CSC148 Week 12

Larry Zhang

Announcements

No lecture this Friday

Last lecture next Monday: exam review

Exam Jam: April 5th, 4 PM - 5:30 PM, in IB-120

Fibonacci Numbers

Fibonacci Sequence / Number

A sequence of numbers where every number after the first two is the sum of the two preceding ones.

The 0th and 1st numbers are 0 and 1 (base cases).

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ….

The n-th number in the Fibonacci sequence is called the n-th Fibonacci number.

e.g., **fib(n)**

We want to write a program to, given n, compute **fib(n)**

Easy, recursive solution!

```
1 def fib(n: int) -> int:
2     """Return the nth fibonacci number
3     """
4     if n <= 1:
5         return n
6     else:
7         return fib(n - 1) + fib(n - 2)
```

DEMO:

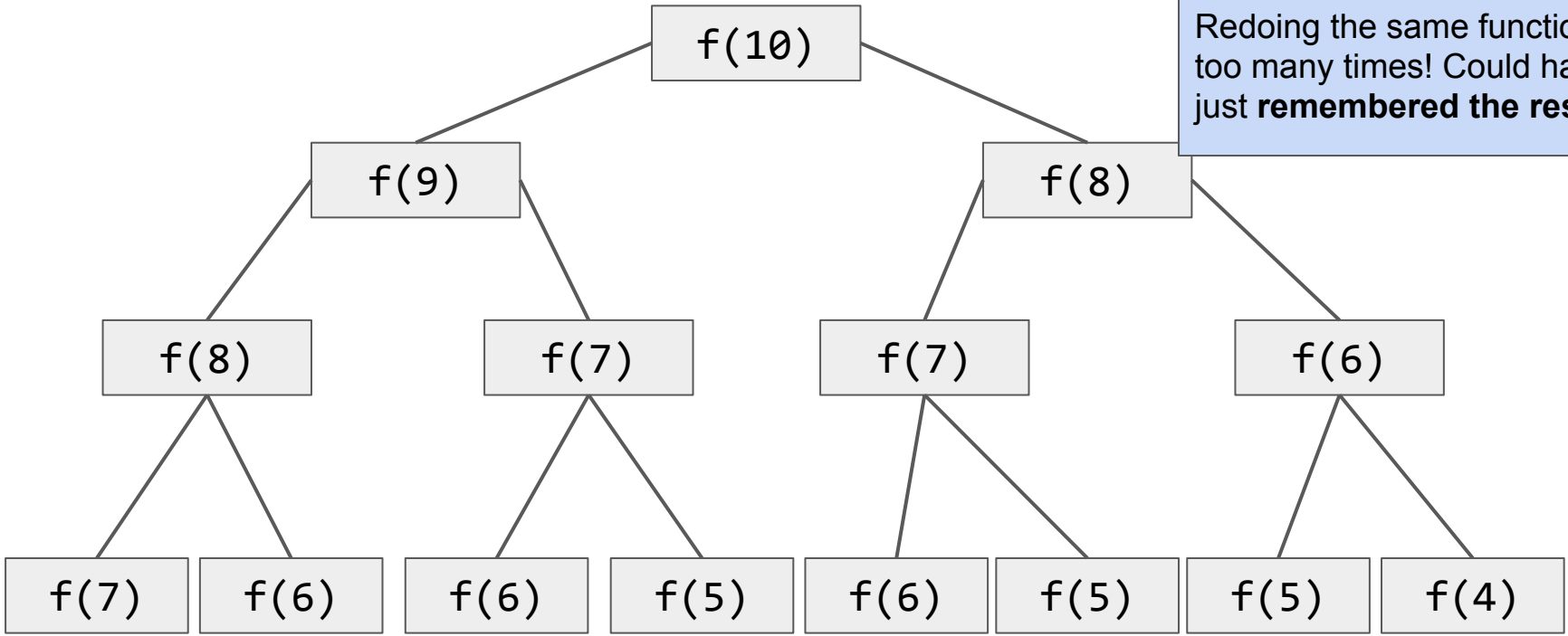
fib.py

fib_count.py

Why so many function calls?

Are we wasting time somewhere?

Redoing the same function call too many times! Could have just **remembered the result**!



Memoization

A general technique to avoid redoing the same work over and over again.

Store the results of recursive calls (e.g., in a dictionary) as are being computed.

When making a function call, first look up the result in the dictionary rather than setting off a chain of recursive calls again.

fib(n) with memoization

```
 1 def fib_mem(n: int) -> int:
 2     """Return the nth fibonacci number
 3     """
 4     known = {}
 5     def fib_helper(m: int) -> int:
 6         if m not in known:
 7             if m < 2:
 8                 known[m] = m
 9             else:
10                 known[m] = fib_helper(m - 1) + fib_helper(m - 2)
11         return known[m]
12     return fib_helper(n)
```

DEMO: fib_mem.py

- Try fib(100), it's much faster now.
- Try fib(1000) …
- Fix #1: change maximum recursion depth in Python configuration.
  - this does not fix the problem, it just delays it.
- Recursion is risky when we need to recurse for many levels because of possible stack overflow.
- We can use an iterative algorithm that avoid recursion entirely.
- Strategy:
  - start from the base case, for which we already know the answer
  - solve the small problems first, store the results
  - when we solve larger problems, the smaller results they need are ready
- This strategy is called (bottom-up) **dynamic programing**.
  - Useful when recursion causes the same subproblems to be solved repeatedly

Dynamic Programming Fib

```python
def fib_dyn(n: int) -> int:
    """Return the nth fibonacci number
    """
    known = {0: 0, 1: 1}
    for m in range(2, n + 1):
        known[m] = known[m - 1] + known[m - 2]
    return known[n]
```

fib(1000) not a problem anymore.

Anything else to optimize?

For Fib(1000), we don't need to remember all of Fib(0), Fib(1)...Fib(999)

Simple Loop Fib

```
 1 def fib_loop(n):
 2     """Return the nth fibonacci number
 3     """
 4     if n <= 1:
 5         return n
 6     a = 0
 7     b = 1
 8     c = 1
 9     for i in range(3, n+1):
10         a = b
11         b = c
12         c = a + b
13     return c
```

Only using 4 variables besides n (a, b, c, i)

```
 1 def fib_loop2(n):
 2     """Return the nth
 3        fibonacci number
 4     """
 5     a = 0
 6     b = 1
 7     while n != 0:
 8         a = a + b
 9         b = a - b
10         n = n - 1
11     return a
```

Clever: only using 2 variables besides n (a, b)

Can we optimize even more?

Right now, the runtime for Fib we have is ...

O(n)

There is a way to do it in **O(log n)** time.

It will require some 2nd-year CSC and MAT courses to fully understand (involving optimized matrix chain multiplication)

In fact, there is also a way to get the n-th Fib number in **O(1)** time …, because there is a closed form for it: https://en.wikipedia.org/wiki/Fibonacci_number

Summary

When is it risky to use recursion?

- When the parameter value can be very large (stack overflow)
- When recursive calls solve the same problems over and over

It’s generally advised to avoid recursion in these cases

Sometimes iterations work better!