

Recursion

Model 1 Factorial Function

"In mathematics, the factorial of a non-negative integer n , denoted by $n!$, is the product of all positive integers less than or equal to n . For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$."

Source: <https://en.wikipedia.org/wiki/Factorial>

n	$n!$
0	1
1	1
2	2
3	6
4	24
5	120

1. Consider how to calculate $4! = 24$.

a) Write out all the numbers that need to be multiplied:

$$4! = 4 * 3 * 2 * 1$$

b) Rewrite the expression using $3!$ instead of $3 \times 2 \times 1$:

$$4! = 4 * 3!$$

2. Write expressions similar to #1b showing how each factorial can be calculated in terms of a smaller factorial. Each answer should end with a factorial (!).

a) $2! = 2 * 1!$

b) $3! = 3 * 2!$

c) $100! = 100 * 99!$

d) $n! = n * (n - 1)!$

3. What is the value of $0!$ based on Model 3? Does it make sense to define $0!$ in terms of a simpler factorial? Why or why not?

$0!$ is 1 (by convention for an empty product). We can't say $0 \times -1!$, because factorial is only defined for non-negative integers. At some point we need to define the solution in concrete terms, without referencing itself.

*If we repeatedly break down a problem into smaller versions of itself, we eventually reach a basic problem that can't be broken down any further. Such a problem, like $0!$, is referred to as the **base case**.*

4. Consider the following Python function that takes n as a parameter and returns $n!$:

```
1 def factorial(n):
2     # base case
3     if n == 0:
4         return 1
5     # general case
6     product = 1
7     for i in range(n, 0, -1):
8         product *= i
9     return product
```

- a) Review your answer to #2c that shows how to compute $100!$ using a smaller factorial. Convert this expression to Python by using the function above instead of the $!$ operator.

```
100 * factorial(99)
```

- b) Now rewrite your answer to #2d in Python using the variable n and the function above.

```
n * factorial(n - 1)
```

- c) In the source code above, replace the “1” on Line 6 with your answer from b). Then cross out Lines 7 and 8. Test the resulting function in a Python Shell. Does it still work?

Yes, amazingly.

- d) What specific function is being called on Line 6?

The factorial function that is being defined.

- e) Why is the `if` statement required on Line 3?

Without the base case, it would call itself forever (until running out of memory).

5. A function that refers to itself is called **recursive**. What two steps were necessary to define the recursive version of `factorial`?

1. Write the base case, which was implemented using an `if` statement.
2. Write the recursive case, which was implemented using a function call.

6. Was a loop necessary to cause the recursive version of `factorial` to run multiple times? Explain your reasoning.

No; the function keeps calling itself until the base case is reached. The `for` loop was removed when we added the recursive call.

Model 2 Fibonacci Numbers

The Fibonacci numbers are a sequence where every number (after the first two) is the sum of the two preceding numbers: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Source: https://en.wikipedia.org/wiki/Fibonacci_number

We can define a recursive function to compute Fibonacci numbers. Enter the following code into a Python Editor, and run the program to see the sequence.

```
1 def fibonacci(n):
2     # base case
3     if n == 1 or n == 2:
4         return 1
5     # general case
6     return fibonacci(n - 1) + fibonacci(n - 2)
7
8 def main():
9     for i in range(1, 6):
10        print(fibonacci(i))
11
12 main()
```

7. Based on the source code:

- a) How many function calls are needed to compute `fibonacci(3)`? Identify the value of the parameter `n` for each of these calls.

3 calls: `n=3`, `n=2`, `n=1`

- b) How many function calls are needed to compute `fibonacci(4)`? Identify the value of the parameter `n` for each of these calls.

5 calls: `n=4`, `n=3`, `n=2`, `n=1`, `n=2`

- c) How many function calls are needed to compute `fibonacci(5)`? Identify the value of the parameter `n` for each of these calls.

9 calls: `n=5`, `n=4`, `n=3`, `n=2`, `n=1`, `n=2`, `n=3`, `n=2`, `n=1`

8. Check your answers for the previous question by adding the following `print` statements to the code and rerunning the program:

- Insert `print("n is", n)` at Line 2, before the `# base case` comment
- Insert `print("fib({i}) is...")` at Line 10, before the `print` statement

9. What happens if you try to compute `fibonacci(0)` in the Python Shell?

The function is called with decreasing negative numbers until the Shell displays the error: "RuntimeError: maximum recursion depth exceeded while calling a Python object".

10. How could you modify the code so that this situation doesn't happen?

Add an `if` statement that returns -1 if $n < 1$.

Model 3 Summation

"In mathematics, summation (capital Greek sigma symbol: Σ) is the addition of a sequence of numbers; the result is their sum or total."

$$\sum_{i=1}^{100} i = 1 + 2 + 3 + \dots + 100 = 5050$$

Source: <https://en.wikipedia.org/wiki/Summation>

11. Consider how to calculate $\sum_{i=1}^4 i = 10$.

a) Write out all the numbers that need to be added:

$$\sum_{i=1}^4 i = 1 + 2 + 3 + 4$$

b) Show how this sum can be calculated in terms of a smaller summation.

$$\sum_{i=1}^4 i = 4 + \sum_{i=1}^3 i$$

12. Write an expression similar to #11b showing how any summation of n integers can be calculated in terms of a smaller summation.

$$\sum_{i=1}^n i = n + \sum_{i=1}^{n-1} i$$

13. What is the base case of the summation? (Write the complete formula, not just the value.)

$$\sum_{i=1}^1 i = 1$$

Here are important questions to consider before writing a recursive function:

- *How can you define the problem in terms of a smaller similar problem?*
- *What is the base case, where you solve an easy problem in one step?*
- *For the recursive call, how will you make the problem size smaller?*

To avoid infinite recursion, make sure that each recursive call brings you closer to the base case!

14. Implement a recursive function named `summation` that takes a parameter `n` and returns the sum $1 + 2 + \dots + n$. It should only have an `if` statement and two `return` statements (no loops).

```
def summation(n):  
    if n == 1:  
        return 1  
    else:  
        return n + summation(n - 1)
```

15. Enter your code into a Python Editor, and test the function. Make sure that `summation(100)` correctly returns 5050.

16. Implement a recursive function named `geometric` that takes three parameters (`a`, `r`, and `n`) and returns the sum " $a + ar + ar^2 + ar^3 \dots$ " where $n + 1$ is the total number of terms.

a) What is the base case?

`geometric(a, r, 0)` returns: `a`

b) What is the recursive case?

`geometric(a, r, n)` returns: `a * r ** n + geometric(a, r, n - 1)`

c) Write the function in Python:

```
def geometric(a, r, n):  
    if n == 0:  
        return a  
    else:  
        return a * r ** n + geometric(a, r, n - 1)
```

17. Enter your code into a Python Editor, and test the function. For example, if $a = 10$ and $r = 3$, the first five terms would be 10, 30, 90, 270, and 810. Make sure that `geometric(10, 3, 4)` correctly returns 1210 (the sum of those five terms).

(demonstrate to instructor)