CS 61A
Summer 2026

Linked Lists, Efficiency

Discussion 8: July 23, 2026

Linked Lists

A linked list is a `Link` object or `Link.empty`.

You can mutate a `Link` object `s` in two ways:

- Change the first element with `s.first = ...`
- Change the rest of the elements with `s.rest = ...`

```
class Link:
    """A linked list is either a Link object or Link.empty

    >>> s = Link(3, Link(4, Link(5)))
    >>> s.rest
    Link(4, Link(5))
    >>> s.rest.rest.rest is Link.empty
    True
    >>> s.rest.first * 2
    8
    >>> print(s)
    (3 4 5)
    """
    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) + ')'
```

Q1: Strange Loop

Here is a `Link` object with a cycle that represented an infinite repeating list of 1's. It may be useful as a reference doctest to help you solve the problem.

```
>>> ones = Link(1)
>>> ones.rest = ones
>>> [ones.first, ones.rest.first, ones.rest.rest.first, ones.rest.rest.rest.first]
[1, 1, 1, 1]
>>> ones.rest is ones
True
```

Implement `strange_loop`, which takes no arguments and returns a `Link` object `s` for which `s.rest.first.rest` is `s`.

Draw a picture of the linked list you want to create, then write code to create it.

```
def strange_loop() -> Link:
    """Return a Link s for which s.rest.first.rest is s.

    >>> s = strange_loop()
    >>> s.rest.first.rest is s
    True
    """
    s = Link(1, Link(Link(2)))
    s.rest.first.rest = s
    return s
```

Q2: Sum Two Ways

Implement both `sum_rec` and `sum_iter`. Each one takes a linked list of numbers `s` and a non-negative integer `k` and returns the sum of the first `k` elements of `s`. If there are fewer than `k` elements in `s`, all of them are summed. If `k` is 0 or `s` is empty, the sum is 0.

Use recursion to implement `sum_rec`. Don't use recursion to implement `sum_iter`; use a `while` loop instead.

To get started on the recursive implementation, consider the example `a = Link(1, Link(6, Link(8)))`, and the call `sum_rec(a, 2)`. Write down the recursive call to `sum_rec` that would help compute `sum_rec(a, 2)`. Then, write down what that recursive call should return. Discuss how this return value is useful in computing the return value of `sum_rec(a, 2)`.

```
def sum_rec(s: Link, k: int) -> int:
    """Return the sum of the first k elements in s.

    >>> a = Link(1, Link(6, Link(8)))
    >>> sum_rec(a, 2)
    7
    >>> sum_rec(a, 5)
    15
    >>> sum_rec(Link.empty, 1)
    0
    """
    # Use a recursive call to sum_rec; don't call sum_iter
    if k == 0 or s is Link.empty:
        return 0
    return s.first + sum_rec(s.rest, k - 1)

def sum_iter(s: Link, k: int) -> int:
    """Return the sum of the first k elements in s.

    >>> a = Link(1, Link(6, Link(8)))
    >>> sum_iter(a, 2)
    7
    >>> sum_iter(a, 5)
    15
    >>> sum_iter(Link.empty, 1)
    0
    """
    # Don't call sum_rec or sum_iter
    total = 0
    while k > 0 and s is not Link.empty:
        total, s, k = total + s.first, s.rest, k - 1
    return total
```

Q3: Overlap

Implement `overlap`, which takes two linked lists of numbers called `s` and `t` that are sorted in increasing order and have no repeated elements within each list. It returns the count of how many numbers appear in both lists.

This can be done in *linear* time in the combined length of `s` and `t` by always advancing forward in the linked list whose first element is smallest until both first elements are equal (add one to the count and advance both) or one list is empty (time to return).

```
def overlap(s: Link, t: Link) -> int:
    """For increasing s and t, count the numbers that appear in both.

    >>> a = Link(3, Link(4, Link(6, Link(7, Link(9, Link(10))))))
    >>> b = Link(1, Link(3, Link(5, Link(7, Link(8)))))
    >>> overlap(a, b)  # 3 and 7
    2
    >>> overlap(a.rest, b)  # just 7
    1
    >>> overlap(Link(0, a), Link(0, b))
    3
    """
    if s is Link.empty or t is Link.empty:
        return 0
    if s.first == t.first:
        return 1 + overlap(s.rest, t.rest)
    elif s.first < t.first:
        return overlap(s.rest, t)
    elif s.first > t.first:
        return overlap(s, t.rest)


def overlap_iterative(s, t):
    """For increasing s and t, count the numbers that appear in both.

    >>> a = Link(3, Link(4, Link(6, Link(7, Link(9, Link(10))))))
    >>> b = Link(1, Link(3, Link(5, Link(7, Link(8)))))
    >>> overlap(a, b)  # 3 and 7
    2
    >>> overlap(a.rest, b)  # just 7
    1
    >>> overlap(Link(0, a), Link(0, b))
    3
    """
    res = 0
    while s is not Link.empty and t is not Link.empty:
        if s.first == t.first:
            res += 1
            s = s.rest
            t = t.rest
        elif s.first < t.first:
            s = s.rest
        else:
            t = t.rest
    return res
```

Q4: Iterate in Order

Implement `iterate_in_order`, which takes two linked lists of numbers called `s` and `t` that are sorted in increasing order and have no repeated elements within each list. It returns a generator that iterates over all items in `s` and `t` in non-decreasing order.

```
def iterate_in_order(s: Link, t: Link):
    """For increasing s and t, yields the elements in s and t, in non-decreasing order.

    >>> a = Link(3, Link(4, Link(6, Link(7, Link(9, Link(10))))))
    >>> b = Link(1, Link(3, Link(5, Link(7, Link(8)))))
    >>> t = iterate_in_order(a, b)
    >>> for item in t:
    ...     print(item)
    1
    3
    3
    4
    5
    6
    7
    7
    8
    9
    10
    >>> t = iterate_in_order(Link.empty, b)
    >>> for item in t:
    ...      print(item)
    1
    3
    5
    7
    8
    """
    while s is not Link.empty or t is not Link.empty:
        if s is Link.empty:
            yield t.first
            t = t.rest
        elif t is Link.empty or s.first <= t.first:
            yield s.first
            s = s.rest
        else:
            yield t.first
            t = t.rest
```

Efficiency

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.

Q5: The First Order...of Growth

What is the efficiency of `rey`?

```
def rey(finn):
    poe = 0
    while finn >= 2:
        poe += finn
        finn = finn / 2
    return
```

Choose one of:

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

Solution: Logarithmic, because our while loop iterates at most log(`finn`) times, due to `finn` being halved in every iteration. This is commonly known as $\Theta(\log(\texttt{finn}))$ runtime. Another way of looking at this if you duplicate the input, we only add a single iteration to the time, which also indicates logarithmic.

What is the efficiency of `mod_7`?

```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

Choose one of:

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

Solution: Constant, since in the worst case scenario our function `mod_7` will require 6 recursive calls to reach the base case. Consider the worst case where we have an input `n` such that our first call to `mod_7` evaluates `n % 7` as `6`. Each recursive call will decrement `n` by `1`, allowing us to eventually reach the base case of returning `0` in 6 recursive calls (`n` will range from 0 to 6). Since the growth of the computation is independent of the input, we say this is constant, which is commonly known as a $\Theta(1)$ runtime.

Q6: Efficiency Practice

Choose the term that fills in the blank for the functions defined below: `<function>` runs in `____` time in the length of its input.

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

Assume that `len` runs in constant time and `all` runs in linear time in the length of its input. Selecting an element of a list by its index requires constant time. Constructing a range requires constant time.

```
def count_partitions(n, m):
    """Counts the number of partitions of a positive integer n,
    using parts up to size m."""
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m

def is_palindrome(s):
    """Return whether a list of numbers s is a palindrome."""
    return all([s[i] == s[len(s) - i - 1] for i in range(len(s))])

def binary_search(lst, n):
    """Takes in a sorted list lst and returns the index where integer n
    is contained in lst. Returns -1 if n does not exist in lst."""
    low = 0
    high = len(lst)
    while low <= high:
        middle = (low + high) // 2
        if lst[middle] == n:
            return middle
        elif n < lst[middle]:
            high = middle - 1
        else:
            low = middle + 1
    return -1
```

The `is_palindrome` question was reformatted from question 6(d) on fall 2019's final.

count_partitions: exponential.

`count_partitions` takes two inputs `n` and `m` so the runtime depends on both inputs. However, when determining what the order of growth is, we're mostly concerned with what happens to the runtime for larger values of `n` and `m`.

The conditional statements such as `if n == 0` are logical comparisons that take constant time. Therefore, what affects our runtime the most are the number of recursive calls we to `count_partitions`. For every function call to `count_partitions` we make two more recursive calls; one that uses `m` as a partition, and one that does not.

Visualizing the tree of recursive calls for this function, we'll see that the number of function calls at each level doubles. Thus, the runtime for `count_partitions` is exponential.

is_palindrome: linear.

We're interested in seeing how the function runs in relation to the length of its input, which is `len(s)`.

In `is_palindrome`, it takes constant time to calculate `len(s)` and then to construct a range from 0 to `len(s)` (exclusive).

For each element in this range, we will select two elements from `s` (`s[i]` and `s[len(s)-i-1]`) and compare them, which takes some constant time for each element. Once we've done this for all the elements, we will have built up the input list to `all` in linear time in relation to the length of `is_palindrome`'s input.

We assume that `all` runs in linear time in the length of *its* input, which is the length of the list we've just built and the same as the length of `is_palindrome`'s input.

Overall, `is_palindrome` will therefore take linear time in relation to the length of its input.

binary_search: logarithmic.

The `binary_search` algorithm's runtime is dependent on the length of `lst` because we are searching through it to find some integer `n`.

Again, assuming `len(lst)` and logical comparisons take constant time to calculate, we look at how many iterations of the while loop we might go through before completing the function call. First we check the element at the middle of our sorted list. If `lst[middle]` is not equal to `n`, the function determines whether `lst[middle]` is less than or greater than `n` and narrows our search down to either the lower half or upper half of `lst`, respectively.

For every iteration of the while loop, `binary_search` cuts down the number of elements in the input `lst` we are searching through by half. Thus, the maximum number of iterations for the while loop before returning an index or `-1` is log(len(lst)) in base 2. `binary_search` takes logarithmic time in relation to the length of its input `lst`.