Efficiency

Announcements

Memoization

Memoization

Idea: Remember the 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
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoization

Memoization is built into Python as the cache function.

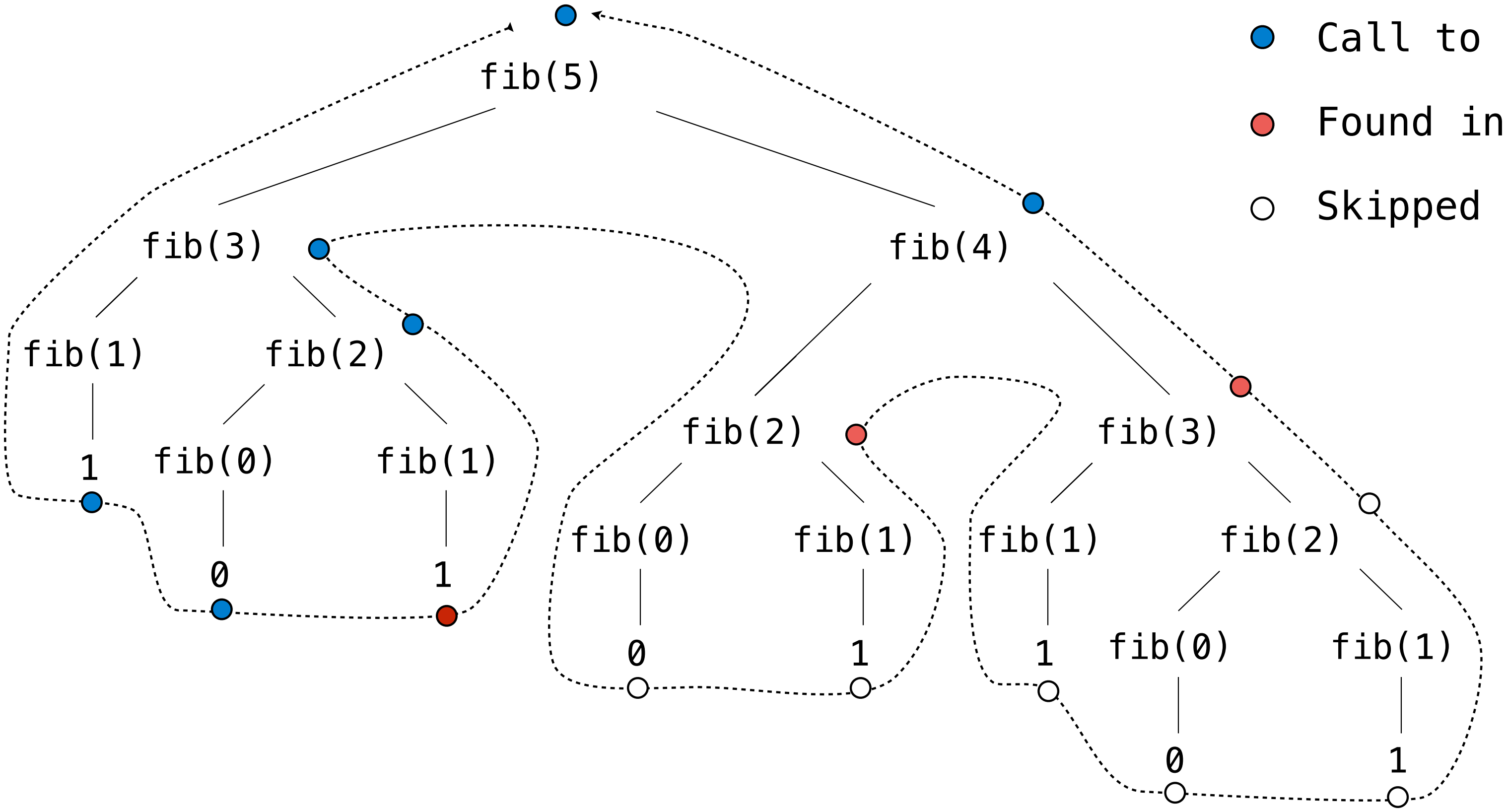
```
from functools import cache
faster_fib = cache(fib)
```

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

A decorator is a HOF which “wraps” the function definition.

(Demo)

Memoized Tree Recursion



🔵 Call to fib

🔴 Found in cache

○ Skipped

Orders of Growth

Common Orders of Growth

Exponential growth. O(2^n) E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

Quadratic growth. O(n^2)

Incrementing *n* increases *time* by *n* times a constant

Linear growth. O(n) E.g., iterative fib

Incrementing *n* increases *time* by a constant

Logarithmic growth. O(log(n))

Doubling *n* only increments *time* by a constant

Constant growth. O(1) Increasing *n* doesn't affect time

Match each function to its order of growth

Exponential growth. E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

Quadratic growth.

Incrementing *n* increases *time* by *n* times a constant

Linear growth.

Incrementing *n* increases *time* by a constant

Logarithmic growth.

Doubling *n* only increments *time* by a constant

Constant growth. Increasing *n* doesn't affect time

```
def search(s, v):
    """Return whether v is in the sorted list s.

    >>> evens = [2*x for x in range(50)]
    >>> search(evens, 22)
    True
    >>> search(evens, 23)
    False
    """
    if len(s) == 0:
        return False
    for item in s:
        if item == v:
            return True
    return False
```

Match each function to its order of growth

Exponential growth. E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

Quadratic growth.

Incrementing *n* increases *time* by *n* times a constant

Linear growth.

Incrementing *n* increases *time* by a constant

Logarithmic growth.

Doubling *n* only increments *time* by a constant

Constant growth. Increasing *n* doesn't affect time

```
def search_sorted(s, v):
    """Return whether v is in the sorted list s.

    >>> evens = [2*x for x in range(50)]
    >>> search_sorted(evens, 22)
    True
    >>> search_sorted(evens, 23)
    False
    """
    if len(s) == 0:
        return False
    center = len(s) // 2
    if s[center] == v:
        return True
    if s[center] > v:
        rest = s[:center]
    else:
        rest = s[center + 1:]
    return search_sorted(rest, v)
```

Match each function to its order of growth

Exponential growth. E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

Quadratic growth.

Incrementing *n* increases *time* by *n* times a constant

Linear growth.

Incrementing *n* increases *time* by a constant

Logarithmic growth.

Doubling *n* only increments *time* by a constant

Constant growth. Increasing *n* doesn't affect time

```
def near_pairs(s):
    """Return the length of the longest contiguous
    sequence of repeated elements in s.

    >>> near_pairs([3, 5, 2, 2, 4, 4, 4, 2, 2])
    3
    """
    count, max_count, last = 0, 0, None
    for i in range(len(s)):
        if count == 0 or s[i] == last:
            count += 1
            max_count = max(count, max_count)
        else:
            count = 1
        last = s[i]
    return max_count
```

```
def max_sum(s):
    """Return the largest sum of a contiguous
    subsequence of s.

    >>> max_sum([3, 5, -12, 2, -4, 4, -1, 4, 2, 2])
    11
    """
    largest = 0
    for i in range(len(s)):
        total = 0
        for j in range(i, len(s)):
            total += s[j]
            largest = max(largest, total)
    return largest
```

Match each function to its order of growth

Exponential growth. E.g., recursive fib

Incrementing *n* multiplies *time* by a constant

Quadratic growth.

Incrementing *n* increases *time* by *n* times a constant

Linear growth.

Incrementing *n* increases *time* by a constant

Logarithmic growth.

Doubling *n* only increments *time* by a constant

Constant growth. Increasing *n* doesn't affect time

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

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

faster_fib = memo(fib)
```

Recursion Visualizer with @cache:

https://www.recursionvisualizer.com/?function_definition=from%20functools%20import%20cache%0A%0A%40cache%0Adef%20fib%28n%29%3A%0A%20%20if%20n%20%3D%3D%200%3A%0A%20%20%20%20return%200%0A%20%20if%20n%20%3D%3D%201%3A%0A%20%20%20%20return%201%0A%20%20else%3A%0A%20%20%20%20return%20fib%28n%20-%201%29%20%2B%20fib%28n%20-%202%29%0A%20%20%20%20%0Adef%20memo%28f%29%3A%0A%20%20%20%20cache%20%3D%20%7B%7D%0A%20%20%20%20def%20memoized%28n%29%3A%0A%20%20%20%20%20%20%20%20if%20n%20not%20in%20cache%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20cache%5Bn%5D%20%3D%20f%28n%29%0A%20%20%20%20%20%20%20%20return%20cache%5Bn%5D%0A%20%20%20%20return%20memoized%0A&function_call=fib%2810%29

Practice:
Orders of Growth

Spring 2023 Midterm 2 Question 3(a) Part (iii)

Definition. A *prefix sum* of a sequence of numbers is the sum of the first n elements for some positive length n.

(1 pt) What is the order of growth of the time to run prefix(s) in terms of the length of s? Assume append takes one step (constant time) for any arguments.

```
def prefix(s):
    "Return a list of all prefix sums of list s."
    t = 0
    result = []
    for x in s:
        t = t + x
        result.append(t)
    return result
```