1 The Fibonacci sequence

Return the n-th number in the Fibonacci sequence. The first two numbers in the Fibonacci sequence are equal to 1; any other number is equal to the sum of the preceding two numbers.

Example: for n = 6, the first 6 numbers of the sequence are `[1, 1, 2, 3, 5, 8]` so the result is 8.

Solution 1: brute force, $O(2^n)$ time

A straightforward solution is to implement the function recursively:

```
def fibonacci(n):
    if n <= 2:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

The above code is correct but too slow due to redundancies. We can see this if we add logging to the function:

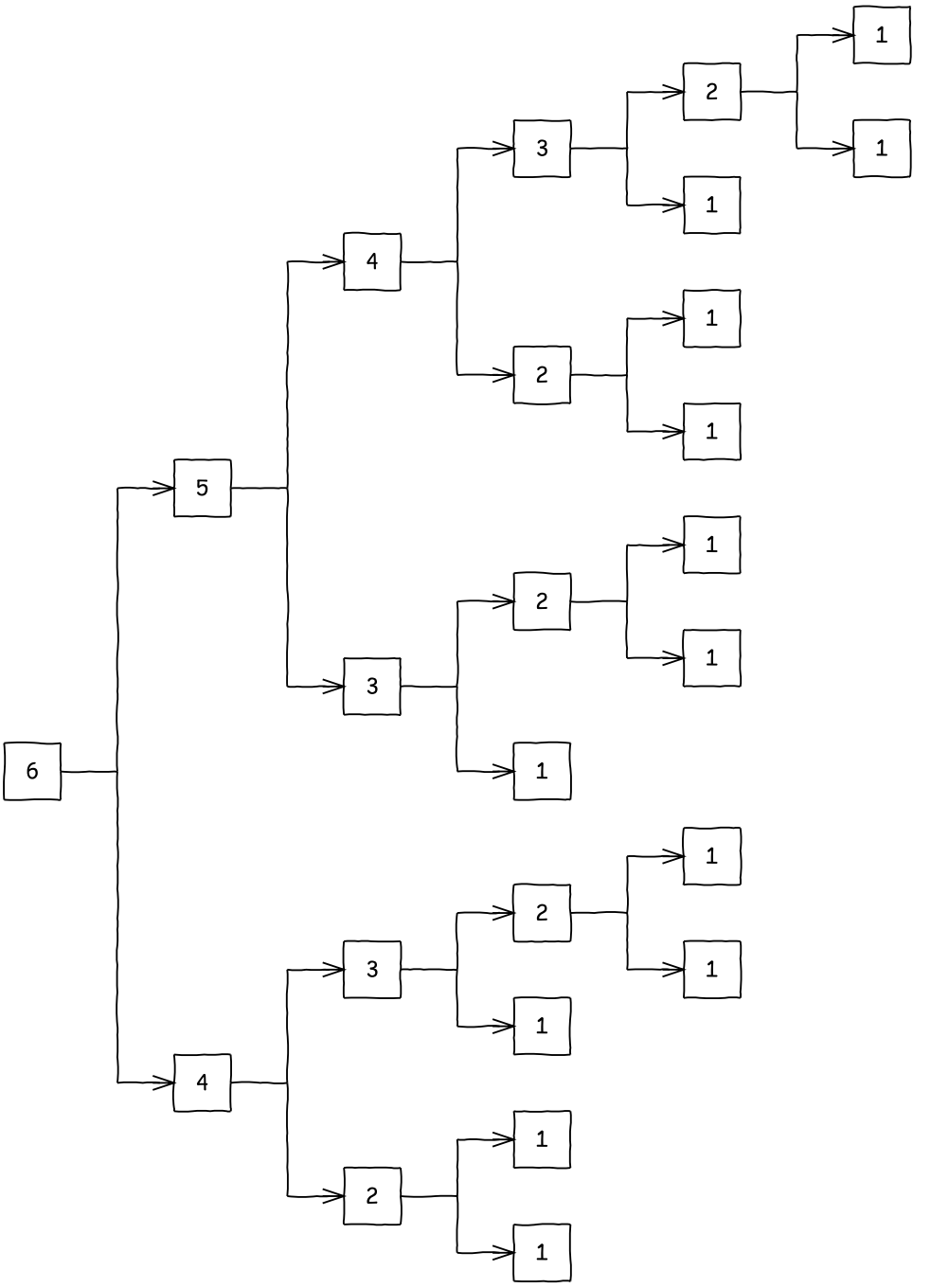
```
import inspect
def stack_depth():
    return len(inspect.getouterframes(inspect.currentframe())) - 1

def fibonacci(n):
    print("{indent}fibonacci({n}) called".format(
                indent="  " * stack_depth(), n=n))
    if n <= 2:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

fibonacci(6)
```

We changed the code to print the argument passed to the `fibonacci` function. The message is indented by the call stack depth, so that we can see better which function call is causing which subsequent calls. Running the above code prints:

```
fibonacci(6) called
  fibonacci(5) called
    fibonacci(4) called
      fibonacci(3) called
        fibonacci(2) called
        fibonacci(1) called
      fibonacci(2) called
    fibonacci(3) called
      fibonacci(2) called
      fibonacci(1) called
  fibonacci(4) called
```

```
fibonacci(3) called
  fibonacci(2) called
  fibonacci(1) called
fibonacci(2) called
```

That's a lot of calls! If we draw the call graph, we can see that it's an almost full binary tree:



Notice that the height of the binary tree is `n` (in this case, 6). The tree is almost full, thus it has $O(2^n)$ nodes. Since each node represends a call of our function, our algorithm has exponential complexity.

Solution 2: dynamic programming, top-down

We can optimize the previous solution by avoiding redundant computations. These redundancies are visible in the call graph:

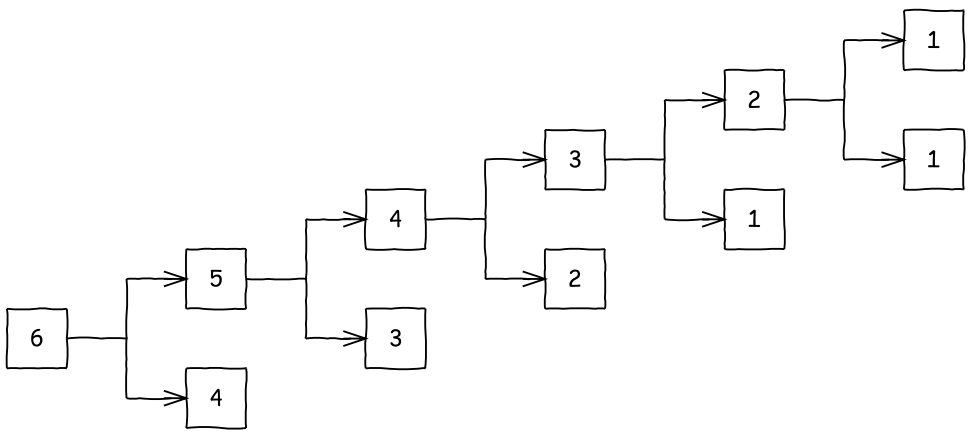
- `fibonacci(4)` is called twice, once by `fibonacci(5)` and once by `fibonacci(6)`;
- `fibonacci(3)` is called 3 times;
- `fibonacci(2)` is called 5 times;
- `fibonacci(1)` is called 3 times.

It does not make sense to compute, for example, the 4-th Fibonacci number twice, since it does not change. We should compute it only once and cache the result.

Let's use a dictionary to store the cache:

```
fibonacci_cache = {}
def fibonacci(n):
    if n <= 2:
        return 1
    if n not in fibonacci_cache:
        fibonacci_cache[n] = fibonacci(n - 1) + fibonacci(n - 2)
    return fibonacci_cache[n]
```

The call graph of the optimized code looks like this:



Notice how only the first call to `fibonacci(n)` recurses. All subsequent calls return from the cache the value that was previously computed.

This implementation has $O(n)$ time complexity, since exactly one function call is needed to compute each number in the series.

This strategy of caching the results of subproblems is called *dynamic programming*.

While the above code is correct, there are some code style issues:

- We introduced the global variable `fibonacci_cache`; it would be great if we could avoid global variables, since they impact code readability;
- The code is more complicated than before due to the cache manipulations.

We can avoid adding the global variable by using instead an attribute called `cache` that is attached to the function:

```
def fibonacci(n):
    if n <= 2:
        return 1
    if not hasattr(fibonacci, 'cache'):
        fibonacci.cache = {}
    if n not in fibonacci.cache:
        fibonacci.cache[n] = fibonacci(n - 1) + fibonacci(n - 2)
    return fibonacci.cache[n]
```

The advantage is that the cache variable is now owned by the function, so no external code is needed anymore to initialize it. The disadvantage is that the code has become even more complicated, thus harder to read and modify.

A better approach is to keep the original function simple, and wrap it with a decorator that performs the caching:

```
def cached(f):
    cache = {}
    def worker(*args):
        if args not in cache:
            cache[args] = f(*args)
        return cache[args]
    return worker

@cached
def fibonacci(n):
    if n <= 2:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

The good news is that Python 3 has built-in support for caching decorators, so there is no need to roll your own:

```
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci(n):
    if n <= 2:
```

```
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

By default `lru_cache` is limited to 128 entries, with least-recently used entries evicted when the size limit is hit. Passing `maxsize=None` to `lru_cache` ensures that there is no memory limit and all values are cached. In practice, it might be preferrable to set a limit instead of letting memory usage increase without bounds.

> **Prefer standard library functionality to rolling your own**
>
> Using the standard `lru_cache` decorator makes the code easier to read, since it has well-known behavior. The advange over using a custom caching method is that the reader does not need to spend time to understand its details.

Solution 2: dynamic programming, bottom-up

While computing the Fibonacci sequence recursively is useful for pedagogical reasons, it is more intuitive to compute it iteratively starting from the smaller numbers, just like a human would do:

```
def fibonacci(n):
    series = [1, 1]
    while len(series) < n:
        series.append(series[-1] + series[-2])
    return series[-1]
```

The code has $O(n)$ time complexity, as well as $O(n)$ space complexity. In practice the performance is better than the recursive implementation, since there is no overhead due to extra function calls.

The space complexity can be reduced to $O(1)$ if we notice that we do not need to store the entire sequence, just the last two numbers:

```
def fibonacci(n):
    previous = 1
    current = 1
    for i in range(n - 2):
        next = current + previous
        previous, current = current, next
    return current
```

We have written an algorithm that starts from the smallest subproblem (the first two numbers in the sequence), then expands the solution to reach the original problem (the n-th number in the sequence). This approach is called *bottom-up* dynamic programming. By contrast, the previous approach of solving the problem recursively starting from the top is called *top-down* dynamic programming. Both approaches are equally valid; one or the other may be more intuitive, depending on the problem.

In the rest of the book, we will look at how we can apply dynamic programming to solving non-trivial problems. In general, we will show both top-down and bottom-up solutions. We will see that the top-down approach is often easier to understand and implement, however it offers less optimization opportunities compared to bottom-up.