Decomposition (Order of Growth & Linked List Practice)

Announcements

Advice from your fellow students

Before lectures

- "Highly recommend watching the videos before attending lecture..."
- "Watch the pre-lecture videos! Then its easier to focus on the examples in class..."
- "I would recommend watching the Youtube videos as they are very well made..."

Lectures

- "Watch lecture and do practice problems over and over again"
- "Watch the lecture videos, attend live lectures..."
- "Just go to lecture."

Practice

- "Do as many practice exams as you can..."
- "Do 1-2 practice test problems every day..."
- "Do a lot of practice tests. They are really helpful..."

Exams

- "Grind all the practice exams in the website..."
- "Do past exams and stay on top of the material"
- "Take previous midterms for practice."

Other advice

- "Don't fall behind and be on top of your work"
- "Make sure to stay up to date with lectures and videos"
- "Start studying earlier than you think you need to..."

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

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

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
```

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.

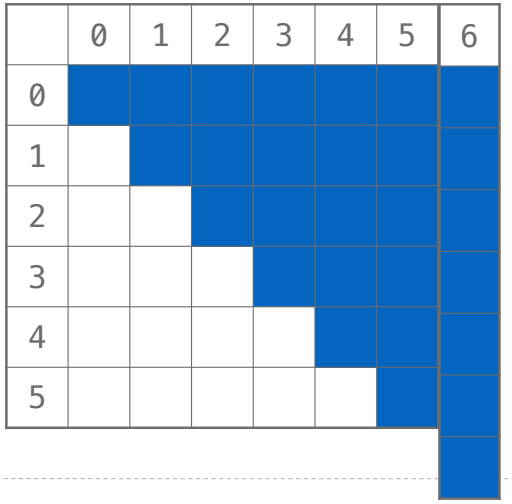
Incrementing *n* increases *time* by a constant

Logarithmic growth.

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

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

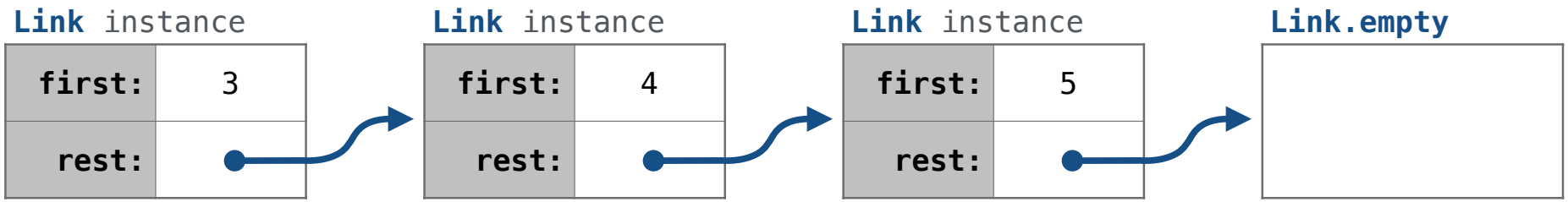
```
def max_sum(s):
    """Return the largest sum of a contiguous
    subsequence of s.
    >>> max_sum([3, 5, -12, 2, -4, 4, -1, 4, 2, 2])
    11
    """
    largest = 0
    for i in range(len(s)):
        total = 0
        for j in range(i, len(s)):
            total += s[j]
            largest = max(largest, total)
    return largest
```

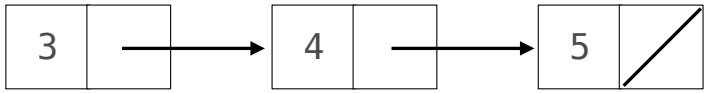


Linked Lists Practice

Linked List Notation

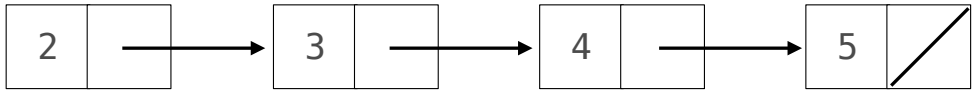
```
s = Link(3, Link(4, Link(5)))
```



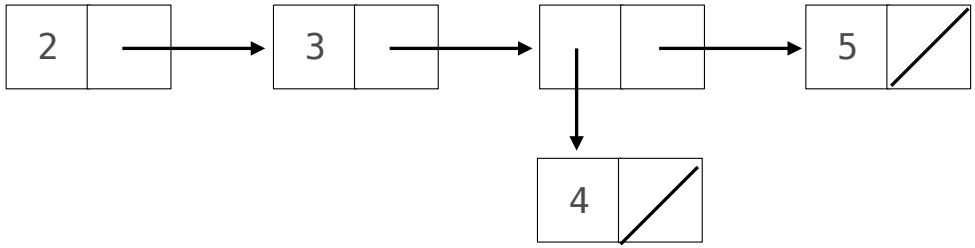


Nested Linked Lists

```
>>> s = Link(2, Link(3, Link(    4     , Link(5))))
>>> t = Link(2, Link(3, Link(  Link(4) , Link(5))))
>>> print(s)
<2 3 4 5>
```

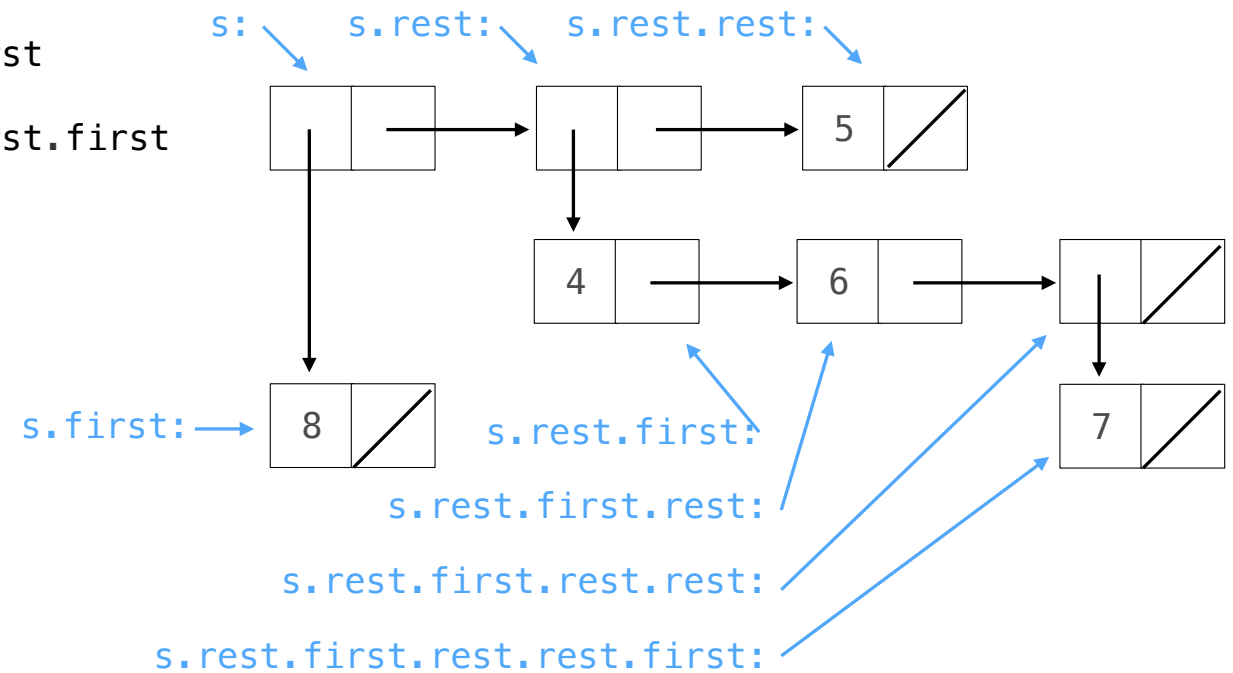



```
>>> print(t)
<2 3 <4> 5>
```



Nested Linked Lists

```
>>> s = Link(Link(8), Link(Link(4, Link(6, Link(Link(7)))), Link(5)))
>>> print(s)
<<8> <4 6 <7>> 5>
>>> s.first.first
8
>>> s.rest.first.rest.rest.first
Link(7)
>>> s.rest.first.rest.rest.first.first
7
```



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)
```

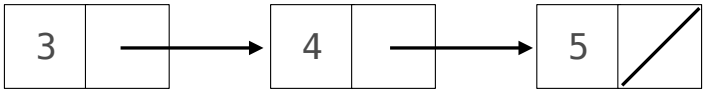
Iterative approach:

- 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))
```

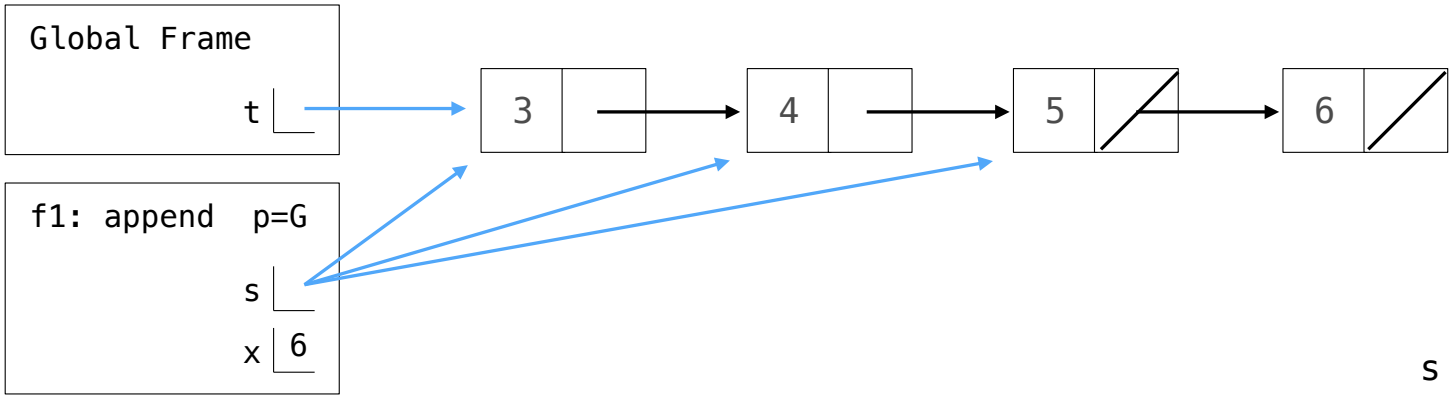
```
def range_link(start, end):
    """Return a Link containing consecutive
    integers from start to end.

    >>> 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?

```
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)
```

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>
    """
    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
```