

CS 106

Introduction to Computer Science I

07 / 15 / 2026

Instructor: Michael Eckmann

Today's Topics

- Questions / comments?
- Recursion
- Lab work on nested loops

Factorial function using iteration

non-recursive (*aka iterative*) solution to the

factorial problem

```
def compute_factorial(num):
```

```
    temp_factorial = 1
```

```
    while num > 0:
```

```
        temp_factorial *= num
```

```
        num -= 1
```

```
    return temp_factorial
```

Recursion

A recursive function is a function that calls itself (directly or indirectly.)

- Recursion is often used when the problem to be solved can be easily implemented with recursion as opposed to an iterative technique.
- All problems that can be solved recursively can be solved iteratively.
- Sometimes though the recursive solution seems more appropriate and other times the iterative solution might be more appropriate.

Recursion

- Let's say we know the solution to some case of a problem.
- If we want to find a solution to a slightly more complex case of the problem, we might be able to use the solution to the easier problem as part of our solution.
- For example, if we know the factorial of the number 6 = 720, to find the factorial of 7, we can just multiply 7 * the factorial of 6, that is: $7! = 7 * 6!$.
- Here, the solution we know ($6! = 720$) is used as part of the solution to $7!$, which is $7 * 720 = 5040$.

Recursion

- So, more generally, if we know the solution to $(n-1)!$ We can find the solution to $n!$ by just multiplying $n * (n-1)!$.
- In addition to knowing this though, we need to also know that $0! = 1$. This is the simplest factorial.
- Now, powered with the knowledge that
 - $0!$ is 1 and
 - the fact that given $(n-1)!$, we can compute $n! = n * (n-1)!$we have enough information to compute the factorial of any positive integer.
- Do we all agree?

Recursion

- This is a recursive solution to the factorial problem because
 - we know how to solve the simplest case ($0!$) and
 - we know how to solve a more complex case given a solution to a slightly less complex case
 - $n! = n * (n-1)!$

Factorial function using recursion

```
# recursive compute_factorial function
```

```
# assumes num >=0
```

```
def compute_factorialR(num):
```

```
    if num == 0:
```

```
        return 1
```

```
    else:
```

```
        return num * compute_factorialR(num - 1)
```

Factorial function using recursion

- When the recursive function is called, if the number passed in as an argument is equal to 0, then 1 is returned and the function ends.
- That is the *base case*.
- If the number passed in as an argument is greater than 0, then the other line of the function is executed. This says to return *that **number** times the **result of another call** to the function with the argument of one less than the original argument*.
- This part of the function is the *recursive step*.

Factorial function using recursion

- Let's examine what happens when the number 1 is passed in as an argument to `compute_factorialR`

`fact_value = compute_factorialR( 1 )`

- The function would check if `1 == 0` and since that is `False`, the other code would execute which will end up calling `compute_factorialR(0)`.
- This second call to the function invokes another “copy” of the function. This second call checks if `0 == 0` and this is `True` so it returns 1.

Factorial function using recursion

- This value of 1 gets returned to the most recent call (and hence the second call to the function ends its execution.)
- This value of 1 gets multiplied by 1 (the value of num).
- The result, 1, gets returned to the original call of the function (and hence the first call to the function ends its execution.)
- If `compute_factorialR` was called with the number 2 as an argument, the function would end up being called three times.

Recursion

- When the function is recursively called, the prior call to the function *has not finished executing*. It is waiting on the result of this new function call.
- There will be *multiple active copies* of the recursive function when the else portion executes.
- The recursion ends when the base case is finally met. The base case then **returns** a result to the most recent call of the function and a sequence of **returns** occurs until the original function call gets its returned value.

Factorial function using recursion

- Let's look at a program that will illustrate, through print statements, what happens in what order when the function is called recursively.

Recursion

- Understand that the recursive Factorial example was for instructional purposes only.
- For the factorial function, in practice, we would always prefer the non-recursive (iterative) solution. It is more efficient in terms of (uses less) space (and will also be faster in practice.)

Recursion

- Direct vs. indirect recursion
 - When a function calls itself, it is direct recursion
 - When function1 calls function2 and function2 calls function1, it is indirect.
 - There can be more than just two function involved in the indirection.

Towers of Hanoi

- Three pegs
- Start with different size discs on first peg
- Move them all to the third peg using the following rules
 - Move only one disc at a time
 - Can't place a larger disc on a smaller disc
 - All discs must be on some peg except for the disc in transit between pegs

Towers of Hanoi

- Do a solution with 3 discs.
- Do a solution with 4 discs.
- See any patterns emerging?
- Is it recursive?

Towers of Hanoi

- To move N discs from first peg to third
 - Move topmost $N-1$ discs from first peg to second
 - Move the largest disc from the first peg to third
 - Move the $N-1$ discs from the second peg to the third

Towers of Hanoi

- During the solution, each peg takes on a role
 - Either the start peg, end peg, or intermediate peg.
 - Notice:
 - In the first and third parts, notice that they are the same as the whole, with the roles of the pegs changing.
 - First part:
 - Move from: first peg (start)
 - To: second peg (end)
 - Using as intermediate: third peg (intermediate)
 - Third part:
 - Move from: second peg (start)
 - To: third peg (end)
 - Using as intermediate: first peg (intermediate)

Towers of Hanoi

```
def move_tower(num_discs, start, end, intermed):  
    if num_discs == 1:  
        move_one_disc(start, end)  
    else:  
        move_tower(num_discs - 1, start, intermed, end)  
        move_one_disc(start, end)  
        move_tower(num_discs - 1, intermed, end, start)  
  
def move_one_disc(p1, p2):  
    print('Move the disc from peg', p1, 'to peg', p2)
```

Towers of Hanoi

- Let's put this in a program and run it.
- Does anyone think that an iterative solution would be possible? If so, do you think it'll be easier?
- You can think about this on your own if you wish and try to come up with an iterative (that is, non-recursive) solution.

Recursion

- Every recursive algorithm needs
 - At least one base case
 - At least one recursive step - And it must work towards the base case
- Tips on writing recursive algorithms
 - What can be described recursively in problem?
 - Figure out the base case(s) --- situations that can be directly solved
 - As you write the algorithm, trust that your recursive call(s) will work the way they are supposed to (that they will indeed solve the smaller instance of the problem)

Recursion

- Fibonacci numbers
 - 0,1,1,2,3,5,8,13,21,34,55,89, ...
- Let's write determine the number of odds in a list, iteratively and then recursively.
- Let's write an iterative version and a recursive version of summing up the elements of a list.