Efficiency

Announcements / Mid-Sem Feedback Preview

Efficiency

Efficiency

A measure of how much resource consumption a computational task takes.

An analysis of computer programs rather than a technique for writing them.

In computer science, we are concerned with time and space efficiency.

The time efficiency of could determine how long a user has to wait for a webpage to load.

The space efficiency of your algorithm could determine how much memory running your application takes.

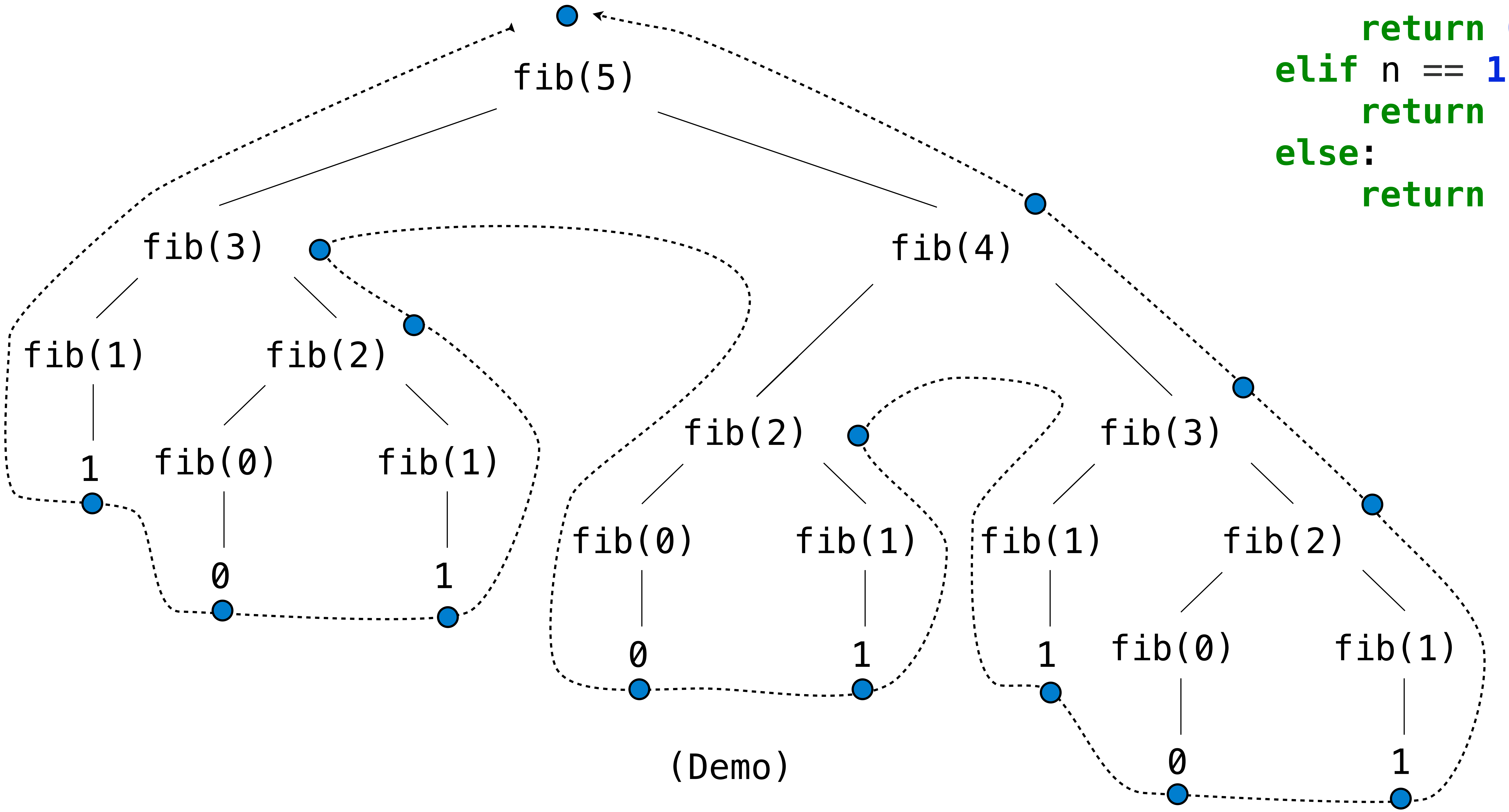
We are going down a layer of abstraction — opening up the black box.

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

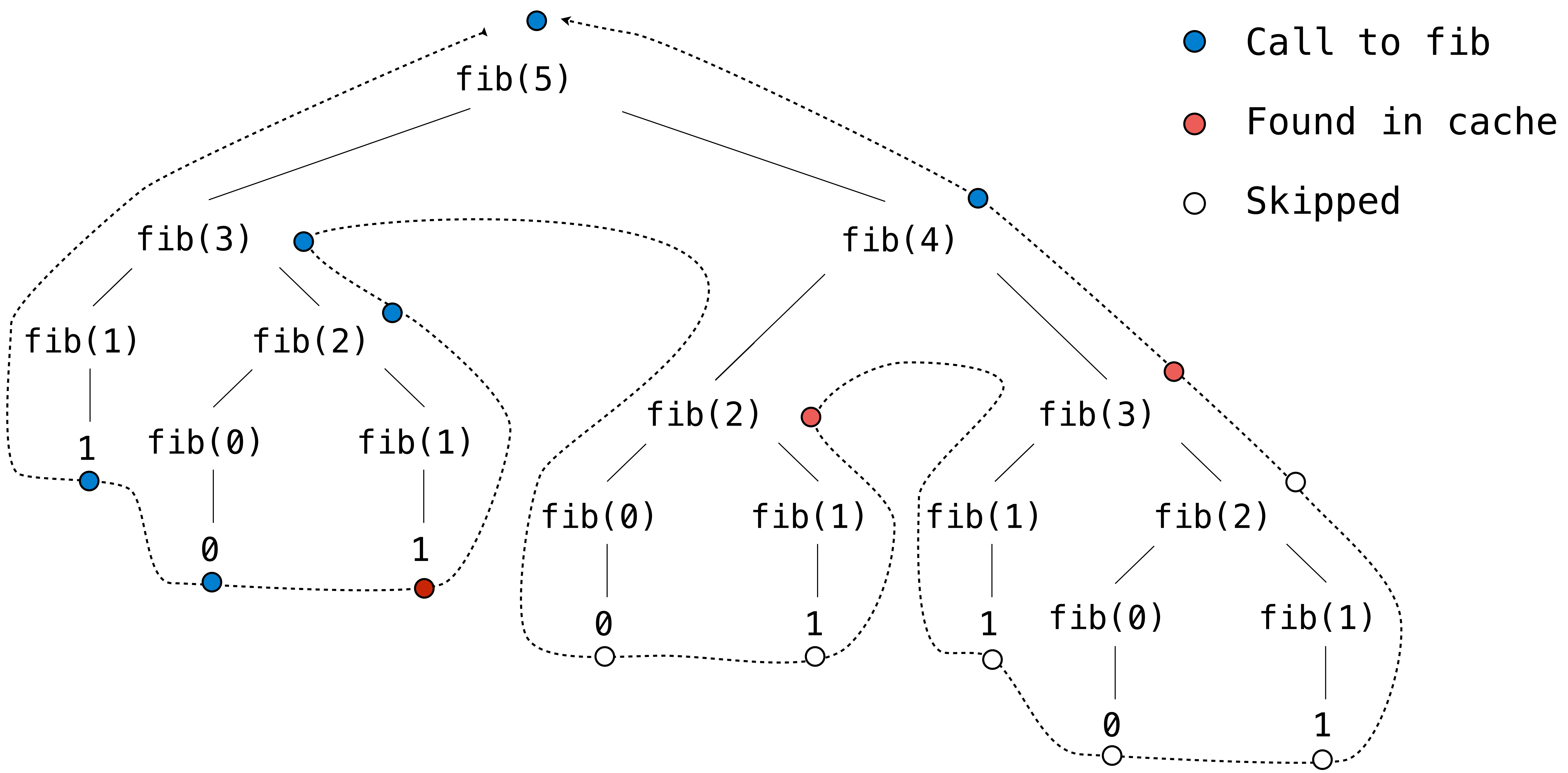
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

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

Order of Growth Practice

Match each function to its order of growth

Exponential growth. E.g., recursive fib
Incrementing *n* multiplies *time* by a constant
b ** n

Quadratic growth.
Incrementing *n* increases *time* by *n* times a constant
n ** 2

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

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 and + take one step.

```
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
1 + 1 + (len(s) * 2) + 1
n := len(s)
cost(prefix) = 3 + 2n
```

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.

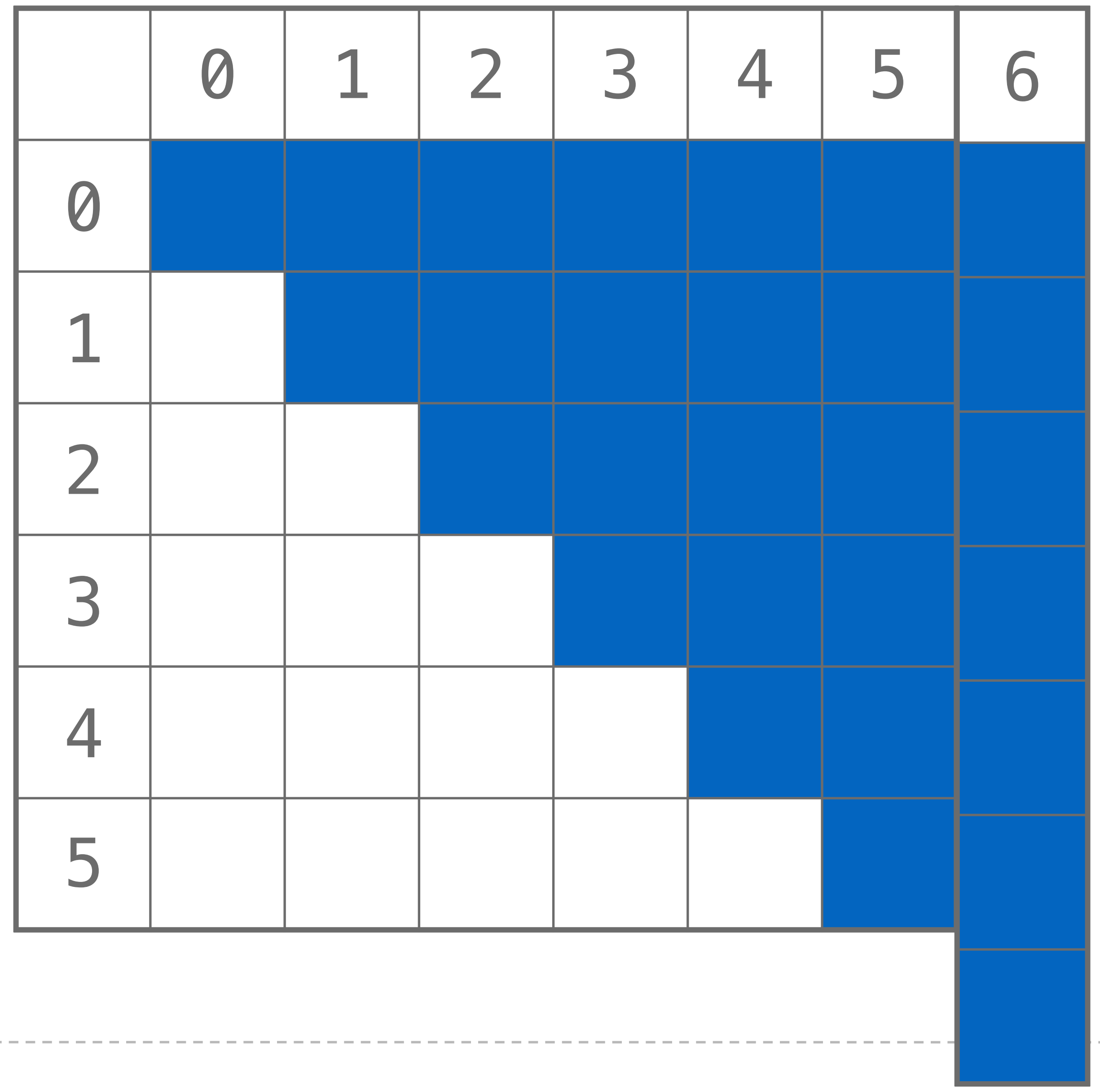
Incrementing *n* increases *time* by a constant

Logarithmic growth.

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

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

```
def 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 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
```

Mathematical Approach:

Sum of first n positive integers is

S_n = (n(n+1)) / 2

Expression for counting number of operations is quadratic with respect to `n`.

Visualizing Function Efficiency

(Demo)

More Linked Lists Practice

Recursion and Iteration

Many linked list processing functions can be written both iteratively and recursively

Recursive approach:

- What recursive call do you make?
- What does this recursive call do/return?
- How is this result useful in solving the problem?

```
def length(s):
    """The number of elements in s.

    >>> length(Link(3, Link(4, Link(5))))
    3
    """

    if s is Link.empty:

        return 0

    else:

        return ____1 + length(s.rest)____
```

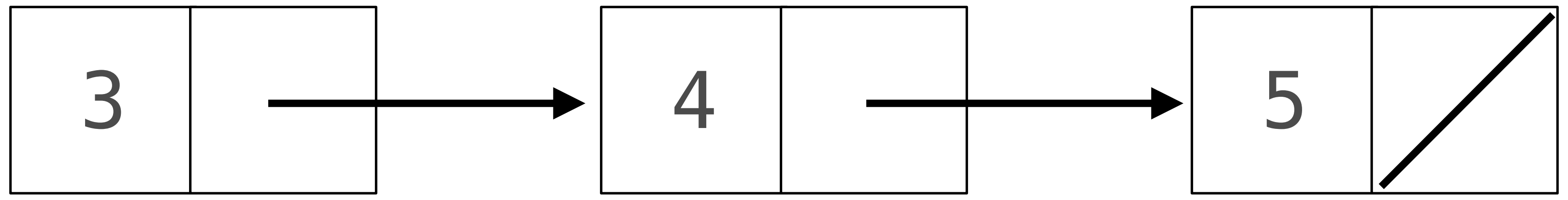
Iterative approach:

- Describe a process that solves the problem.
- Figure out what additional names you need to carry out this process.
- Implement the process using those names.

```
def length(s):
    """The number of elements in s.

    >>> length(Link(3, Link(4, Link(5))))
    3
    """

    k = __0__

    while _s is not Link.empty_:

        s, k = s.rest, _k + 1___

    return k
```

Constructing a Linked List

Build the rest of the linked list, then combine it with the first element.




```
s = Link.empty
s = Link(5, s)
s = Link(4, s)
s = Link(3, s)
```

```
def range_link(start, end):
    """Return a Link containing consecutive
    integers from start up to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    if start >= end:
        return Link.empty
    else:
        return Link(start, range_link(start + 1, end))
```

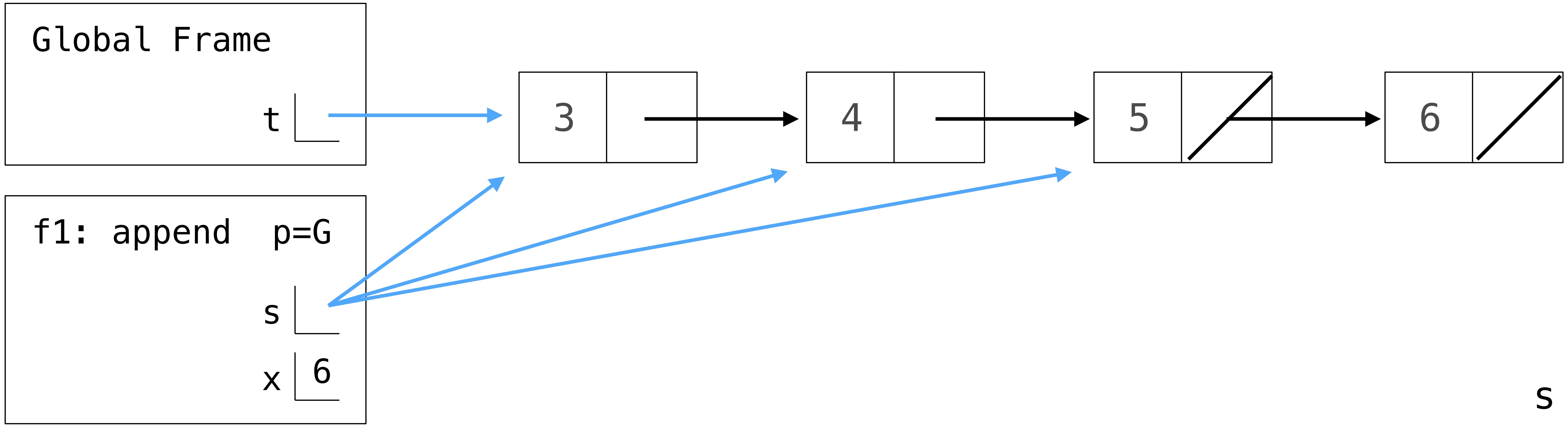
```
def range_link(start, end):
    """Return a Link containing consecutive
    integers from start to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """
    s = Link.empty
    k = end - 1
    while k >= start:
        s = Link(k, s)
        k -= 1
    return s
```

Linked List Mutation

To change the contents of a linked list, assign to first and rest attributes

Example: Append x to the end of non-empty s

```
>>> t = Link(3, Link(4, Link(5)))
>>> append(t, 6)
>>> t
Link(3, Link(4, Link(5, Link(6))))
```




```
s = s.rest

s.rest = Link(x)
```

Recursion and Iteration

Many linked list processing functions can be written both iteratively and recursively

Recursive approach:

- What recursive call do you make?
- What does this recursive call do/return?
- How is this result useful in solving the problem?

Iterative approach:

- Describe a process that solves the problem.
- Figure out what additional names you need to carry out this process.
- Implement the process using those names.

```
def append(s, x):
    """Append x to the end of non-empty s.
    >>> append(s, 6)   # returns None!
    >>> print(s)
    <3 4 5 6>
    """
    if s.rest is not Link.empty:
        append(s.rest, x)
    else:
        s.rest = Link(x)
```

```
def append(s, x):
    """Append x to the end of non-empty s.
    >>> append(s, 6)   # returns None!
    >>> print(s)
    <3 4 5 6>
    """
    while s.rest is not Link.empty:
        s = s.rest
    s.rest = Link(x)
```

Example: Pop

Implement pop, which takes a linked list s and positive integer i. It removes and returns the element at index i of s (assuming s.first has index 0).

```
def pop(s, i):
    """Remove and return element i from linked list s for positive i.
    >>> t = Link(3, Link(4, Link(5, Link(6))))
    >>> pop(t, 2)
    5
    >>> pop(t, 2)
    6
    >>> pop(t, 1)
    4
    >>> t
    Link(3)
    """
    assert i > 0 and i < length(s)

    for x in range(__i - 1__):

        s = s.rest
    result = s.rest.first
    _________________________
    s.rest = s.rest.rest
    _________________________

    return ____result____
```