CS 61A
Fall 2022

Linked Lists, Mutable Trees, Efficiency

Discussion 8: October 19, 2022

Representation: Repr, Str

Q1: WWPD: Repr-esentation

Note: This is not the typical way repr is used, nor is this way of writing repr recommended, this problem is mainly just to make sure you understand how repr and str work.

```
class A:
    def __init__(self, x):
        self.x = x

    def __repr__(self):
         return self.x

    def __str__(self):
         return self.x * 2

class B:
    def __init__(self):
         print('boo!')
         self.a = []

    def add_a(self, a):
         self.a.append(a)

    def __repr__(self):
         print(len(self.a))
         ret = ''
         for a in self.a:
             ret += str(a)
         return ret
```

Given the above class definitions, what will the following lines output?

```
>>> A('one')
```

one

```
>>> print(A('one'))
```

oneone

```
>>> repr(A('two'))
```

‘two’

```
>>> b = B()
```

boo!

```
>>> b.add_a(A('a'))
>>> b.add_a(A('b'))
>>> b
```

2

aabb

Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

To check if a linked list is an empty linked list, compare it against the class attribute `Link.empty`:

```python
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

You can find an implementation of the `Link` class below:

```python
class Link:
    """A linked list."""
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Q2: The Hy-rules of Linked Lists

In this question, we are given the following Linked List:

```
ganondorf = Link('zelda', Link('link', Link('sheik', Link.empty)))
```

What expression would give us the value `'sheik'` from this Linked List?

`ganondorf.rest.rest.first`

What is the value of `ganondorf.rest.first`?

`'link'`

What would be the value of `str(ganondorf)`?

`'<zelda link sheik>'`

What expression would mutate this linked list to `<zelda ganondorf sheik>`?

`ganondorf.rest.first = 'ganondorf'`

Q3: Sum Nums

Write a function that takes in a linked list and returns the sum of all its elements. You may assume all elements in `s` are integers. Try to implement this recursively!

```
def sum_nums(s):
    """
    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """
    if s == Link.empty:
        return 0
    return s.first + sum_nums(s.rest)
```

Q4: Multiply Links

Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
    # Implementation Note: you might not need all lines in this skeleton code
    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))
```

For our base case, if we detect that any of the lists in the list of `Links` is empty, we can return the empty linked list as we're not going to multiply anything.

Otherwise, we compute the product of all the firsts in our list of `Links`. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling `multiply_lnks` will be a linked list! We'll use the product we've built so far as the first item in the returned `Link`, and then the result of the recursive call as the rest of that `Link`.

Next, we have the iterative solution:

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest is Link.empty
    True
    """
    # Alternate iterative approach
    import operator
    from functools import reduce
    def prod(factors):
         return reduce(operator.mul, factors, 1)

    head = Link.empty
    tail = head
    while Link.empty not in lst_of_lnks:
        all_prod = prod([l.first for l in lst_of_lnks])
        if head is Link.empty:
            head = Link(all_prod)
            tail = head
        else:
            tail.rest = Link(all_prod)
            tail = tail.rest
        lst_of_lnks = [l.rest for l in lst_of_lnks]
    return head
```

The iterative solution is a bit more involved than the recursive solution. Instead of building the list **backwards** as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we'll build the resulting linked list as we go along.

We use`head` and `tail` to track the front and end of the new linked list we're creating. Our stopping condition for the loop is if any of the `Link`s in our list of `Link`s runs out of items.

Finally, there's some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

Q5: Flip Two

Write a recursive function `flip_two` that takes as input a linked list `s` and mutates `s` so that every pair is flipped.

```
def flip_two(s):
    """
    >>> one_lnk = Link(1)
    >>> flip_two(one_lnk)
    >>> one_lnk
    Link(1)
    >>> lnk = Link(1, Link(2, Link(3, Link(4, Link(5)))))
    >>> flip_two(lnk)
    >>> lnk
    Link(2, Link(1, Link(4, Link(3, Link(5)))))
    """
    # Recursive solution:
    if s is Link.empty or s.rest is Link.empty:
        return
    s.first, s.rest.first = s.rest.first, s.first
    flip_two(s.rest.rest)


    # For an extra challenge, try writing out an iterative approach as well below!
    return # separating recursive and iterative implementations

    # Iterative approach
    while s is not Link.empty and s.rest is not Link.empty:
        s.first, s.rest.first = s.rest.first, s.first
        s = s.rest.rest
```

If there's only a single item (or no item) to flip, then we're done.

Otherwise, we swap the contents of the first and second items in the list. Since we've handled the first two items, we then need to recurse on `s.rest.rest`.

Although the question explicitly asks for a recursive solution, there is also a fairly similar iterative solution (see python solution).

We will advance `s` until we see there are no more items or there is only one more Link object to process. Processing each `Link` involves swapping the contents of the first and second items in the list (same as the recursive solution).

Trees

We define a tree to be a recursive data abstraction that has a label (the value stored in the root of the tree) and branches (a list of trees directly underneath the root). Previously, we implemented the tree abstraction using Python lists. Let's look at another implementation using objects instead:

```
class Tree:
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = branches

    def is_leaf(self):
        return not self.branches
```

With this implementation, we can mutate a tree using attribute assignment, which wasn't possible in the previous implementation using lists. That's why we sometimes call these objects "mutable trees."

```
>>> t = Tree(3, [Tree(4), Tree(5)])
>>> t.label = 5
>>> t.label
5
```

Q6: Make Even

Define a function `make_even` which takes in a tree `t` whose values are integers, and mutates the tree such that all the odd integers are increased by 1 and all the even integers remain the same.

```
def make_even(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(3)]), Tree(4), Tree(5)])
    >>> make_even(t)
    >>> t.label
    2
    >>> t.branches[0].branches[0].label
    4
    """
    if t.label % 2 != 0:
        t.label += 1
    for branch in t.branches:
        make_even(branch)
    return
    # Alternate Solution

    t.label += t.label % 2
    for branch in t.branches:
        make_even(branch)
    return
```

Q7: Add Leaves

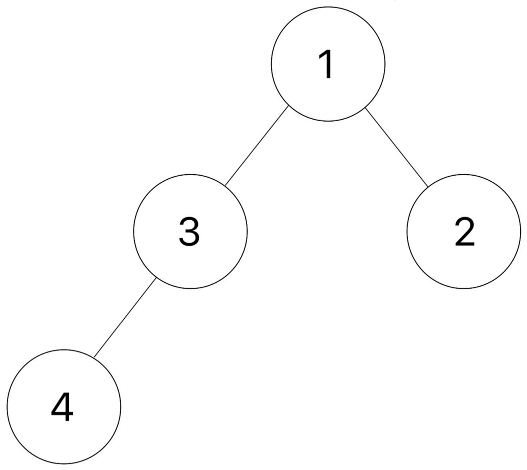
Implement `add_d_leaves`, a function that takes in a `Tree` instance `t` and a number `v`.

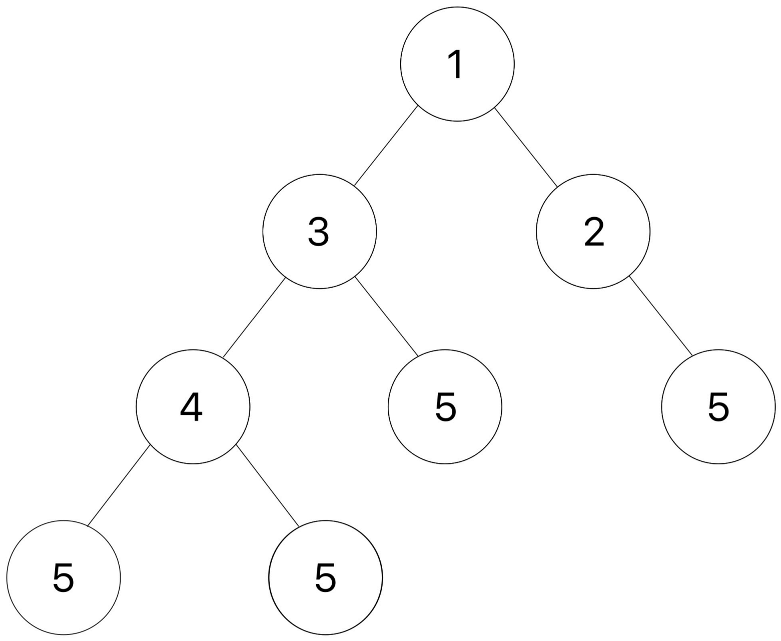
We define the depth of a node in `t` to be the number of edges from the root to that node. The depth of root is therefore 0.

For each node in the tree, you should add `d` leaves to it, where `d` is the depth of the node. Every added leaf should have a label of `v`. If the node at this depth has existing branches, you should add these leaves to the end of that list of branches.

For example, you should be adding 1 leaf with label `v` to each node at depth 1, 2 leaves to each node at depth 2, and so on.

Here is an example of a tree `t`(shown on the left) and the result after `add_d_leaves` is applied with `v` as 5.





Try drawing out the second doctest to visualize how the function is mutating `t3`.

Hint: Use a helper function to keep track of the depth!

```
def add_d_leaves(t, v):
    """Add d leaves containing v to each node at every depth d.

    >>> t_one_to_four = Tree(1, [Tree(2), Tree(3, [Tree(4)])])
    >>> print(t_one_to_four)
    1
      2
      3
        4
    >>> add_d_leaves(t_one_to_four, 5)
    >>> print(t_one_to_four)
    1
      2
        5
      3
        4
          5
          5
        5

    >>> t1 = Tree(1, [Tree(3)])
    >>> add_d_leaves(t1, 4)
    >>> t1
    Tree(1, [Tree(3, [Tree(4)])])
    >>> t2 = Tree(2, [Tree(5), Tree(6)])
    >>> t3 = Tree(3, [t1, Tree(0), t2])
    >>> print(t3)
    3
      1
        3
          4
      0
      2
        5
        6
    >>> add_d_leaves(t3, 10)
    >>> print(t3)
    3
      1
        3
          4
            10
            10
            10
          10
          10
        10
      0
        10
      2
        5
```

A first thought is to recursively call `add_d_leaves` on each branch to add the leaves to the branches. However, notice that each recursive call would need to know what the current depth is in order to add that many leaves. We could try initializing a variable within the body of the function, but by now we know that in order to keep track of changing values across recursive calls we should use a helper function!

The helper function should take in a tree and a depth value, and we will define it as a function that adds `d` leaves to the branches of the root node, `d + 1` leaves to the branches of each of the root node's branches, and so on:

```
def add_leaves(t, d):
    """Adds a number of leaves to each node in t equivalent to the depth of
    the node, assuming that the root node is at depth d, the children of
    the root node are at depth d + 1, and so on."""
    ...
```

We don't need a parameter for `v` since that value won't change and we can access it from the parent frame. With this function defined as such, we can call `add_leaves` with arguments `t` and 0 to add leaves starting at depth 0.

```
def add_d_leaves(t, v):
    def add_leaves(t, d):
        """Adds a number of leaves to each node in t equivalent to the depth of
        the node, assuming that the root node is at depth d, the children of
        the root node are at depth d + 1, and so on."""
        ...
    add_leaves(t, 0)
```

Inside the helper function, we can now call it recursively on each branch. Each node's branch is one depth level greater than the node itself, so we should update `d` to `d + 1`:

```
def add_leaves(t, d):
    for b in t.branches:
        add_leaves(b, d + 1)
    ...
```

Now that we've made these recursive calls, let's take a step back and look at our progress. Taking the leap of faith, we know that each recursive call should've successfully added the correct number of leaves at each node in each branch. That means that the only step left is to add the correct number of leaves to the current node!

The parameter `d` tells us how many leaves to add at this node. Since we are mutating `t` to add these leaves, we need to mutate the list of `t`'s branches. We know a few different ways to mutatively add elements to a list:`insert`, `append`, and `extend`. Which one makes most sense to use here? Well, we know that we have to add `d` elements to the end of `t.branches`. Index doesn't matter so we can rule out `insert`. `append` is good for adding a single element, while `extend` is useful for adding multiple elements contained in a list, so let's use `extend`!

The input to `extend` should be a list, so how do we create a list with the leaves that we need? The most concise way is with a list comprehension. To create each leaf, we call `Tree(v)`:

```
[Tree(v) for _ in range(d)]
```

Now, we just have to extend `t.branches` by this list:

```
def add_leaves(t, d):
    for b in t.branches:
        add_leaves(b, d + 1)
    t.branches.extend([Tree(v) for _ in range(d)])
```

Do we need an explicitly base case? Let's take a look at what happens when `t` is a leaf. In that case, `t.branches` would be an empty list, so we would not enter the `for` loop. Then, the function will extend `t.branches`, which is an empty list, by a list containing the new leaves. This is exactly the desired result, so no base case is needed!

Efficiency (Orders of Growth)

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by *runtime*?

Example 1: `square(1)` requires one primitive operation: multiplication. `square(100)` also requires one. No matter what input `n` we pass into `square`, it always takes a *constant* number of operations (1). In other words, this function has a runtime complexity of $\Theta(1)$.

As an illustration, check out the table below:

input	function call	return value	operations
1	`square(1)`	1*1	1
2	`square(2)`	2*2	1
...	...	...	...
100	`square(100)`	100*100	1
...	...	...	...
n	`square(n)`	n*n	1

Example 2: `factorial(1)` requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of n, the runtime (number of operations) increases **linearly** proportional to the input. In other words, this function has a runtime complexity of Θ(`n`).

As an illustration, check out the table below:

input	function call	return value	operations
1	`factorial(1)`	1*1	1
2	`factorial(2)`	2*1*1	2
...	...	...	...
100	`factorial(100)`	100*99*...*1*1	100
...	...	...	...
n	`factorial(n)`	n*(n-1)*...*1*1	n

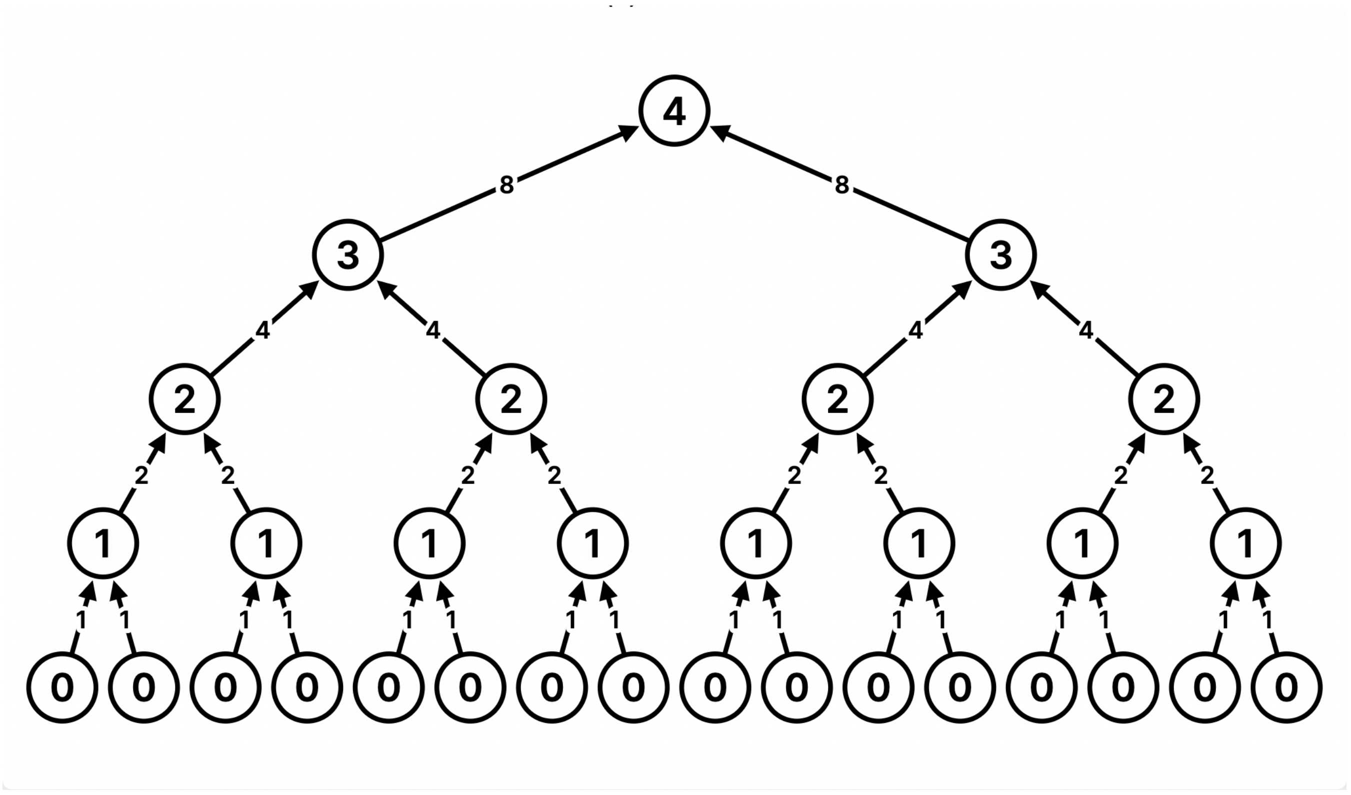
Example 3: Consider the following function: def bar(n): for a in range(n): for b in range(n): print(a,b)

`bar(1)` requires 1 print statements, while `bar(100)` requires `100*100 = 10000` print statements (each time `a` increments, we have 100 print statements due to the inner for loop). Thus, the runtime increases **quadratically** proportional to the input. In other words, this function has a runtime complexity of Θ(`n^2`).

input	function call	operations (prints)
1	`bar(1)`	1
2	`bar(2)`	4
...	...	...
100	`bar(100)`	10000
...	...	...
n	`bar(n)`	n^2

Example 4: Consder the following function: def rec(n): if n == 0: return 1 else: return rec(n - 1) + rec(n - 1)

`rec(1)` requires one addition, as it returns `rec(0) + rec(0)`, and `rec(0)` hits the base case and requires no further additions. but `rec(4)` requires `2^4 - 1` = 15 additions. To further understand the intuition, we can take a look at the recurisve tree below. To get `rec(4)`, we need one addition. We have two calls to `rec(3)`, which each require one addition, so this level needs two additions. Then we have four calls to `rec(2)`, so this level requires four additions, and so on down the tree. In total, this adds up to 1 + 2 + 4 + 8 = 15 additions.



Recursive Call Tree

As we increase the input size of n, the runtime (number of operations) increases **exponentially** proportional to the input. In other words, this function has a runtime complexity of Θ(`2^n`).

As an illustration, check out the table below:

input	function call	return value	operations
1	`rec(1)`	2	1
2	`rec(2)`	4	3
...	...	...	...
10	`rec(10)`	1024	1023
...	...	...	...
n	`rec(n)`	`2^n`	`2^n`

Here are some general guidelines for finding the order of growth for the runtime of a function:

- If the function is recursive or iterative, you can subdivide the problem as seen above:
  - Count the number of recursive calls/iterations that will be made in terms of input size `n`.

  - Find how much work is done per recursive call or iteration in terms of input size `n`.
  - The answer is usually the product of the above two, but be sure to pay attention to control flow!
- If the function calls helper functions that are not constant-time, you need to take the runtime of the helper functions into consideration.
- We can ignore constant factors. For example `1000000n` and `n` steps are both linear.
- We can also ignore smaller factors. For example if `h` calls `f` and `g`, and `f` is Quadratic while `g` is linear, then `h` is Quadratic.
- For the purposes of this class, we take a fairly coarse view of efficiency. All the problems we cover in this course can be grouped as one of the following:
  - Constant: the amount of time does not change based on the input size. Rule: `n --> 2n` means `t --> t`.
  - Logarithmic: the amount of time changes based on the logarithm of the input size. Rule: `n --> 2n` means `t --> t + k`.
  - Linear: the amount of time changes with direct proportion to the size of the input. Rule: `n --> 2n` means `t --> 2t`.
  - Quadratic: the amount of time changes based on the square of the input size. Rule: `n --> 2n` means `t --> 4t`.
  - Exponential: the amount of time changes with a power of the input size. Rule: `n --> n + 1` means `t --> 2t`.

Q8: WWPD: Orders of Growth

What is the *worst case* (i.e. when `n` is prime) order of growth of `is_prime` in terms of `n`?

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

Linear ($\Theta(n)$).

Explanation: In the worst case, n is prime, and we have to execute the loop n - 2 times. Each iteration takes constant time (one conditional check and one return statement). Therefore, the total time is (n - 2) x constant, or simply linear.

What is the order of growth of `bar` in terms of `n`?

```
def bar(n):
    i, sum = 1, 0
    while i <= n:
        sum += biz(n)
        i += 1
    return sum

def biz(n):
    i, sum = 1, 0
    while i <= n:
        sum += i**3
        i += 1
    return sum
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

Quadratic $\Theta(n2)$.

Explanation: The body of the while loop in `bar` is executed n times. Each iteration, one call to `biz(n)` is made. Note that n never changes, so this call takes the same time to run each iteration. Taking a look at `biz`, we see that there is another while loop. Be careful to note that although the term being added to `sum` is cubed (`i**3`), `i` itself is only incremented by 1 in each iteration. This tells us that this while loop also executes n times, with each iteration taking constant time, so the total time of `biz(n)` is n x constant, or linear. Knowing that each call to `biz(n)` takes linear time, we can conclude that each iteration of the while loop in `bar` is linear. Therefore, the total runtime of `bar(n)` is quadratic.

What is the order of growth of `foo` in terms of `n`, where `n` is the length of `lst`? Assume that slicing a list and calling `len` on a list can both be done in constant time. Write your answer in Θ notation.

```
def foo(lst, i):
    mid = len(lst) // 2
    if mid == 0:
        return lst
    elif i > 0:
        return foo(lst[mid:], -1)
    else:
        return foo(lst[:mid], 1)
```

Logarithmic $\Theta(\log(n))$.

Explanation: A single recursive call is made in the body of `foo` on half the input list (either the first half or the second half depending on the input flag `i`). The base case is executed when the list either is empty or has only one element. We start with an n element list and halve the list until there is at most 1 element, which means there will be $\log(n)$ total calls. Each call, constant work is done if we ignore the recursive call. The total runtime is then $\log(n)$ * (1).

Note: We simplified this problem by assuming that slicing a list takes constant time. In reality, this operation is a bit more nuanced and may take linear time. As an additional exercise, try determining the order of growth of this function if we assuming slicing takes linear time.