CS 61A
Fall 2023

Linked Lists, Efficiency

Discussion 8: October 18, 2023

Note: For formal explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

Linked Lists

A **linked list** is a recursive data structure that represents sequences. The `Link` class implements linked lists in Python. Each `Link` instance has a `first` attribute (which stores the first value of the linked list) and a `rest` attribute (which points to the rest of the linked list). An empty linked list is represented as `Link.empty`, and a non-empty linked list is represented as a `Link` instance.

Q1: WWPD: Linked Lists

What would Python display? Try drawing the box-and-pointer diagram if you get stuck!

```
>>> link = Link(1, Link(2, Link(3)))
>>> link.first
```

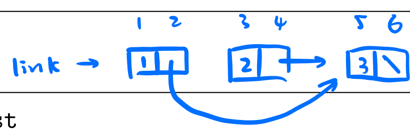
1

```
>>> link.rest.first
```

2

```
>>> link.rest.rest.rest is Link.empty
```

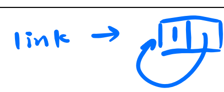
True




```
>>> link.rest = link.rest.rest
>>> link.rest.first
```

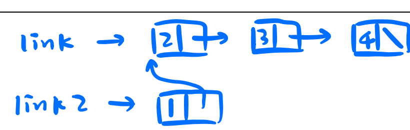
3

```
>>> link = Link(1)
>>> link.rest = link
>>> link.rest.rest.rest.rest.first
```



1

```
>>> link = Link(2, Link(3, Link(4)))
>>> link2 = Link(1, link)
>>> link2.rest.first
```



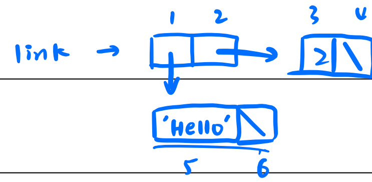
2

```
>>> link = Link(1000, 2000)
```

Error

```
>>> link = Link(1000, Link())
```

Error




```
>>> link = Link(Link("Hello"), Link(2))
>>> link.first
```

Link("Hello")

>>> obj

↕

>>> print(repr(obj))

```
>>> link = Link(Link("Hello"), Link(2))
>>> link.first.rest is Link.Empty
```

True

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Q2: Sum Nums

Write a function `sum_nums` that receives a linked list `s` and returns the sum of its elements. You may assume the elements of `s` are all integers. Try to implement `sum_nums` with recursion!

```
def sum_nums(s):
    """
    Returns the sum of the elements in s.

    >>> a = Link(1, Link(6, Link(7)))
    >>> sum_nums(a)
    14
    """
    "*** YOUR CODE HERE ***"
    if s is Link.empty:
        return 0
    return s.first + sum_nums(s.rest)

# You can use more space on the back if you want
```

```
sum = 0
while s is not Link.empty:
    sum += s.first
    s = s.rest
return sum
```

Q3: Remove All

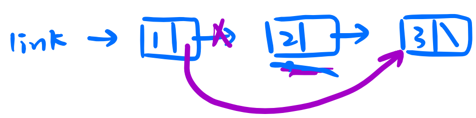
Write a function `remove_all` that takes a linked list and a `value` as input. This function mutates the linked list by removing all nodes that store `value`.

You may assume the first element of the linked list is not equal to `value`. You should mutate the input linked list; `remove_all` does not return anything.

```
def remove_all(link, value):
    """Removes all nodes in link that contain value. The first element of
    link is never equal to value.

    >>> l1 = Link(0, Link(2, Link(2, Link(3, Link(1, Link(2, Link(3)))))))
    >>> print(l1)
    <0 2 2 3 1 2 3>
    >>> remove_all(l1, 2)
    >>> print(l1)
    <0 3 1 3>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    >>> remove_all(l1, 3)
    >>> print(l1)
    <0 1>
    """
    "*** YOUR CODE HERE ***"
    if link is Link.empty or
        link.rest is Link.empty:
        return
    if link.rest.first == value:
        link.rest = link.rest.rest
        remove_all( link , value)
    else:
        remove_all(link.rest, value)

# You can use more space on the back if you want
```



link.rest = link.rest.rest

remove_all(link, 2)

```
while link is not Link.empty or
      link.rest is not Link.empty:
    if link.rest.first == value:
        link.rest = link.rest.rest
    else:
        link = link.rest
return
```

Q4: Flip Two

Write a recursive function `flip_two` that receives a linked list `s` and flips every pair of values in `s`.

```
def flip_two(s):
    """
    Flips every pair of values in 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)))))
    """
    "*** YOUR CODE HERE ***"









"*** YOUR CODE HERE ***"
```

Q5: Make Circular

Write a function `make_circular` that takes in a non-circular, non-empty linked list `s` and mutates `s` so that it becomes circular.

```
def make_circular(s):
    """Mutates linked list s into a circular linked list.

    >>> lnk = Link(1, Link(2, Link(3)))
    >>> make_circular(lnk)
    >>> lnk.rest.first
    2
    >>> lnk.rest.rest.first
    3
    >>> lnk.rest.rest.rest.first
    1
    >>> lnk.rest.rest.rest.rest.first
    2
    """
    "*** YOUR CODE HERE ***"







# You can use more space on the back if you want
```

Efficiency

A function's runtime complexity is a measure of how the runtime of the function changes as its input changes. A function `f(n)` has...

- constant runtime if the runtime of `f` does not depend on `n`. Its runtime is $\Theta(1)$.
- logarithmic runtime if the runtime of `f` is proportional to `log(n)`. Its runtime is $\Theta(\log(n))$.
- linear runtime if the runtime of `f` is proportional to `n`. Its runtime is $\Theta(n)$.
- quadratic runtime if the runtime of `f` is proportional to `n^2`. Its runtime is $\Theta(n^2)$.
- exponential runtime if the runtime of `f` is proportional to `b^n`, for some constant `b`. Its runtime is $\Theta(b^n)$.

Q6: WWPD: Orders of Growth

What is the *worst-case* runtime of `is_prime`?

```python
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

What is the order of growth of the runtime of `bar(n)` with respect to `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

What is the order of growth of the runtime of `foo` in terms of `n`, where `n` is the length of `lst`? Assume that slicing a list and evaluating `len(lst)` take constant time.

Express your answer with 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)
```

Appendix: Explanation of Material

Linked Lists

The `Link` class implements linked lists in Python. Each `Link` instance has a `first` attribute (which stores the first value of the linked list) and a `rest` attribute (which points to the rest of the linked list).

```
class Link:
    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 is not Link.empty:
            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) + '>'
```

An empty linked list is represented as `Link.empty`, and a non-empty linked list is represented as a `Link` instance.

The `rest` attribute of a `Link` instance is always another linked list! When `Link` instances are linked via their `rest` attributes, a sequence is formed.

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

Efficiency

Throughout this class, we have mainly focused on *correctness* — whether a program produces the correct output. However, computer scientists are also interested in creating *efficient* solutions to problems. One way to quantify efficiency is to determine how a function's *runtime* changes as its input changes. In this class, we measure a function's runtime by the number of operations it performs.

A function `f(n)` has...

- constant runtime if the runtime of `f` does not depend on `n`. Its runtime is Θ(`1`).
- logarithmic runtime if the runtime of `f` is proportional to `log(n)`. Its runtime is Θ(`log(n)`).
- linear runtime if the runtime of `f` is proportional to `n`. Its runtime is Θ(`n`).
- quadratic runtime if the runtime of `f` is proportional to `n^2`. Its runtime is Θ(`n^2`).
- exponential runtime if the runtime of `f` is proportional to `b^n`, for some constant `b`. Its runtime is Θ(`b^n`).

Example 1: It takes a single multiplication operation to compute `square(1)`, and it takes a single multiplication operation to compute `square(100)`. In general, calling `square(n)` results in a *constant* number of operations that does not vary according to `n`. We say `square` has a runtime complexity of Θ(1).

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: It takes a single multiplication operation to compute `factorial(1)`, and it takes 100 multiplication operations to compute `factorial(100)`. As `n` increases, the runtime of `factorial` increases *linearly*. We say `factorial` has a runtime complexity of Θ(`n`).

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)
```

Evaluating `bar(1)` results in a single `print` call, while evalulating `bar(100)` results in 10,000 `print` calls. As `n` increases, the runtime of `bar` increases *quadratically*. We say `bar` 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: Consider the following function:

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

Evaluating `rec(1)` results in a single addition operation. Evaluating `rec(4)` results in `2^4 - 1` = 15 addition operations, as shown by the diagram below.

During the evaulation of `rec(4)`, there are two calls to `rec(3)`, four calls to `rec(2)`, eight calls to `rec(1)`, and 16 calls to `rec(0)`.

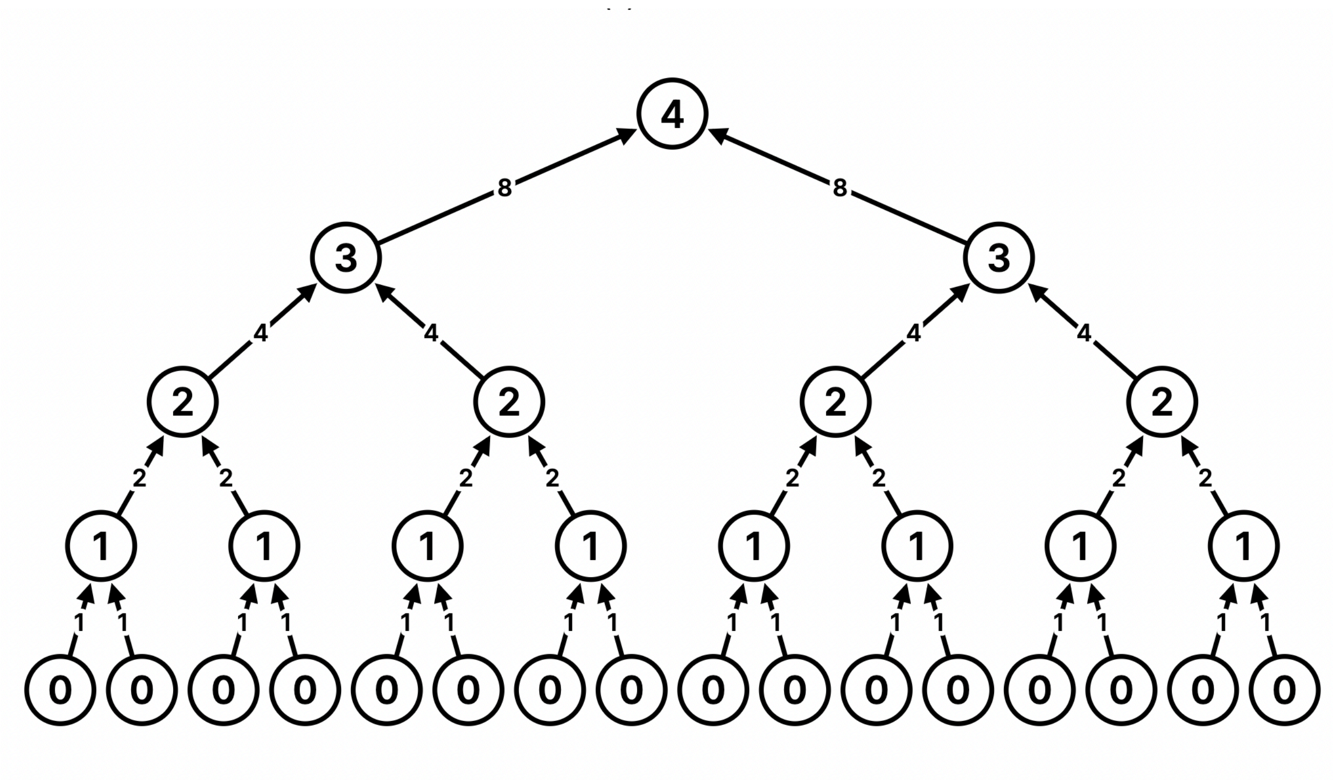
So we have eight instances of `rec(0) + rec(0)`, four instances of `rec(1) + rec(1)`, two instances of `rec(2) + rec(2)`, and a single instance of `rec(3) + rec(3)`, for a total of $1 + 2 + 4 + 8 = 15$ addition operations.

As `n` increases, the runtime of `rec` increases *exponentially*. In particular, the runtime of `rec` approximately doubles when we increase `n` by 1. We say `rec` has a runtime complexity of Θ(`2^n`).

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 - 1

Tips for finding the order of growth of a function's runtime:

- If the function is recursive, determine the number of recursive calls and the runtime of each recursive call.
- If the function is iterative, determine the number of inner loops and the runtime of each loop.
- Ignore coefficients. A function that performs `n` operations and a function that performs `100 * n` operations are both linear.
- Choose the largest order of growth. If the first part of a function has a linear runtime and the second part has a quadratic runtime, the overall function has a quadratic runtime.
- In this course, we only consider constant, logarithmic, linear, quadratic, and exponential runtimes.



Above: Call structure of rec(4).