CS61A Lecture 21

Wednesday, October 16th, 2019

Announcements

- Guerilla section on Saturday in Soda 2nd floor labs from 12 to 2.
- Midterm 2 in next Thursday, 10/24 from 8 to 10pm.
  - Covers lecture content through Wednesday 10/16.
  - You can bring two 2-sided sheets of handwritten notes.
  - Different treatment of orders of growth this semester.
    - Questions will be framed differently from past exams.
  - No BTree class covered in lecture this semester.
  - Seats assigned Wednesday 10/23.
  - No lecture next Wednesday 10/23.
  - No discussion section Wed 10/23 through Fri 10/25.

Efficiency

You should write programs that not only does the thing it's supposed to do, but also do it quickly and not use too many resources, whether we are talking about memory or network bandwidth.

Memoization

Let's go back to our `fib` function from the Tree Recursion lecture. We learnt that it's a pretty slow function, because evaluating `fib` on a number involved running `fib` on every number lower than itself multiple times.

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

We can measure exactly how efficient it is by measuring how many times the `fib` function was called:

```
def count(f):
    def counted(n):
        counted.call_count += 1
        return f(n)
    counted.call_count = 0
    return counted
```

We use this function like this:

```
>>> fib = count(fib)
>>> fib(5)
5
>>> fib.call_count
15
```

`fib` is slow because calling it on 5, it calls itself 15 times! And what if we count a bigger number?

```
>>> fib.call_count = 0 # We need to reset the counter
>>> fib(30)
832040
>>> fib.call_count
2692537
```

We can also use `count` as a decorator:

```
@count
def fib(n):...
```

So now:

```
>>> fib(30)
832040
>>> fib.call_count
2692537
>>> fib
<function count.<locals>.counted at ...>
```

We can fix this with a simple decorator called memoization. It remembers results that have been computed before:

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

The keys act as arguments that map to their return values. This memo assumes that calling `f` on the same number will always result in the same values.

Why is this faster? Well, when we call `fib` on 5, we call `fib` on 3 and 4. Calling `fib` on 4 calls `fib` on 2 and 3. So when we need the result of `fib(4)` to calculate `fib(5)`, the memoize decorator pulls the result from its cache instead of calculating it again!

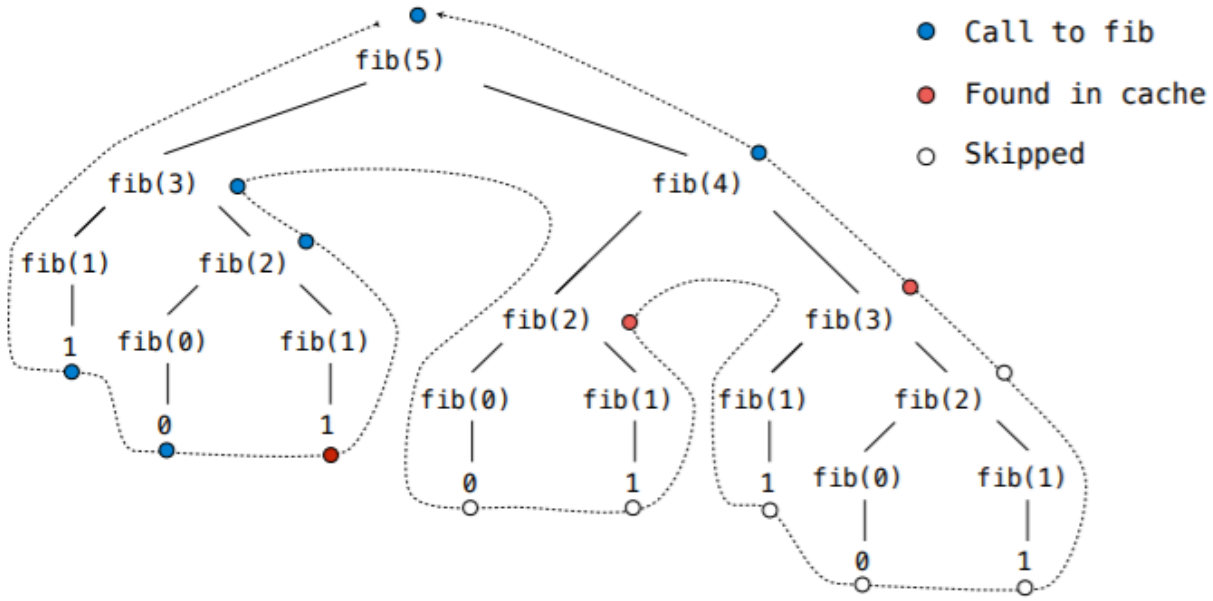
We can either use it as a decorator or like `count`:

```
@memo
...
```

or

```
>>> fib = memo(fib)
```

This makes the code dramatically faster:



```
>>> fib = count(fib)
>>> counted_fib = fib
>>> fib = memo(fib)
>>> fib = count(fib)
>>> fib(35)
9227465
>>> fib.call_count
69
>>> counted_fib.call_count
36
```

> **Building decorators**
>
> You can see that decorators all share a common structure:
>
> ```
> def <name>(function):
>     def <helper>(argument):
>         ... # call function on argument
>         return (value)
>     return <helper>
> ```

There is a built-in version of `memoize` called `lru_cache`, which we can import:

```
from functools import lru_cache
@lru_cache(None)
```

Sometimes, when we want to make our code more efficient, we do have to rewrite our code. Let's use the following example

Exponentiation

Goal: one more multiplication lets us double the problem size

```
def exp(b,n):
    if n == 0:
        return 1
    else:
        return b * exp(b,n-1)
```

What does the goal mean? For example, if we call `b` on `2n`, it will double the amount of work we need, because it calls `2n-1`, `2n-2` ...

We want to be able to do it so that doubling `n` to `2n`, requires only one more extra calculation.

```
def exp_fast(b,n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b,n//2))
    else:
        return b * exp_fast(b,n-1)
```

The above is not a tree recursion problem. A tree recursive problem is called tree recursive because it makes two or more recursive calls at each step, which the above function does not.

We will use Jupyter to draw a graph comparing `n` against its median runtime. The general shape of `exp` is a **linear runtime**. Doubling the input takes double the times.

Meanwhile, `exp_fast` produces a generally downward-shaping curve called a **logarithmic curve**. The property is that doubling the argument size adds a constant amount of runtime. In computing, it's called logarithmic time scaling.

For example, for linear time, 1024 times the input takes 1024 as much time, while for logarithmic time, 1024 times the input only increases the runtime by 10 times.

Orders of Growth

We categorize our code into general orders of growth, and below are the most common ones:

Quadratic time

Functions that process all pairs of values of values in a sequence of length `n` takes quadratic time.

```
def overlap(a,b):
    count = 0
    for item in a:
```

```
        for other in b:
            if item == other:
                count += 1
    return count
```

This above function takes exponentially longer to process the calculation for a slightly larger argument. When we graph it, it looks like the function $f(x) = x^2$.

Exponential time

This is even slower than quadratic! The easiest example we can use is `fib`.

Common Orders of Growth

Example	Growth
$a \times b^{n+1} = (a \times b^n) \times b$	Exponential growth. For example, recursive `fib`. Incrementing `n` multiplies time by a constant.
$a \times (n+1)^2 = (a \times n^2) + a - (2n+1)$	Quadratic growth. For example, `overlap`. Incrementing `n` increases time by `n` times a constant.
$a \times (n+1) = (a \times n) + a$	Linear growth. For example, slow `exp` incrementing `n` increases time by a constant.
$a \times ln(2 \times n) = (a \times ln(n)) + a \times ln(2)$	Logarithmic growth. For example `exp_fast`.
...	Constant growth. The time to evaluate does not depend on the size of the argument For example, looking up a dictionary key does not depend on the size of the dictionary.

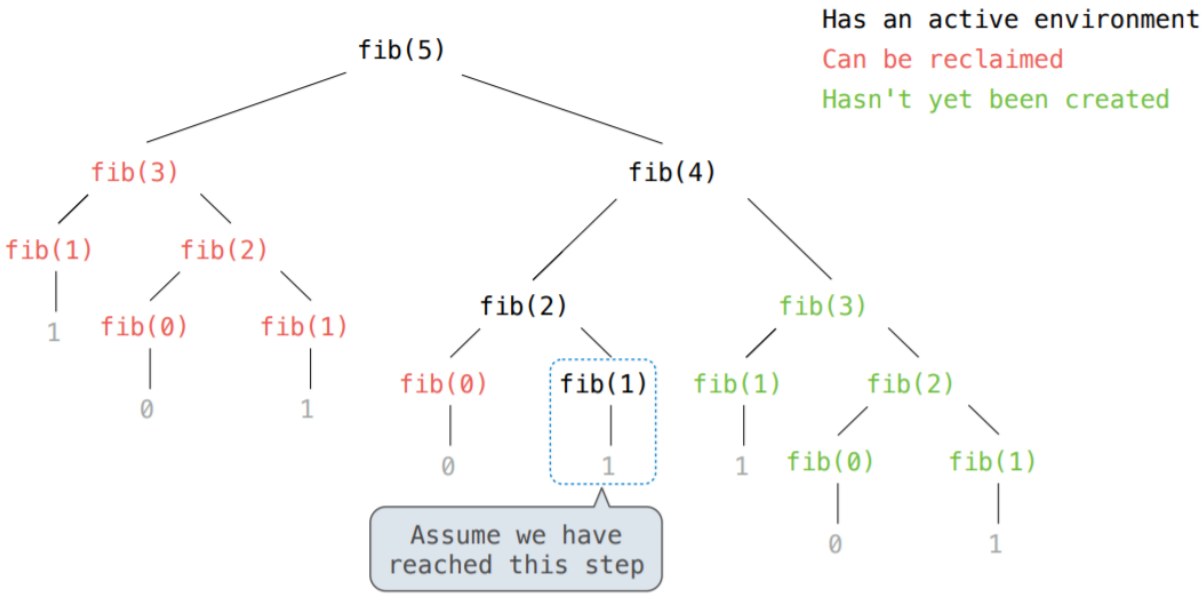
Space

Time is the most obvious measure of your program's efficiency, but it is not the only one. The other important measure is space, or how much memory your function takes.

Turns out, we already have a great way of keeping track of memory use, which is environments!

We can write a function to keep track of memory use:

```
def count_frames(f):
    def counted(n):
        counted.open_count += 1
        if counted.open_count > counted.max_count:
            counted.max_count = counted.open_count
        result = f(n)
        counted.open_count -= 1
        return result
    counted.open_count, counted.max_count = 0,0
    return counted
```



Python doesn't work like how you draw diagrams. It actively recycles the memory space occupied by frames that have already returned values, as you can see in the above diagram. Turns out, the maximum count of open frames is just the longest path of open frames you can find in a diagram like the one above.