Efficiency

Announcements

Objects Review

Email

```
class Server:
    """An email server.

    >>> a, b = Client('John'), Client('Jack')
    >>> s = Server([a, b])
    >>> s.send(Email('Hi', 'John', 'Jack'))
    >>> b.inbox[0].msg
    'Hi'
    """
    def __init__(self, clients):
        self.clients = {c.name: c for c in clients}

      se        il
        ppend   em             inbox of              t it is addressed to."""
        self.clients[email.recipient_name].inbox.append(email)
```

Server | dict | Client | list

```
class Email:
    def __init__(self, msg, sender, recipient_name):
        self.msg = msg
        self.sender = sender
        self.recipient_name = recipient_name

class Client:
    def __init__(self, name):
        self.inbox = []
        self.name = name
```

A **Server** can send an **Email** to a **Client.**

To do this, it appends the **Email** to that **Client**'s **inbox** (a list).

To find the right **Client**, a **Server** has a dictionary called **clients** from the **name** of the **Client** (a str) to the **Client** instance.

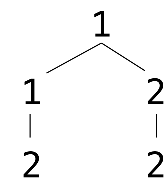
Tree Practice

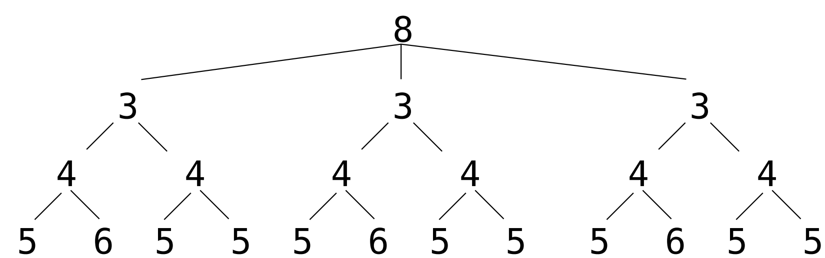
Example: Count Twins

Implement twins, which takes a Tree t. It return the number of pairs of sibling nodes whose labels are equal.

```
def twins(t):
    """Count the pairs of sibling nodes with equal labels.

    >>> t1 = Tree(3, [Tree(4, [Tree(5), Tree(6)]), Tree(4, [Tree(5), Tree(5)])])
    >>> twins(t1)  # 4 and 5
    2
    >>> twins(Tree(1, [Tree(1, [Tree(2)]), Tree(2, [Tree(2)])]))
    0
    >>> twins(Tree(8, [t1, t1, t1]))  # 3 pairs of twins at the top, plus 2 in each branch
    9
    """
    count = 0
    n = len(t.branches)
    for i in range(n-1):
        for j in range(i+1, n):
            if t.branches[i].label == t.branches[j].label:
                count += 1
    return count + sum([twins(b) for b in t.branches])
```



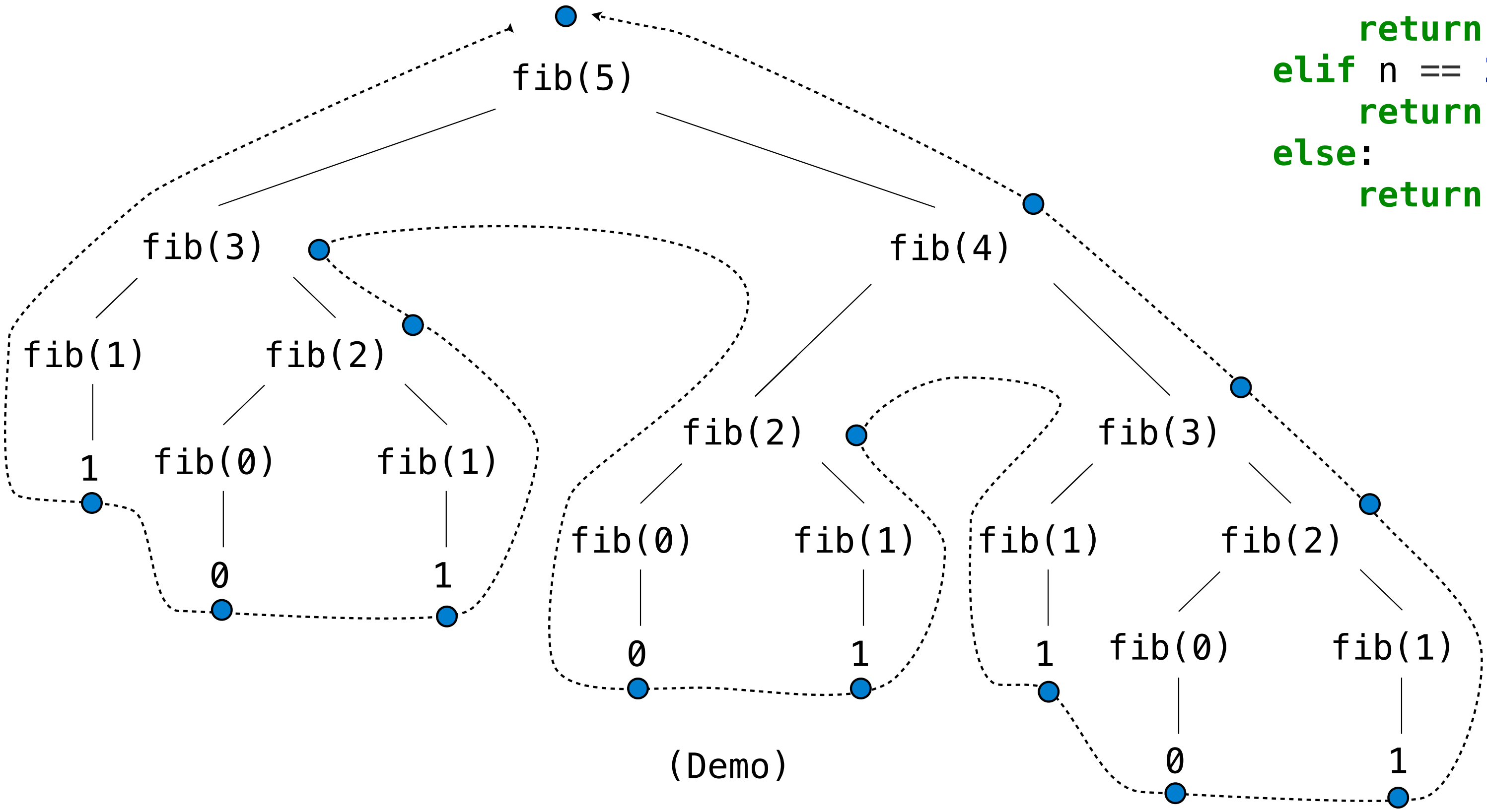


Measuring Efficiency

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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



(Demo)

Memoization

Memoization

Idea: Remember the results that have been computed before

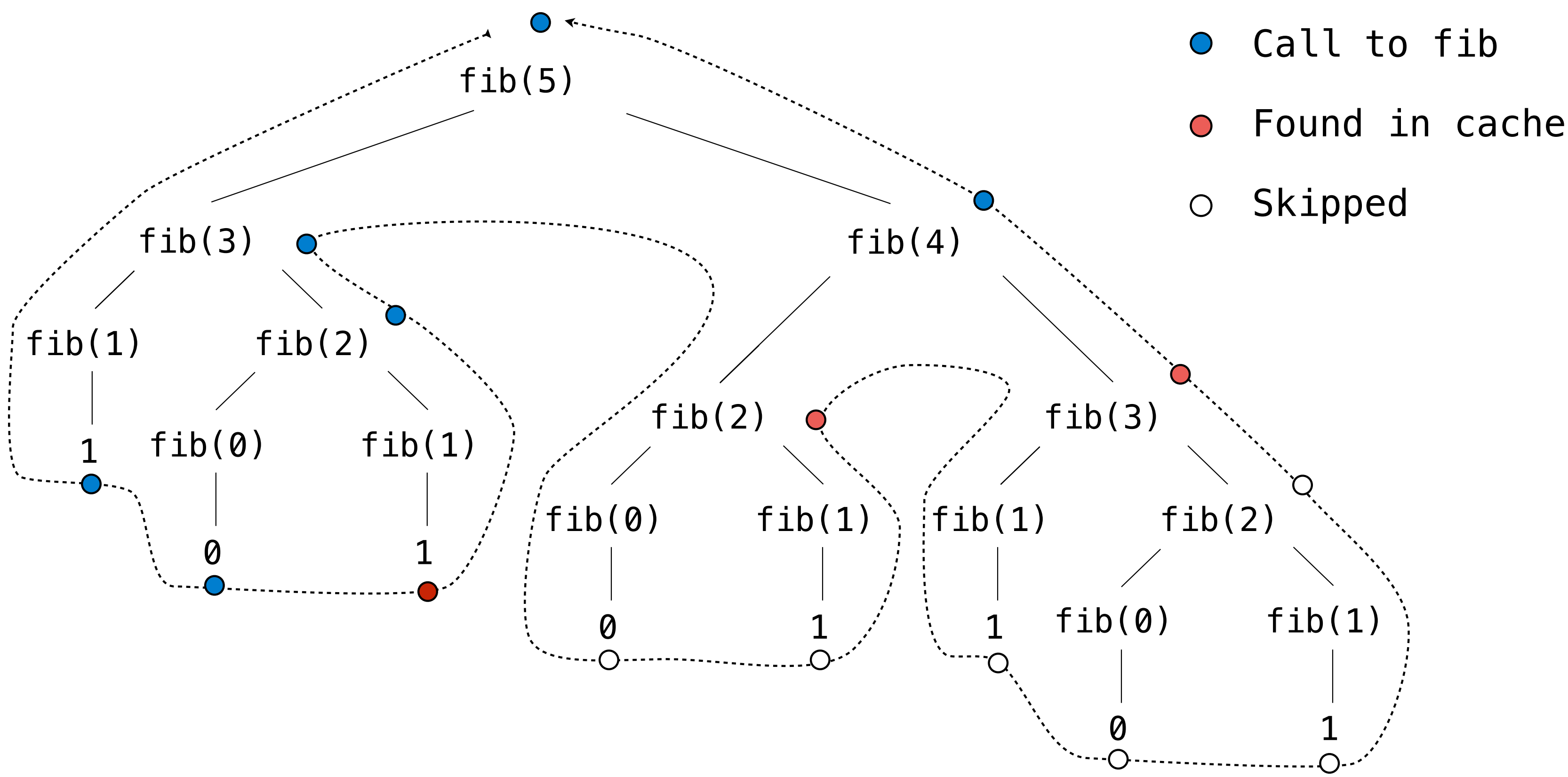
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoized Tree Recursion



Orders of Growth

Common Orders 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

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

```python
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

```python
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
```

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
```