Addendum to Ron's Talk on Performance - Memoizaition

Breitzman 8/7/2018

Edited From Wikipedia

- In computing, **memoization** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again. Although related to caching, memoization refers to a specific case of this optimization, distinguishing it from forms of caching such as buffering or page replacement.

- The term "memoization" was coined by Donald Michie in 1968 and is derived from the Latin word "memorandum" ("to be remembered"), usually truncated as "memo" in American English, and thus carries the meaning of "turning [the results of] a function into something to be remembered." While "memoization" might be confused with "memorization" (because they are etymological cognates), "memoization" has a specialized meaning in computing.

- While memoization may be added to functions internally and explicitly by a computer programmer, referentially transparent functions may also be automatically memoized externally. The techniques employed by Peter Norvig have application not only in Common Lisp (the language in which his paper demonstrated automatic memoization), but also in various other programming languages. Applications of automatic memoization have also been formally explored in the study of term rewriting and artificial intelligence.

Recall From Ron's Talk

- When he did Fibonacci in a loop it was much faster than the recursive version
- However it is often the case that the recursive version of an algorithm is much easier to program and understand
  - In this case for example the recursive version literally uses the definition of the Fibonacci series, while the loop version takes a bit of thought
  - Similarly programming a binary tree traversal without recursion is a huge headache
- Memoization allows us to keep the easy-to-understand recursive algorithm while making it really fast.

Ron's Fibo Code

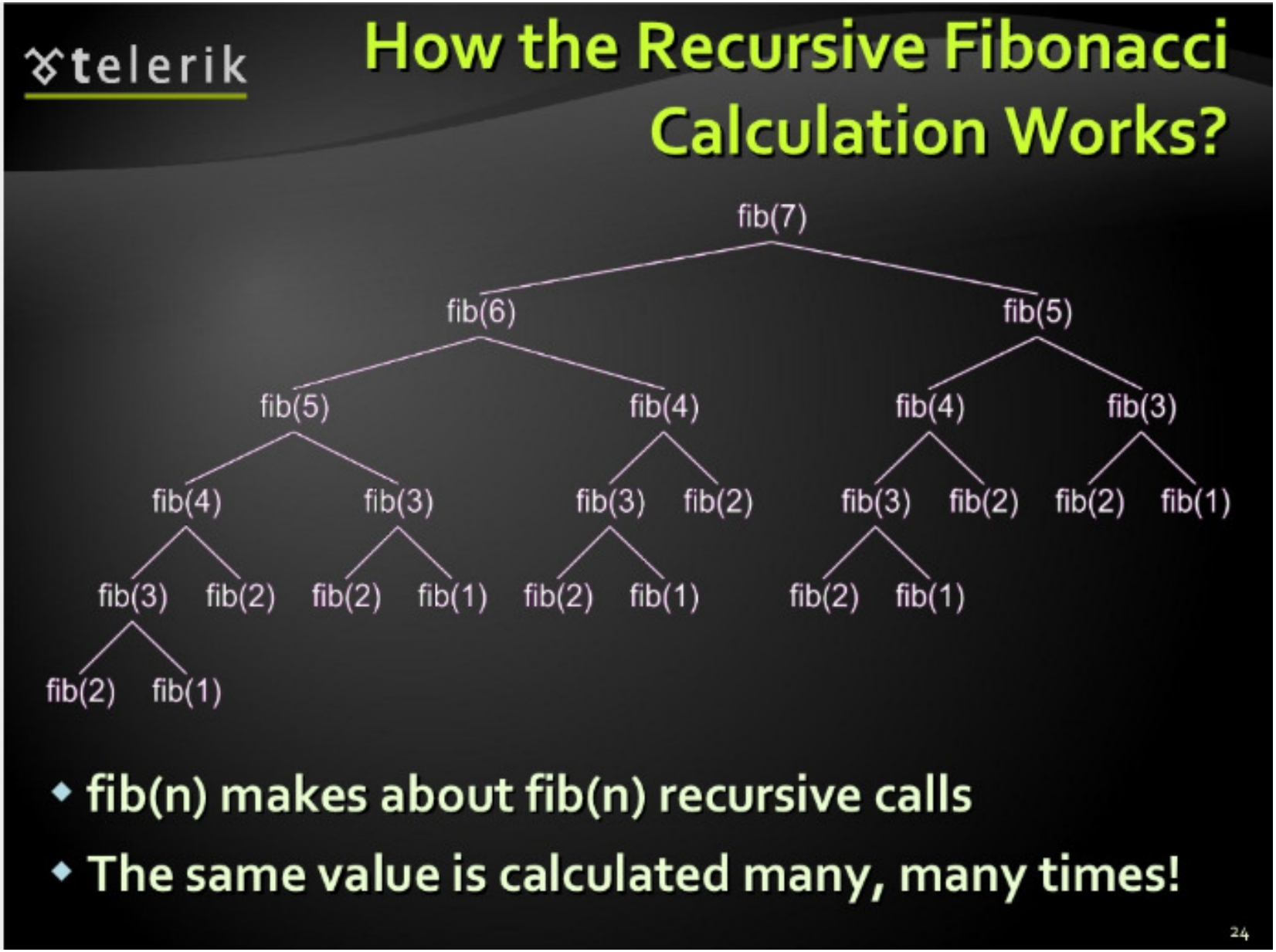
```
# calculate next digit in the fibonacci sequence using a loop
def fibo_loop(n):
    a = 0
    b = 1
    for _ in range(n):
        temp = a
        a = b
        b = temp + a
    return a

# calculate next digit in the fibonacci sequence using recursion
def fibo_recursive(n):
    if n > 1:
        return fibo_recursive(n-1) + fibo_recursive(n-2)
    else:
        return n
```

- Loop version very slick but not as easy to understand as the recursive version

Why the recursive version is slow

(from https://www.slideshare.net/introprogramming/10-recursion)



Memoizing in Python

```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper
```

- This single function can be used to memoize any function in Python
- Example next page

In [1]:
```
# calculate next digit in the fibonacci sequence using a loop
def fibo_loop(n):
    a = 0
    b = 1
    for _ in range(n):
        temp = a
        a = b
        b = temp + a
    return a
```

In [2]:
```
# calculate next digit in the fibonacci sequence using recursion
def fibo_recursive(n):
    if n > 1:
        return fibo_recursive(n-1) + fibo_recursive(n-2)
    else:
        return n
```

In [9]:
```
def memoize(f):
    memo = {}
    def helper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]
    return helper
```

In [10]:
```
fib_memo1 = memoize(fibo_recursive)
```

In [5]:
```
%timeit fibo_loop(15)
```

```
The slowest run took 9.71 times longer than the fastest. This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 1.09 µs per loop
```

In [6]:
```
%timeit fibo_recursive(15)
```

```
The slowest run took 4.69 times longer than the fastest. This could mean that an intermediate result is being cached.
1000 loops, best of 3: 259 µs per loop
```

In [11]:
```
%timeit fib_memo1(15)
```

```
The slowest run took 3170.55 times longer than the fastest. This could mean that an intermediate result is being cached.
10000000 loops, best of 3: 139 ns per loop
```

Results

- As we can see multiple calls are essentially free after the first time through, but note that even the first time through there will be savings.

- For example a call to Fib(15) will call Fib(13) twice and we only pay for 1. It will call Fib(12) three times and we only pay for 1. It will call Fib(11) four times and we only pay for 1 etc.

- So if we were doing Amortized analysis on this (shout out to Alex!) the cost of say Fib(5) becomes almost free (actually 1/10 it's actual cost and less subsequently) even the first time we call Fib(15)

- If we were to do a careful analysis of Fib(15) we would see that **even the first time it is called** the memoized version would only take 1/3 of the time of the regular recursive version

- Note if we call Fib(16) after calling Fib(15) it is virtually free and much faster than the loop version and ridiculously fast compared to the regular recursive version

Summary

- Memoization can be implemented in any language.
- In Python it's particularly easy. If you steal my 7 lines of code, you can memoize any function at the cost of a Python dictionary for each memoized function.
- No readability is sacrificed and performance gains can be substantial