character is evaluated.

Let's go ahead and run these two codes in the code widget below.

```
def find_uppercase_iterative(input_str):
    for i in range(len(input_str)):
        if input_str[i].isupper():
            return input_str[i]
    return "No uppercase character found"

def find_uppercase_recursive(input_str, idx=0):
    if input_str[idx].isupper():
        return input_str[idx]
    if idx == len(input_str) - 1:
        return "No uppercase character found"
    return find_uppercase_recursive(input_str, idx

input_str_1 = "lucidProgramming"
input_str_2 = "LucidProgramming"
input_str_3 = "lucidprogramming"

print(find_uppercase_iterative(input_str_1))
print(find_uppercase_iterative(input_str_2))
print(find_uppercase_iterative(input_str_3))

print(find_uppercase_recursive(input_str_1))
print(find_uppercase_recursive(input_str_2))
print(find_uppercase_recursive(input_str_3))
```

▷

✕

```
Output                                                    0.88s
```

```
P
L
No uppercase character found
P
L
No uppercase character found
```

That’s it for this problem. Yes, it was that simple! In the next lesson, we will discuss how to calculate the length of a string.

← Back
Quiz

Next →
Calculate String Length

☑ Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/find-uppercase-letter-in-string__recursion__data-structures-and-algorithms-in-python)

Calculate String Length

In this lesson, you will learn how to calculate the length of a string using both an iterative and recursive approach in Python.

We'll cover the following

- Iterative Approach
- Recursive Approach

In this lesson, we focus on the following problem:

Given a string, calculate its length.

If you are preparing for an interview or trying to understand the notion of recursion to solve a problem, I hope this lesson is helpful to you.

Python has a built-in `len()` function which returns the length of a string. This is the standard way to obtain the length of a string in Python.

```python
input_str = "LucidProgramming"
print(len(input_str))
```

Output 0.83s

```
16
```

Iterative Approach

Now we are going to code the same functionality ourselves in Python. Let’s begin with the iterative approach.

```
1  #·Iterative·length·calculation:·O(n)
2  def·iterative_str_len(input_str):
3  ····input_str_len·=·0
4  ····for·i·in·range(len(input_str)):
5  ········input_str_len·+=·1
6  ····return·input_str_len
```

On **line 3**, `input_str_len` is initialized to `0`. Then using a `for` loop on **line 4**, `input_str` is iterated character by character and `input_str_len` is incremented by `1` in each iteration on **line 5**. Finally, the final value of `input_str_len` is returned from the function on **line 6**. As the entire length of the string is traversed once, the time complexity for this solution is thus $O(n)$ where n is the length of the string.

Recursive Approach

Let’s go ahead and have a look at the recursive approach:

```
1  #·Recursive·length·calculation:·O(n)
2  def·recursive_str_len(input_str):
3  ····if·input_str·==·'':
4  ········return·0
5  ····return·1·+·recursive_str_len(input_str[1:])
```

The base case for this function is when an empty string is encountered. If `input_str` is empty, `0` is returned to make a count of `0` for the empty string. Otherwise, `1` is added to whatever is returned from the recursive call on **line 5** which takes in `input_str[1:]`. The slicing notation in `input_str[1:]` indicates that all the characters except at the `0th` index are passed into the recursive call. Therefore, every recursive call keeps shortening `input_str` by one character, which is being counted in that recursive call. As there will be n recursive calls, each expending a constant amount of computational effort, the time complexity for this solution is $O(n)$ where n is the length of the string. Wasn't this pretty simple? I hope you are clear about everything that we have studied so far.

In the code widget below, you can run and play with both the iterative and recursive implementations.

```python
# Iterative length calculation: O(n)
def iterative_str_len(input_str):
    input_str_len = 0
    for i in range(len(input_str)):
        input_str_len += 1
    return input_str_len

# Recursive length calculation: O(n)
def recursive_str_len(input_str):
    if input_str == '':
        return 0
    return 1 + recursive_str_len(input_str[1:])

input_str = "LucidProgramming"

print(iterative_str_len(input_str))
print(recursive_str_len(input_str))
```

Output 0.76s

```
16
16
```

Let’s have a look at another problem which we can solve using recursion in the next lesson!

← Back

Find Uppercase Letter in String

Next →

Count Consonants in String

☑ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/calculate-string-length__recursion__data-structures-and-algorithms-in-python)

Count Consonants in String

In this lesson, you will learn how to count consonants of a string using both an iterative and recursive approach in Python.

We'll cover the following

In this lesson, we focus on the following problem:

Given a string, calculate the number of consonants present.

The vowels are the following letters:

```
a e i o u
```

Any other letter that is not a vowel is a consonant.

Let’s have a look at an example:

```
Welcome to Educative!
```

has `9` consonants.

Before you dive into the implementation, consider the edge cases such as spaces and exclamation marks. This implies that if a character is not a vowel, it has to be a letter to be considered a consonant.

Iterative Approach

Check out the iterative approach in the snippet below:

```
1  vowels = "aeiou"
2
3  def iterative_count_consonants(input_str):
4      consonant_count = 0
5      for i in range(len(input_str)):
6          if input_str[i].lower() not in vowels and
7              consonant_count += 1
8      return consonant_count
```

iterative_count_consonants(input_str)

On **line 1**, we define `vowels` globally to be `"aeiou"`. `input_str` is the given string passed to `iterative_count_consonants` function which we have to process to calculate the number of consonants. `consonant_count` is initialized to `0` on **line 4** and then the `input_str` is traversed character by character using a `for` loop on **line 5**. Note that `input_str[i]` is changed to lowercase using `lower()` because `vowels` contains all the vowels in lowercase so that we can compare them correctly on **line 6**. As discussed before, we check if `input_str[i].lower()` is present in `vowels` or not. Additionally, `input_str[i].isalpha()` should also be `True` for the condition on **line 6** to be `True`. Thus, if `input_str[i]` is not a vowel and an alphabet, then `consonant_count` is incremented by `1` on **line** 7. After the condition on **line 6** is evaluated for every character in `input_str`, the final count of `consonant_count` is returned from the function on **line 8**.

The running time complexity for the iterative approach is $O(n)$ because, for each character in the input string of length n, we spend a constant amount of effort.

Recursive Approach

The implementation of the recursive approach will be very similar to the recursive implementations we have covered in the previous lessons. Check it out below:

```
 1  vowels = "aeiou"
 2  
 3  def recursive_count_consonants(input_str):
 4      if input_str == '':
 5          return 0
 6  
 7      if input_str[0].lower() not in vowels and inpu
 8          return 1 + recursive_count_consonants(inpu
 9      else:
10          return recursive_count_consonants(input_s
```

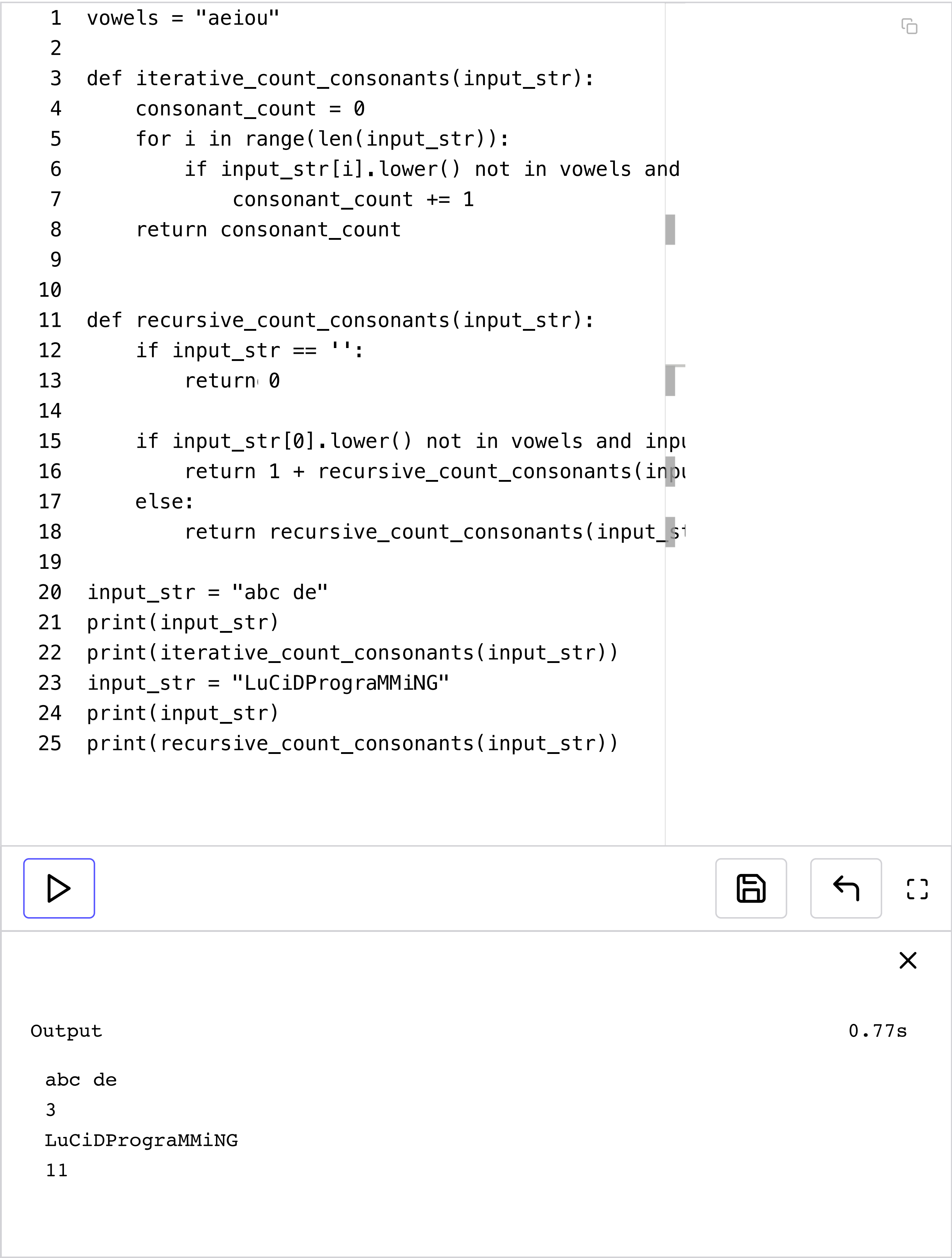
recursive_count_consonants(input_str)

Here, `vowels` is again defined globally on **line 1**. Now let's come to `recursive_count_consonants` function on **line 3**. If `input_str` is empty, `0` is returned from the function. This is the base case for `recursive_count_consonants`.

On **line** 7, we have the same condition as in the iterative implementation. If the condition on **line** 7 evaluates to `True`, we add `1` in addition to the count returned from the recursive call where a truncated string is passed using the slice notation (**line 8**). Otherwise, if the condition on **line** 7 is not `True` and `input_str[0]` is not a consonant, `1` is not added to the count, but a recursive call is made by passing `input_str[1:]` on **line 10**. At every recursive call, the first character is evaluated while the rest of the characters are passed into the next recursive calls.

The running time complexity for the recursive approach is $O(n)$ because, for each character in the input string of length n, we make a recursive call and each recursive call has constant time complexity.

In the code widget below, you can find both the implementations. Go ahead and play around with them using your test cases.



```python
vowels = "aeiou"

def iterative_count_consonants(input_str):
    consonant_count = 0
    for i in range(len(input_str)):
        if input_str[i].lower() not in vowels and
            consonant_count += 1
    return consonant_count


def recursive_count_consonants(input_str):
    if input_str == '':
        return 0

    if input_str[0].lower() not in vowels and inpu
        return 1 + recursive_count_consonants(inpu
    else:
        return recursive_count_consonants(input_s

input_str = "abc de"
print(input_str)
print(iterative_count_consonants(input_str))
input_str = "LuCiDPrograMMiNG"
print(input_str)
print(recursive_count_consonants(input_str))
```

Output 0.77s

```
abc de
3
LuCiDPrograMMiNG
11
```

I hope you have by now grasped the logic of how we have solved problems in this chapter using recursion! Let’s check that in the coding challenge waiting for you in the next lesson.

← Back

Calculate String Length

Next →

Exercise: Product of Two Positive Inte...

☑ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/count-consonants-in-string__recursion__data-structures-and-algorithms-in-python)

Exercise: Product of Two Positive Integers

Challenge yourself with an exercise in which you'll have to find the product of two positive integers.

We'll cover the following

- Problem
- Coding Time!

Problem

Given two numbers, find their product using recursion. In Python, we usually use the `*` operator to multiply two numbers, but you have to use recursion to solve this challenge. Make use of the hint given below.

Hide Hint

5 * 3 = 5 + 5 + 5 = 15

Coding Time!

Your task is to return the product of the two positive integers `x` and `y` from the function `recursive_multiply(x,y)` given in the code widget below. Make sure that you don’t use the built-in multiplication operator of Python.

Good luck!

```python
def recursive_multiply(x, y):
    if x == 1:
        return y
    else:
        return y + recursive_multiply(x-1, y)
```

Show Results Show Console

0.79s

3 of 3 Tests Passed

Result	Input	Expected Output	Actual Output	Reason
✓	recursive_multiply(5,3)	15	15	Succeeded
✓	recursive_multiply(500,2000)	1000000	1000000	Succeeded
✓	recursive_multiply(34,0)	0	0	Succeeded

← Back

Count Consonants in String

Next →

Solution Review: Product of Two Posit...

☑ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/exercise-product-of-two-positive-integers__recursion__data-structures-and-algorithms-in-python)

Solution Review: Product of Two Positive Integers

This lesson contains the solution review for the challenge to find the product of two numbers.

We'll cover the following

- Implementation
- Explanation

Let's discuss the solution to the challenge in the previous lesson. The problem was to find the product of two positive integers.

Implementation

Let's have a look at the implementation below:

```
def recursive_multiply(x, y):

    # This cuts down on the total number of
    # recursive calls:
    if x < y:
        return recursive_multiply(y, x)
    if y == 0:
        return 0
    return x + recursive_multiply(x, y-1)

x = 500
y = 2000

print(x * y)
print(recursive_multiply(x, y))
```

```
Output                                                          0.72s

 1000000
 1000000
```

Explanation

The hint indicated the following:

```
5 * 3 = 5 + 5 + 5 = 15
```

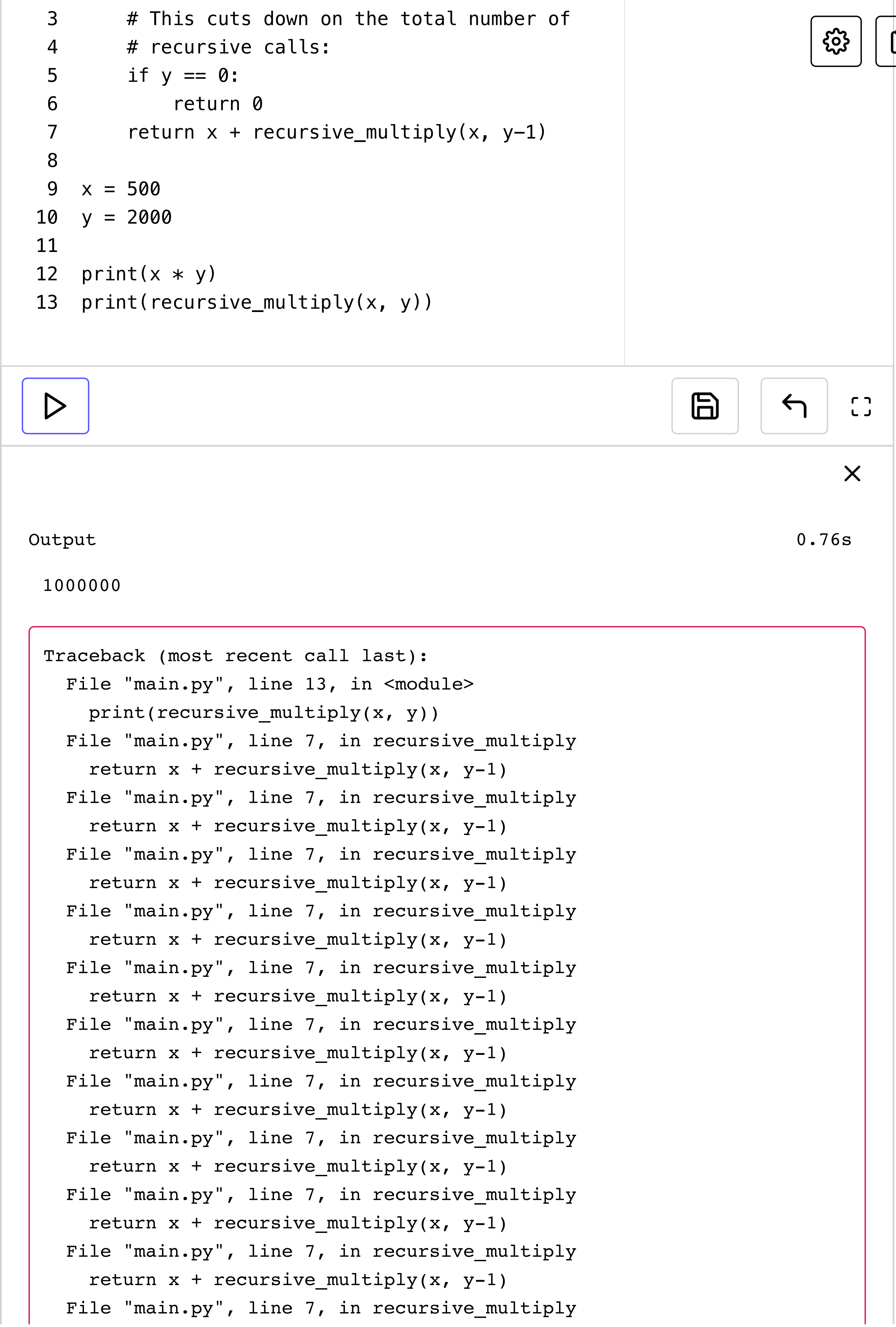
We make use of the hint in the implementation. Let's skip the code on **lines 5-6** for a while and discuss the code afterward. On **line 7**, we check if `y` equals `0`. If it does, `0` is returned on **line 8**. Otherwise, `x` is added to the sum returned from the recursive call on **line 9**. `y-1` is passed to the next recursive call as `x` is added once in the current recursive call. So overall, `x` will be added together `y` times in all the recursive calls. This will return the product of `x` and `y` at the end of all the recursive calls.

Now, in the implementation provided above, we make `y` recursive calls so if `x` equals `500` and `y` equals `2000`, we get the following error:

```
RecursionError: maximum recursion depth exceeded in comparison
```

if we skip the **lines 5-6** from the above implementation. Check out the code below:

```
1  def recursive_multiply(x, y):
2
```



```
    # This cuts down on the total number of
    # recursive calls:
    if y == 0:
        return 0
    return x + recursive_multiply(x, y-1)

x = 500
y = 2000

print(x * y)
print(recursive_multiply(x, y))
```

Output 0.76s

```
1000000
```

```
Traceback (most recent call last):
  File "main.py", line 13, in <module>
    print(recursive_multiply(x, y))
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
```

```
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 7, in recursive_multiply
    return x + recursive_multiply(x, y-1)
  File "main.py", line 5, in recursive_multiply
    if y == 0:
RecursionError: maximum recursion depth exceeded in comparison
```

We get `maximum recursion depth exceeded in comparison` whenever the depth of the recursion tree exceeds a limit.

Therefore, we add the following lines:

```
if x < y:
        return recursive_multiply(y, x)
```

In the code above, we swap `y` and `x` to cut down on the number of recursive calls in case `x` is less than `y`. However, there isn't anything we can do if both `x` and `y` are large enough to cause the `Recursion Error: maximum recursion depth exceeded in comparison`.

With this, we come to an end to the chapter on recursion. In the next chapter, we'll explore quite a few problems on string processing.

← Back

Exercise: Product of Two Positive Inte...

Next →

Quiz

☑ Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/solution-review-product-of-two-positive-integers__recursion__data-structures-and-algorithms-in-python)