1-2. Introduction to recursion

#_area/coding_test/leetcode/1_intro

Recursion is a problem solving method. In code, recursion is implemented using a function that calls itself.
재귀(Recursion)는 문제를 해결하는 방법 중 하나입니다. 코드에서 재귀는 함수가 자기 자신을 호출함으로써 구현됩니다.

The opposite of a recursive algorithm would be an iterative algorithm. There is a branch of study that proves that any iterative algorithm can be written recursively. While iterative algorithms use for loops and while loops to simulate repetition, recursive algorithms use function calls to simulate the same logic.
재귀 알고리즘의 반대 개념은 반복(Iterative) 알고리즘입니다. 계산 가능성 이론(Computability Theory)이라는 학문 분야에서는, 어떤 반복 알고리즘이든 재귀로도 표현할 수 있음을 증명합니다. 반복 알고리즘이 `for` 반복문이나 `while` 반복문 같은 구문을 사용해 반복을 시뮬레이션한다면, 재귀 알고리즘은 함수 호출을 통해 같은 로직을 시뮬레이션합니다.

Let's say that we wanted to print the numbers from 1 to 10. Here's some pseudocode for an iterative algorithm:
예를 들어, 1부터 10까지 숫자를 출력하고 싶다고 합시다. 다음은 반복 알고리즘의 의사 코드입니다:

```
for (int i = 1; i <= 10; i++) {
    print(i)
}
```

Here's some pseudocode for an equivalent recursive algorithm:
아래는 동등한 재귀 알고리즘의 의사 코드입니다:

```
function fn(i):
    print(i)
    fn(i + 1)
    return

fn(1)
```

Each call to `fn` first prints `i` (which starts at 1), and then calls `fn` again, but incrementing `i` (to print the next number).
각 `fn` 호출은 우선 `i`(처음에는 1)를 출력한 뒤, `fn(i + 1)`을 다시 호출하여 다음 수를 출력합니다.

> The first function call prints `1`, then calls `fn(2)`. In `fn(2)`, we print `2`, then call `fn(3)`, and so on.
> 첫 번째 함수 호출은 `1`을 출력한 뒤 `fn(2)`를 호출합니다. `fn(2)`는 `2`를 출력하고 다시 `fn(3)`을 호출하는 식으로 계속 진행됩니다.

However, this code is actually wrong. Do you see the problem? The function calls will never stop! Running this code would print natural numbers (positive integers) infinitely (or until the computer exploded). The `return` line never gets reached because `fn(i + 1)` comes before it.
하지만, 이 코드는 사실 잘못된 것입니다. 어떤 문제가 있을까요? 이 함수 호출은 절대 멈추지 않습니다! 이 코드를 실행하면 양의 정수를 무한히(혹은 컴퓨터가 폭발할 때까지) 출력하게 됩니다. `return` 문이 있긴 하지만, `fn(i + 1)` 호출이 먼저 실행되기 때문에 그 줄에 도달하지 못합니다.

We need what is called a **base case** to make the recursion stop. Base cases are conditions at the start of recursive functions that terminate the calls.
재귀를 멈추게 하려면, **베이스 케이스(base case)** 라고 불리는 조건이 필요합니다. 베이스 케이스는 재귀 함수를 시작할 때, 더 이상 재귀를 진행해서는 안 되는 상황을 정의합니다.

```
function fn(i):
    if i > 10:
        return

    print(i)
    fn(i + 1)
    return

fn(1)
```

> After we call `fn(10)`, we print `10` and call `fn(11)`. In the `fn(11)` call, we trigger the base case and return. So now we are back in the call to `fn(10)` and move to the next line, which is the return statement. This makes us return back to the `fn(9)` call and so on, until we eventually

return from the `fn(1)` call and the algorithm terminates.
이 코드에서는 `fn(10)`을 호출하고 `10`을 출력한 뒤 `fn(11)`을 호출합니다. `fn(11)` 호출에서 베이스 케이스(`i > 10`)가 트리거되어 함수가 반환됩니다. 그러면 우리는 `fn(10)` 호출로 다시 돌아오고, 이제 4번째 줄(`fn(i + 1)`)이 끝났으니 다음 줄(마지막 `return`)로 이동해 해당 함수를 종료합니다. 이렇게 점차 "되돌아가면서" 결국 `fn(1)` 호출까지 모든 호출이 반환되고, 알고리즘은 종료됩니다.

An important thing to understand about recursion is the order in which the code runs – the order in which the computer executes instructions. With an iterative program, it's easy – start at the top, and go line by line. With recursion, it can get confusing because calls can cascade on top of each other. Let's print numbers again, but this time only up to 3. Let's also add another print statement and number the lines:
재귀에서 중요한 점은 코드가 실행되는 **순서**를 이해하는 것입니다. 반복 기반 프로그램에서는 코드가 위에서부터 시작해 아래로 순차적으로 실행되기 때문에 이해하기 쉽습니다. 하지만 재귀의 경우, 함수 호출이 겹겹이 쌓이기 때문에 헷갈릴 수 있습니다. 1부터 3까지 숫자를 출력하되, 추가로 다른 `print` 문을 하나 더 넣고 줄에 번호를 매겨 어떤 순서로 실행되는지 살펴봅시다.

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return

fn(1)
```

If you ran this code, you would see the following in the output:
이 코드를 실행하면 아래와 같은 결과가 나옵니다:

```
// 1
// 2
// 3
// End of call where i = 3
// End of call where i = 2
// End of call where i = 1
```

As you can see, the line where we print text is executed in reverse order. The original call `fn(1)` first prints `1`, then calls `fn(2)`, which prints `2`, then calls `fn(3)`, which prints `3`, then calls `fn(4)`. **Now, this is the important part**: how recursion "moves" back "up." `fn(4)` triggers the base case, which returns. We are now back in the function call where `i = 3` and **line 4** has finished, so we move to the **line 5** which prints `"End of call where i = 3"`. Once that line runs, we move to the next line, which is a `return`. Now, we are back in the function call where `i = 2` and **line 4** line has finished, so again we move to the next line and print `"End of the call where i = 2"`. This repeats until the original function call to `fn(1)` returns.
보시다시피, 각 함수 호출에서 마지막에 출력하는 부분은 역순으로 실행됩니다. `fn(1)`을 처음 호출하면, 먼저 `1`을 출력하고 `fn(2)`를 호출합니다. `fn(2)`는 `2`를 출력하고 `fn(3)`을 호출합니다. `fn(3)`은 `3`을 출력하고 `fn(4)`를 호출합니다.
여기서 중요한 점은 재귀가 어떻게 "위로 올라가느냐"입니다. `fn(4)`에서 베이스 케이스가 발동되어 반환되면, 우리는 `i = 3`이었던 함수 호출로 돌아옵니다. 이제 4번째 줄(`fn(i + 1)`)이 끝났으니 5번째 줄로 이동하여 `"End of call where i = 3"`을 출력합니다. 이후 다음 줄의 `return`을 만나면 `i = 3` 호출이 종료되고, 다시 `i = 2` 호출로 돌아가 같은 과정을 반복합니다. 이런 식으로 `fn(1)`을 호출한 함수까지 모두 반환을 마치면 전체 실행이 종료됩니다.

> Every function call "exists" until it returns. When we move to a different function call, the old one waits until the new one returns. The order in which the calls happen is remembered, and the lines within the functions are executed in order.
> Note that each function call also has its own local scope. So in the example above, when we call `f(3)`, there are 3 "versions" of `i` simultaneously. The first call has `i = 1`, the second call has `i = 2`, and the third call has `i = 3`. Let's say that we were to do `i += 1` in the `f(3)` call. Then `i` becomes `4`, but **only** in the `f(3)` call. The other 2 "versions" of `i` are unaffected because they are in different scopes.
> 모든 함수 호출은 "반환"될 때까지 "존재"합니다. 다른 함수 호출로 이동하면 이전 함수 호출은 잠시 대기하고, 새로 이동한 호출이 반환되어야 다시 돌아와서 남은 코드를 실행합니다. 함수가 호출된 순서는 기억되며, 함수 내 코드는 순서대로 실행됩니다.
> 또한 각 함수 호출에는 자체적인 로컬 스코프가 있습니다. 예를 들어, 위 사례에서 `f(3)`를 호출했을 때, 실제로는 `i = 1`인 호출, `i = 2`인 호출, `i = 3`인 호출 이렇게 3개가 동시에 "존재"하고 있습니다. 만약 `f(3)` 안에서 `i += 1`을 했다면, `i`는 그 호출 스코프에서만 `4`가 되며, 다른 두 호출(`i = 1`, `i = 2`)의 `i`에는 영향을 주지 않습니다. 왜냐하면 각각 독립적인 로컬 스코프를 가지기 때문입니다.

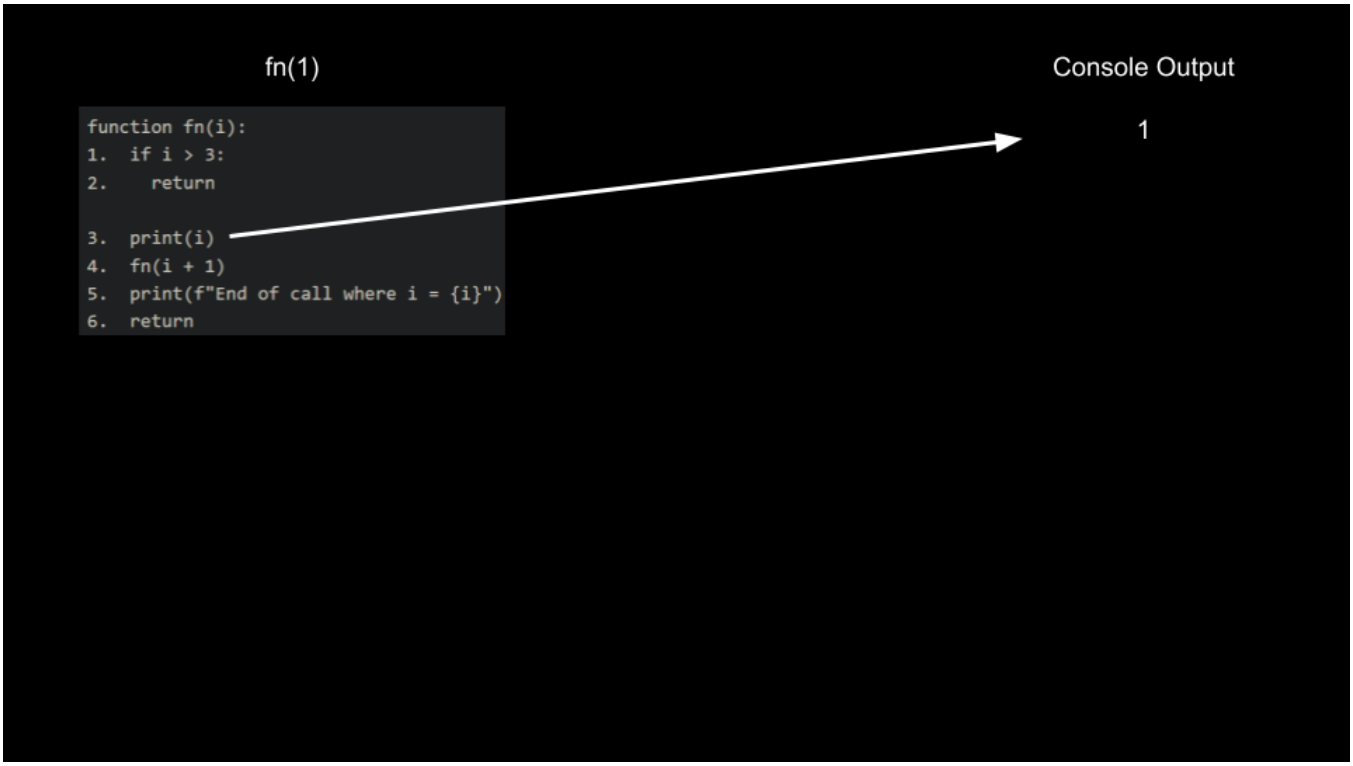
Please see the following slideshow for a visual representation of this example.

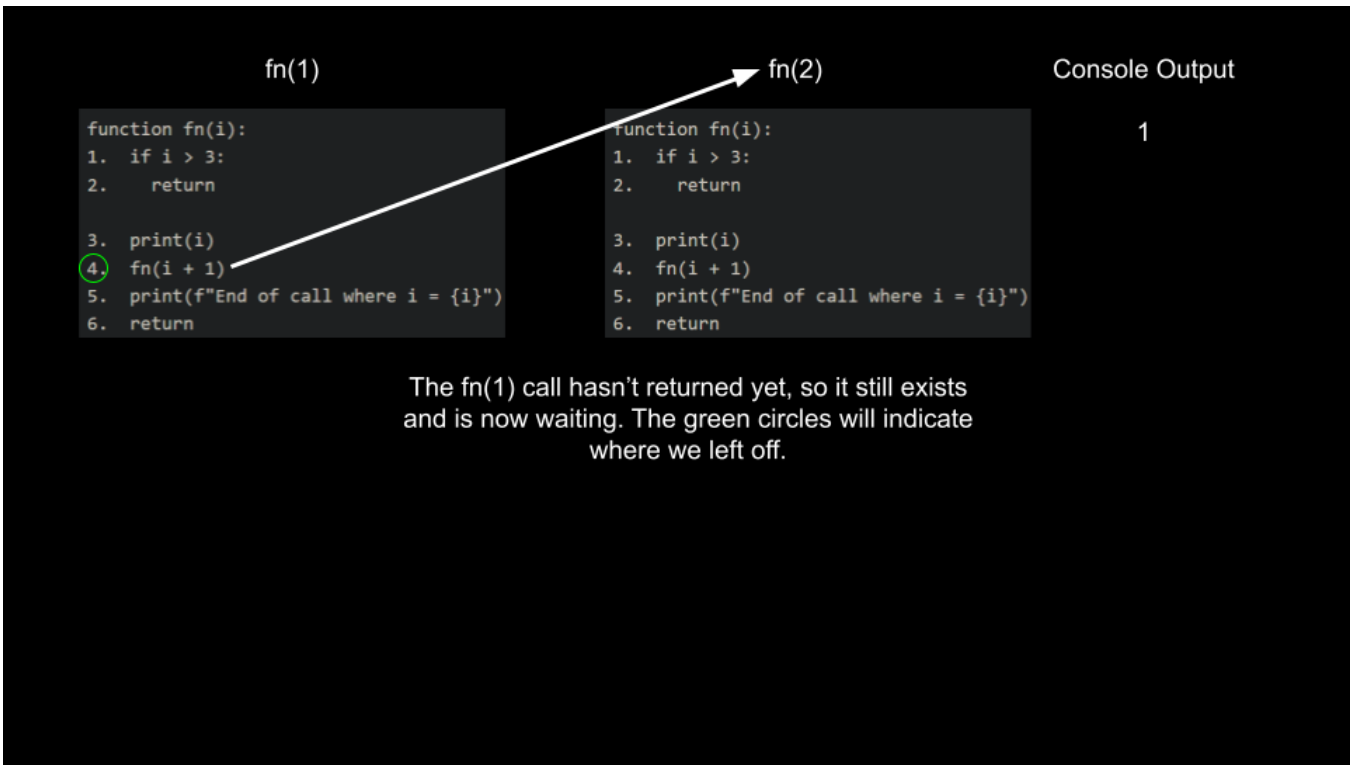
Start by calling fn(1)

fn(1)

Console Output

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```





The fn(1) call hasn't returned yet, so it still exists and is now waiting. The green circles will indicate where we left off.

fn(1)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

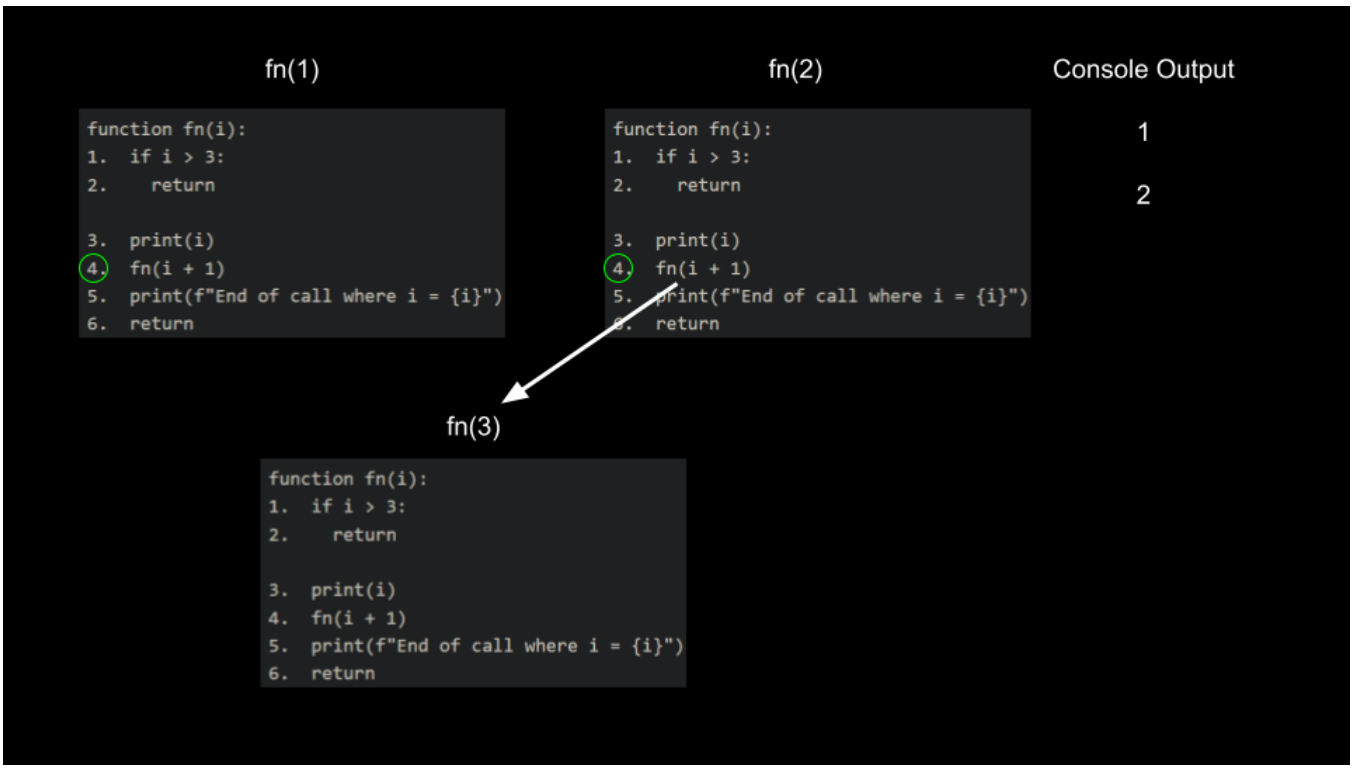
fn(2)

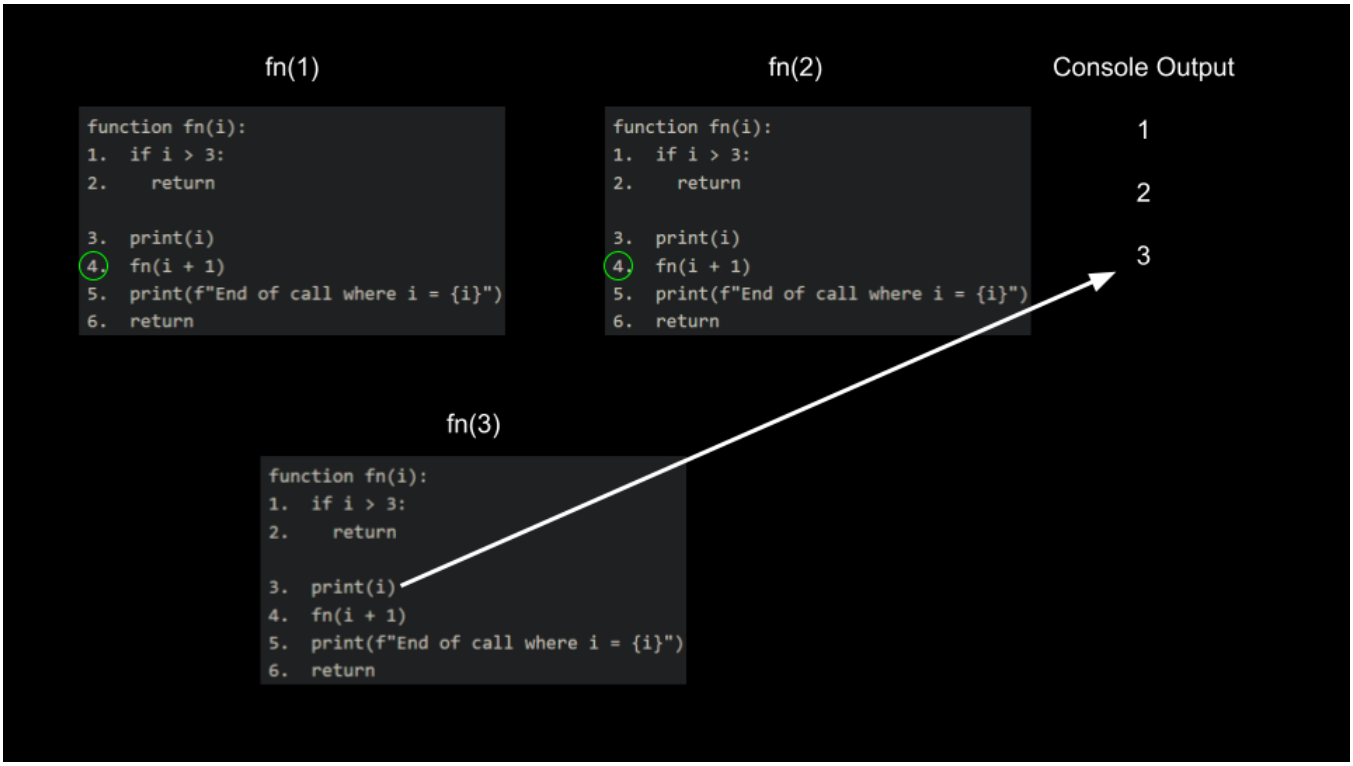
```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

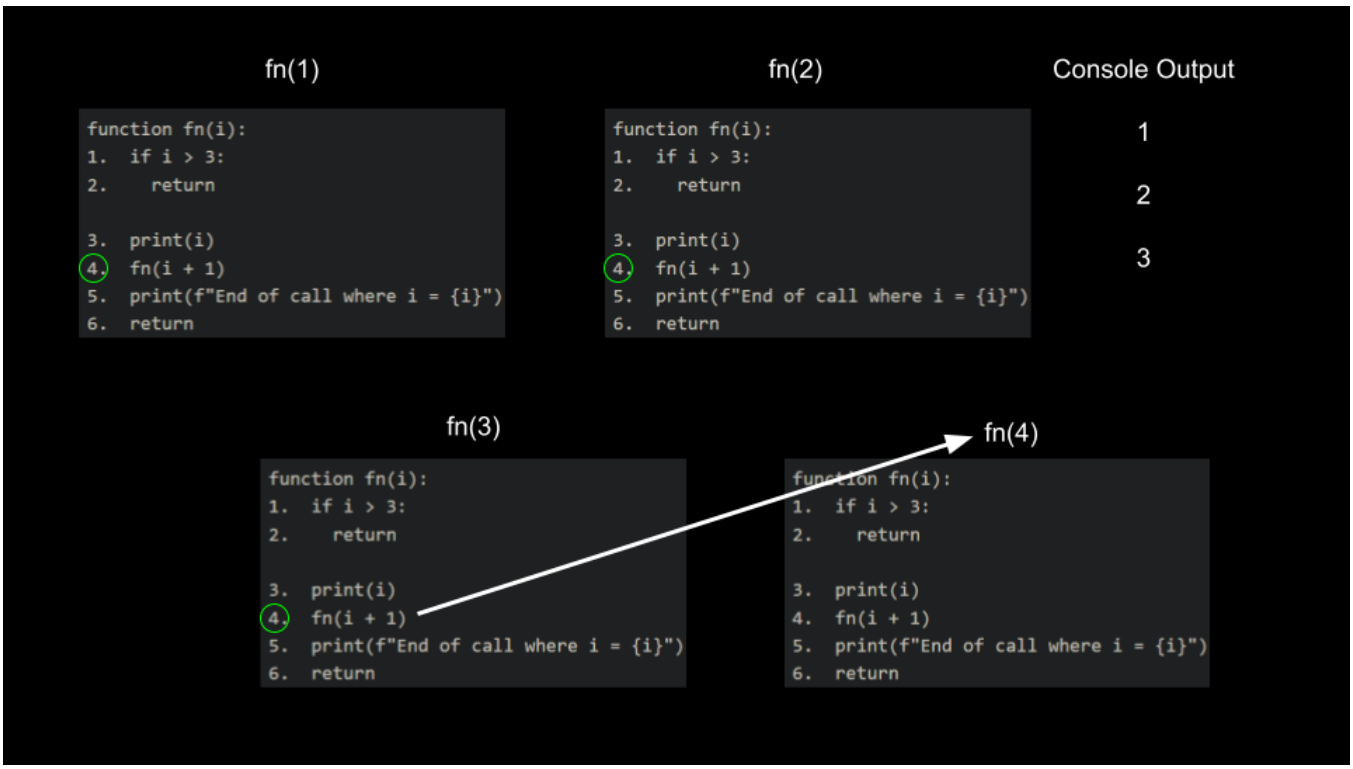
Console Output

1

2




fn(1)
fn(2)
Console Output
function fn(i):
1. if i > 3:
2. return
3. print(i)
4. fn(i + 1)
5. print(f"End of call where i = {i}")
6. return
1
2
3
fn(3)
function fn(i):
1. if i > 3:
2. return
3. print(i)
4. fn(i + 1)
5. print(f"End of call where i = {i}")
6. return


fn(1)
fn(2)
Console Output
function fn(i):
1. if i > 3:
2. return
3. print(i)
4. fn(i + 1)
5. print(f"End of call where i = {i}")
6. return
1
2
3
fn(3)
fn(4)
function fn(i):
1. if i > 3:
2. return
3. print(i)
4. fn(i + 1)
5. print(f"End of call where i = {i}")
6. return

fn(1)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

fn(2)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

Console Output

1

2

3

Base case hit, return to where we called f(4) from

fn(3)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

fn(4)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

fn(1)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

fn(2)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

Console Output

1

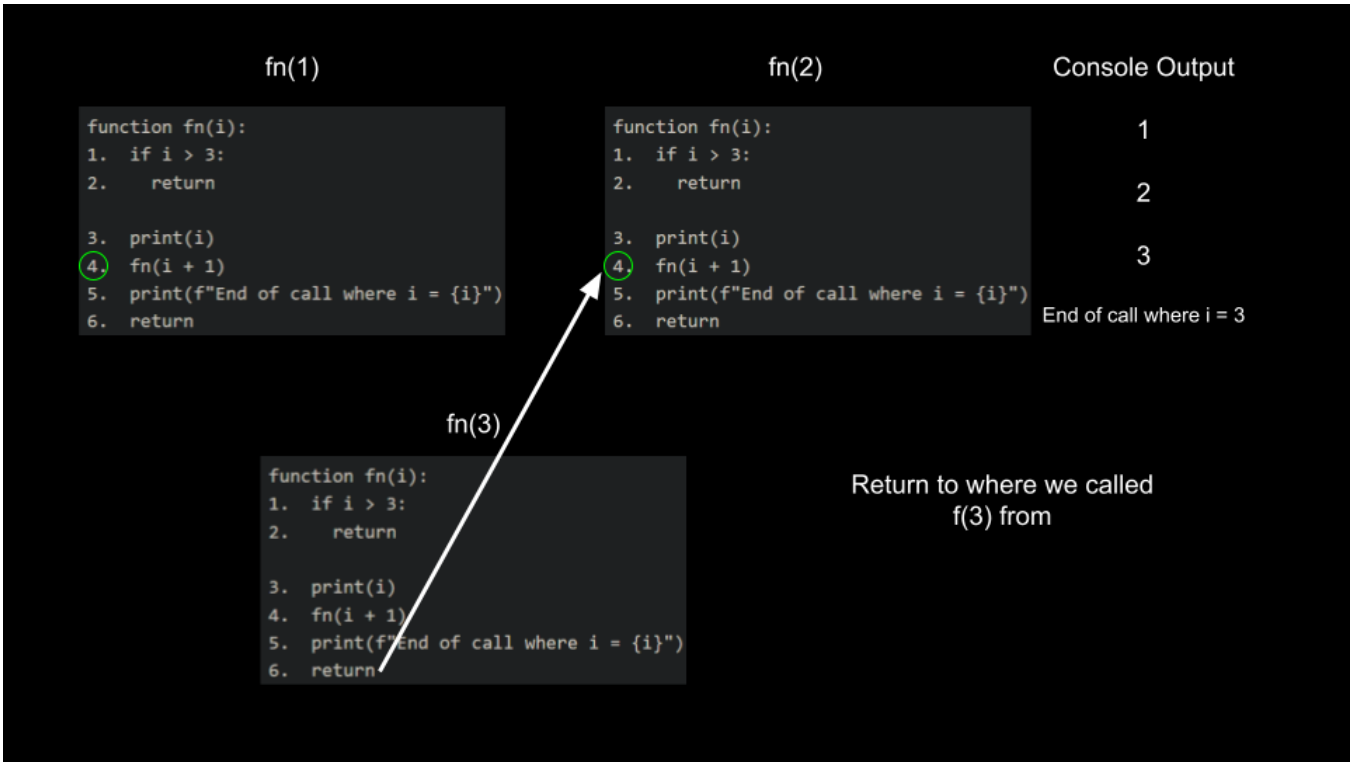
2

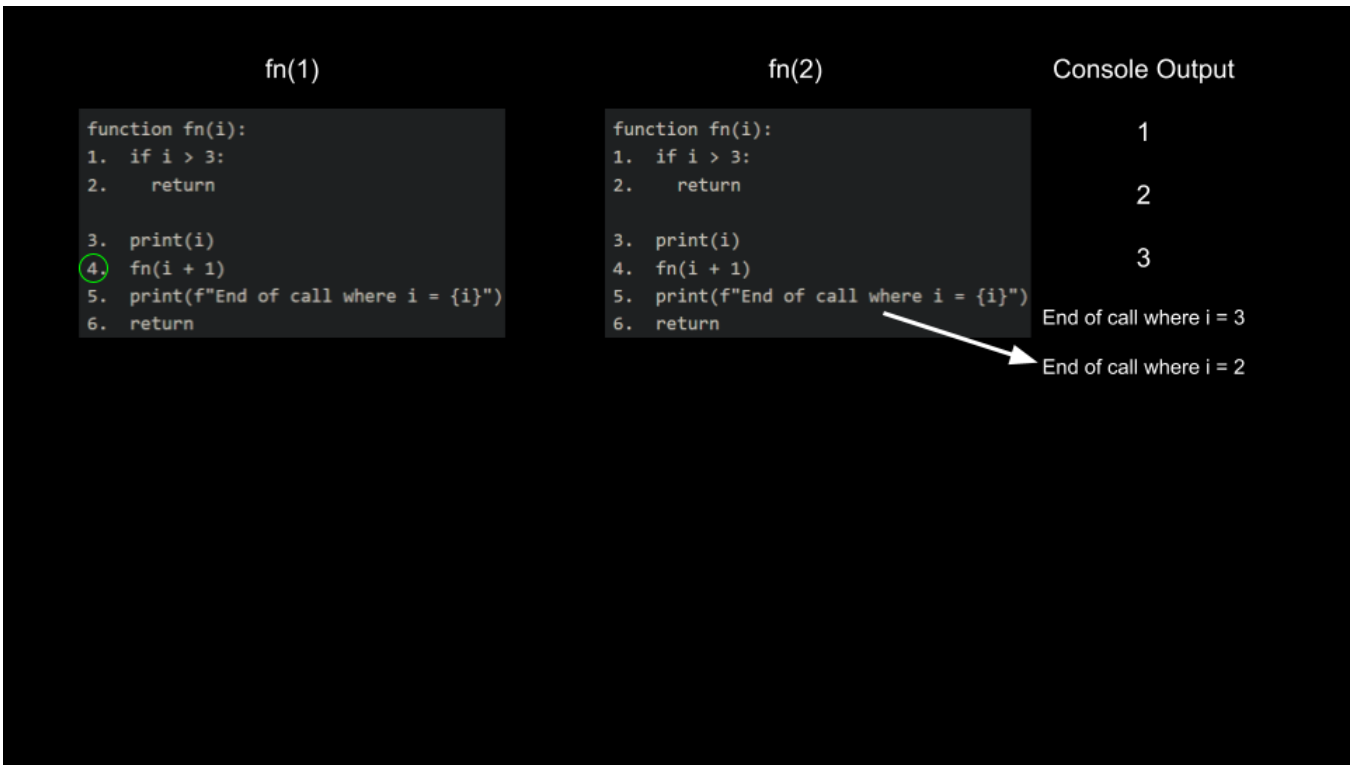
3

End of call where i = 3

fn(3)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```


fn(1)
fn(2)
Console Output
function fn(i):
1. if i > 3:
2. return
3. print(i)
4. fn(i + 1)
5. print(f"End of call where i = {i}")
6. return
1
2
3
End of call where i = 3
fn(3)
Return to where we called f(3) from


fn(1)
fn(2)
Console Output
function fn(i):
1. if i > 3:
2. return
3. print(i)
4. fn(i + 1)
5. print(f"End of call where i = {i}")
6. return
1
2
3
End of call where i = 3
End of call where i = 2

fn(1)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

fn(2)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

Console Output

1

2

3

End of call where i = 3

End of call where i = 2

Return to where we called
f(2) from

fn(1)

```
function fn(i):
1.  if i > 3:
2.    return

3.  print(i)
4.  fn(i + 1)
5.  print(f"End of call where i = {i}")
6.  return
```

Console Output

1

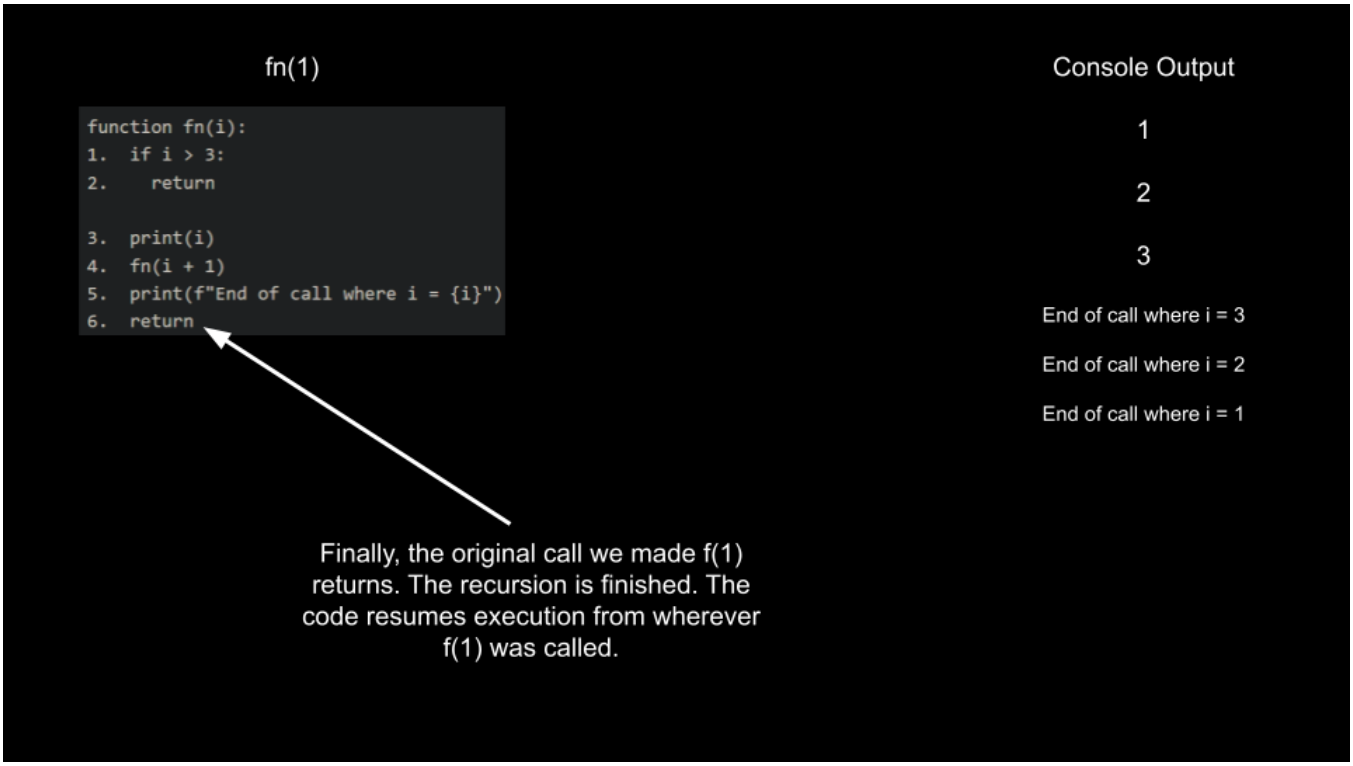
2

3

End of call where i = 3

End of call where i = 2

End of call where i = 1



Breaking problems down

This printing example is pretty pointless – it's easier to use a for loop if you just want to print numbers. Where recursion shines is when you use it to break down a problem into "subproblems," whose solutions can then be combined to solve the original problem.
위의 예시(숫자를 출력하는 것)는 사실 별로 쓸모가 없습니다. 단지 숫자를 출력하기만 한다면 `for` 반복문을 쓰는 것이 훨씬 간편하기 때문입니다. 재귀가 진가를 발휘하는 것은, 문제를 "부분 문제(subproblem)"로 나누고, 각 부분 문제를 풀어서 결합하여 원래 문제를 해결하고자 할 때입니다.

Let's look at the Fibonacci numbers. The Fibonacci numbers are a sequence of numbers starting with `0, 1`. Then, each number is defined as the sum of the previous two numbers. The first few Fibonacci numbers are `0, 1, 1, 2, 3, 5, 8`. More formally, we have:
피보나치 수(Fibonacci numbers)를 예로 들어봅시다. 피보나치 수는 `0, 1`로 시작하는 수열로, 각 수는 바로 앞 두 개의 수를 더한 값으로 정의됩니다. 초기 몇 개는 `0, 1, 1, 2, 3, 5, 8`과 같습니다. 좀 더 정확히 말하면,

$$F_n = F_{n-1} + F_{n-2}$$

This is called a **recurrence relation** – it's an equation that connects the terms together.

라는 **점화식(Recurrence Relation)** 을 갖습니다. 이는 항들끼리 서로 연결되는 수식을 의미합니다.

Let's use pseudocode to write a function `F(n)` that returns the n^{th} Fibonacci number (0 indexed). Don't forget we need base cases with any recursive function. In this case the base cases are explicitly defined: `F(0) = 0` and `F(1) = 1`.
이제, `n`번째(0부터 시작) 피보나치 수를 반환하는 함수 `F(n)`을 의사 코드로 작성해 봅시다. 재귀 함수를 작성할 때는 항상 베이스 케이스가 필요합니다. 이 경우엔 베이스 케이스가 명시적으로 정의되어 있습니다: `F(0) = 0`과 `F(1) = 1`입니다.

```
function F(n):
    if n <= 1:
        return n

    oneBack = F(n - 1)
    twoBack = F(n - 2)
    return oneBack + twoBack
```

Let's say that we wanted to find `F(3)`. Upon calling `F(3)`, we would see the following flow, with each indentation level representing a function call's scope:
예를 들어 `F(3)`을 구하고 싶다고 하면, 다음과 같은 흐름이 진행됩니다. 각 들여쓰기 단계는 함수 호출 스코프를 나타냅니다:

```
oneBack = F(2)
    oneBack = F(1)
        F(1) = 1
    twoBack = F(0)
        F(0) = 0
    F(2) = oneBack + twoBack = 1
twoBack = F(1)
    F(1) = 1
F(3) = oneBack + twoBack = 2
```

As you can see, we took the original problem `F(3)`, and broke it down into two smaller subproblems – `F(2)` and `F(1)`. By combining the recurrence relation and base cases, we can solve the subproblems and use those solutions to solve the original problem.

보시다시피, 원래 문제 `F(3)`을 더 작은 부분 문제인 `F(2)`와 `F(1)`로 나누었습니다. 점화식과 베이스 케이스를 결합해 부분 문제를 해결하고, 그 결과를 합쳐서 원래 문제의 해를 구할 수 있습니다.

This is the most common use of recursion – you have your recursive function **return the answer to the problem you're trying to solve for a given input**. In this example, the problem we're trying to solve for a given input is "What is the n^{th} Fibonacci number?" As such, we designed our function to return a Fibonacci number, according to the input `n`. By determining the base cases and a recurrence relation, we can easily implement the function.
이처럼 재귀를 가장 자주 사용하는 이유는, **주어진 입력에 대해 풀고자 하는 문제의 답을 반환하는** 재귀 함수를 작성하기 위함입니다. 여기서는 "`n`번째 피보나치 수는 무엇인가?"가 문제이므로, 그에 맞추어 함수를 작성했습니다. 베이스 케이스와 점화식을 설정해 두면 구현이 쉽습니다.

By following this idea, solving the subproblems is easy – if we wanted the 100th Fibonacci number, we know by definition that it is the sum of the 99th and 98th Fibonacci number. On the function call to `F(100)`, we know that calling `F(99)` and `F(98)` will give us those numbers.
이 아이디어에 따르면, 부분 문제를 해결하는 것은 간단합니다. 예를 들어 100번째 피보나치 수를 알고 싶다고 합시다. 정의상 그것은 99번째와 98번째 피보나치 수의 합이므로, `F(100)`을 호출하면 `F(99)`와 `F(98)`을 구해 그 합을 반환하면 됩니다.