Lecture 12: Efficiency

CS 61A - Summer 2024
Raymond Tan

Review: Generators

Example: count_partitions revisited

- Definition: A partition of a positive integer `n`, using parts up to size `m`, is a way in which `n` can be expressed as the sum of positive integer parts up to `m` in increasing order.
- We want to yield the string representation of all partitions possible

```
partitions(6, 4)
```

```
2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
```

count_partitions Solution

```python
def yield_partitions(n, m):
    """List partitions.

    >>> for p in yield_partitions(6, 4): print(p)
    2 + 4
    1 + 1 + 4
    3 + 3
    1 + 2 + 3
    1 + 1 + 1 + 3
    2 + 2 + 2
    1 + 1 + 2 + 2
    1 + 1 + 1 + 1 + 2
    1 + 1 + 1 + 1 + 1 + 1
    """
    if n > 0 and m > 0:
        if n == m:
            yield str(m)
        for p in yield_partitions(n-m, m):
            yield p + ' + ' + str(m)
        yield from yield_partitions(n, m-1)
```

- yield_partitions(n-m, m) is a recursive call representing using the largest value, m
    - We expect this to return a generator object, so we can iterate over it
    - Using the recursive leap of faith, this generator is an iterator over all the smaller partitions
- yield_partitions(n, m - 1) is a recursive call representing not using the largest value
    - We must decrement m to go down the path where we don’t use m
    - We can yield from this call directly since a generator is iterable, and there’s no work for us to do in this frame
        - Unlike the other recursive call, where we had to add str(m)

Efficiency

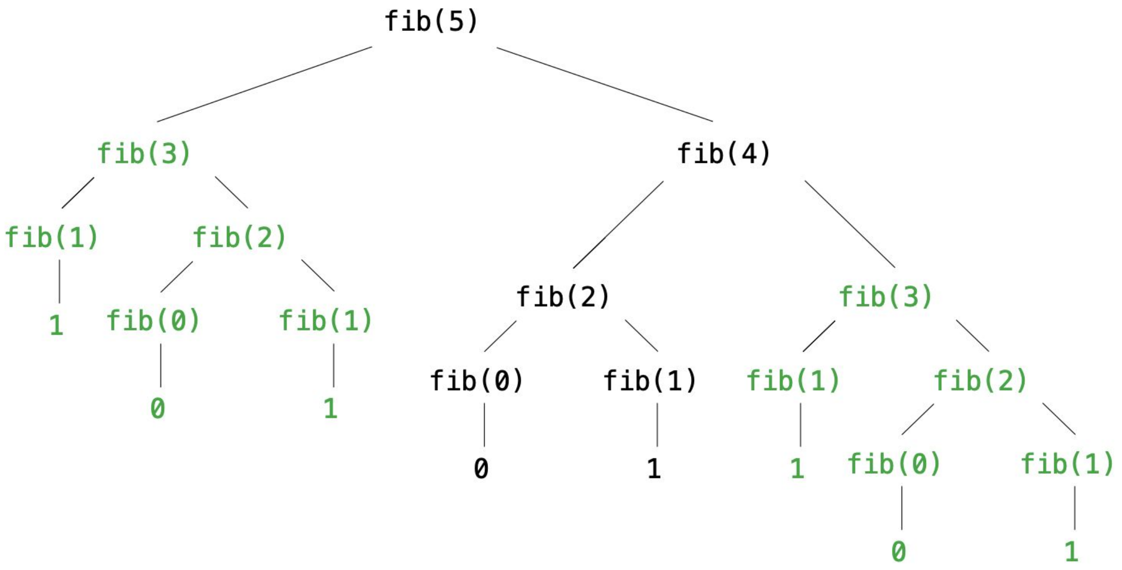
Review: Recursive Fibonacci

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

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987…

Repetition in Tree Recursive Computation

This process is highly repetitive; `fib` is called on the same argument multiple times



Let’s see how we can use ***memoization*** to speed this process up!

Memoization

- The idea of memoization is that each time we execute a recursive computation, we record the result of that computation
- That way, if we ever see exactly the same parameters a second time, we can access the result directly, rather than having to excuse a new series of recursive calls

Demo: Memoization

Example: Exponentiation

- **Goal**: Define our own exponentiation function. Takes two inputs, b and n, and raises b to the power of n.

exp: Implementation #1

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

Notice: We require around `n` recursive calls to evaluate `exp(b, n)`

exp: Implementation #2

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

def square(x):
    return x * x
```

When n is even, we get a chance to **halve our input** at the cost of one multiplication

If n is odd, we make a recursive call subtracting one from n, which leads us to the even case

exp: Comparison

- In the second implementation, one more multiplication operation allowed us to double the input size, n
- Although both implementations are **correct**, the second implementation is more **efficient**

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Example: overlap

Functions that process all pairs of values in a sequence of length n take quadratic time

```python
def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count
```

Note that this doesn't mean all functions with a nested for loop = quadratic!

overlap Visualization

```
overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

	3	5	7	6
4	0	0	0	0
5	0	1	0	0
6	0	0	0	1
5	0	1	0	0

Orders of Growth

Orders of Growth

- We measure the efficiency of a function by analyzing how the number of operations performed **scale as a factor of the input**
    - **Important:** Not measuring the actual amount of time (e.g milliseconds) it takes to execute a function
- The runtimes we'll discuss are (from most efficient to least efficient):
    - Constant, Logarithmic, Linear, Quadratic, Exponential

Orders of Growth

NOA = Number of Operations

- Constant growth: Increasing n doesn't affect NOA
- Logarithmic growth: Doubling n only affects NOA by a constant
  - Ex: Second implementation of `exp`
- Linear growth: Incrementing n increases NOA by a constant
  - Ex: First implementation of `exp`
- Quadratic growth: Incrementing n increases NOA by n times a constant
  - Ex: `overlap`
- Exponential growth: Incrementing n multiples NOA by a constant
  - Ex: Recursive `fib`

Most efficient

.
.
.
.
.
.
.
.
.
.
.

Least efficient

Big O/Big Theta Notation

- Big O/Big Theta is a common notation used when discussing orders of growth in computer science
- We won't be discussing the meaning of these notations in detail, but here they are:
  - Constant: $\Theta(1)$ $O(1)$
  - Logarithmic: $\Theta(\log n)$ $O(\log n)$
  - Linear: $\Theta(n)$ $O(n)$
  - Quadratic: $\Theta(n^2)$ $O(n^2)$
  - Exponential: $\Theta(b^n)$ $O(b^n)$

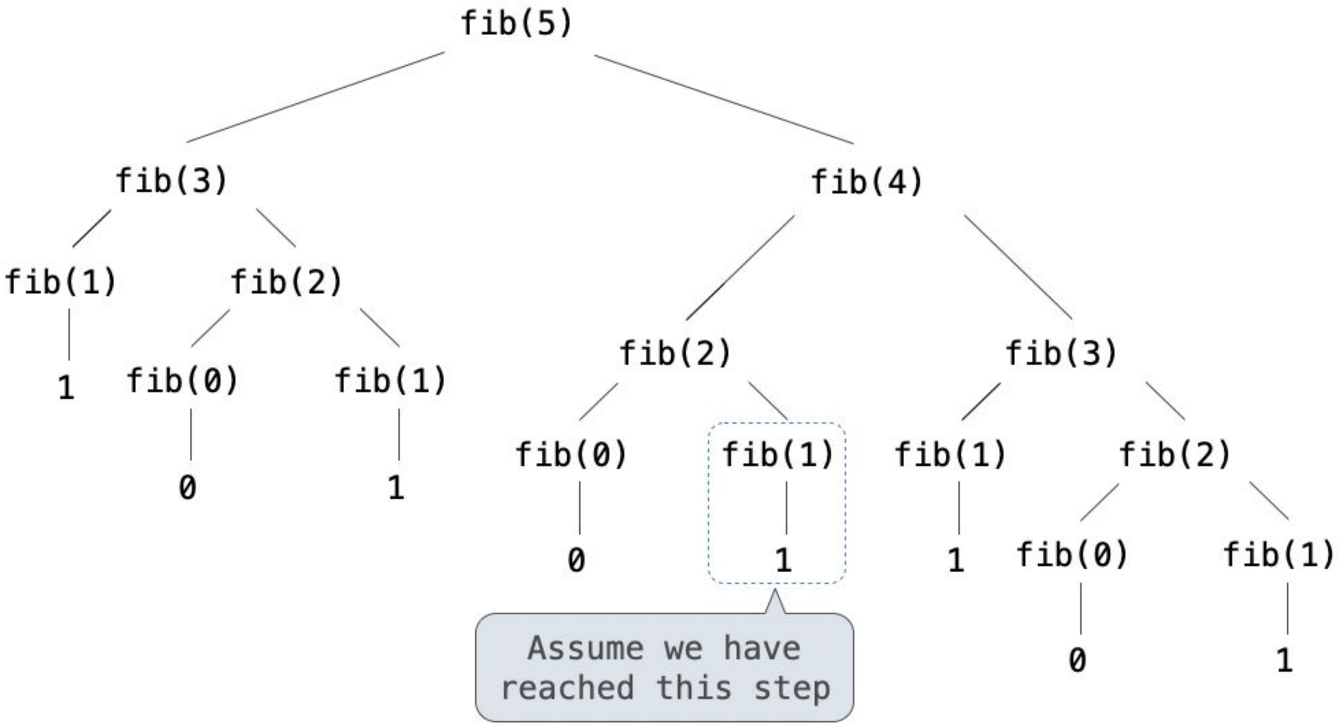
Break

Space

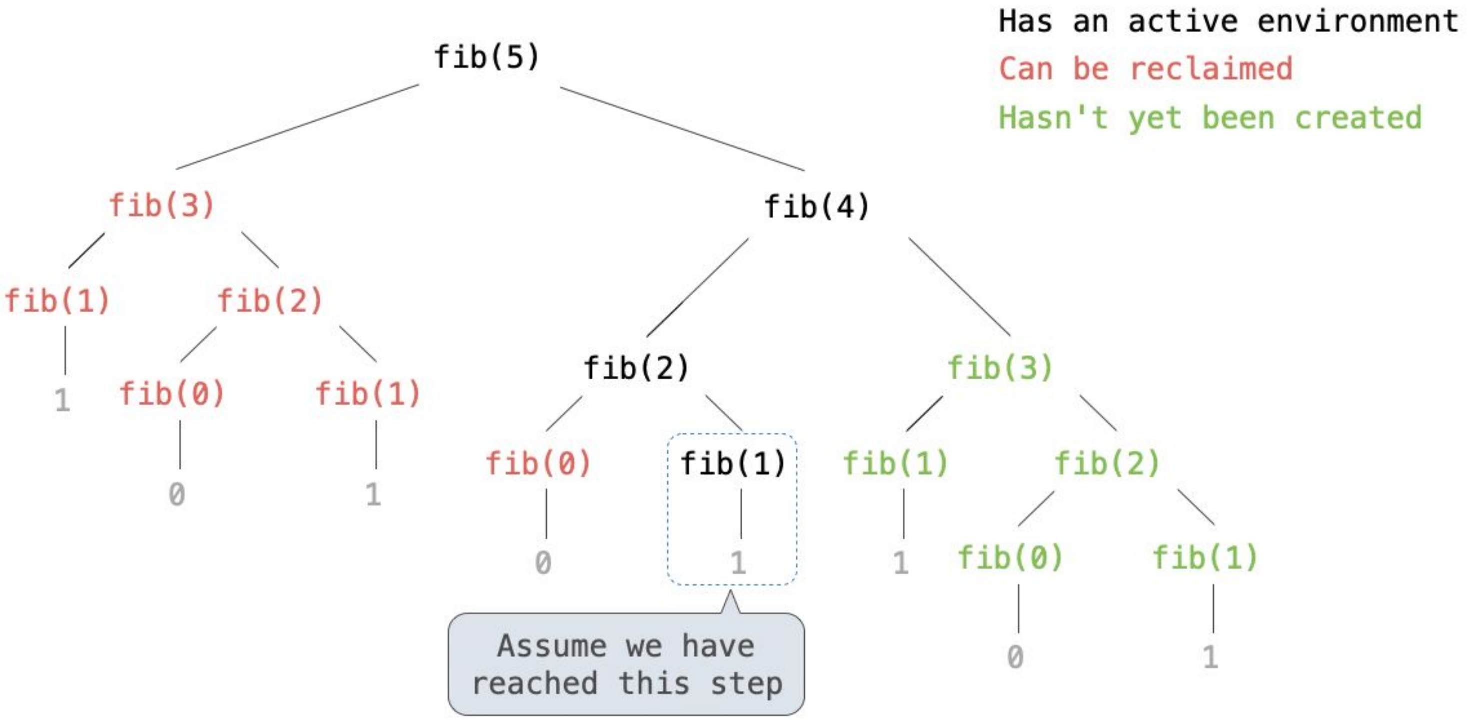
Space efficiency

- Along with being efficient about the amount of time our programs take, we must also be concerned with the amount of **space (memory)**
- Which environment frames do we need to keep during evaluation?
    - At any moment there is a set of active environments
    - Values and frames in active environments consume memory
    - Memory that is used for other value and frames can be recycled
- Which environments are **active**?
    - Environments for any function calls currently being evaluated
    - Parent environments of functions named in active environments

Fibonacci Space Consumption



Fibonacci Space Consumption



Summary

- Memoization is a way we can make repeating computations more efficient by storing the result of previous computations
- Orders of Growth are how we measure efficiency in programs
    - Constant, Logarithmic, Linear, Quadratic, Exponential
- Space (memory) is a consideration for efficiency as well. Must keep environments active that are:
    - Environments for any function calls currently being evaluated
    - Parent environments of functions named in active environments