CS 61A
Fall 2023

Midterm Review

Discussion 9: October 25, 2023

This discussion is optional: that means that attendance is not expected, and you will receive no credit for attending this discussion.

Your TA will not be able to get to all of the problems on this worksheet so feel free to work through the remaining problems on your own. Bring any questions you have to office hours or post them on Ed. Good luck on the midterm!

Fun

Q1: Around and Around

Berkeley students can juggle a lot of responsibilities. But can they juggle balls? Your TA has brought some tennis balls for you to use. Let's learn to juggle!

0. Get comfortable Get into pairs, and introduce yourselves if you don't already know each other. Take a ball and toss it from hand to hand in whatever way feels right — get a feel for things!

1. One ball Now that you're comfortable with the feel of the ball, let's practice our one ball toss. Take a single ball and toss it from one hand to the other. The ball should reach its apex above your opposite shoulder, and you should catch it slightly outside of that shoulder.

> Getting a solid, high throw is crucial here! Don't cut corners!

You should feel like you're "scooping" inward and upward as you do the throws. This will help you transition from catching to throwing.

After you do this for a minute, have one partner stop juggling to observe the other and provide guidance and feedback. Then, switch roles.

2. Two balls Let's double the number of balls we're using. To get the two-ball throw down, start by tossing one ball from hand to hand. When that ball reaches its apex, throw the second ball!

> If you're having trouble throwing the second ball, don't even worry about catching the first one. Your body will know how to catch it. The throw is the hardest part! One tip is to say "left right" or "right left" as you throw the two balls.

Get comfortable throwing two balls, and consider switching up which hand you start with. After you do this for a minute, have one partner stop juggling to observe the other and provide guidance and feedback. Then, switch roles.

3. Three balls Let's add the final ball. Start with two balls in your preferred hand and one ball in your other hand. Throw the two balls as before, but now toss the third ball when the second one reaches its apex!

> Again, the throw is the hardest part! Consider saying "left right left" or "right left right" as you throw to get it into your head.

After you do this for a minute, have one partner stop juggling to observe the other and provide guidance and feedback. Then, switch roles.

4. Cascade Now, all you have to do is keep the cycle going! If you're still having trouble with this, feel free to take some balls home and practice.

Recursion

Q2: Paths List

(Adapted from Fall 2013) Fill in the blanks in the implementation of `paths`, which takes as input two positive integers `x` and `y`. It returns a list of paths, where each path is a list containing steps to reach `y` from `x` by repeated incrementing or doubling. For instance, we can reach 9 from 3 by incrementing to 4, doubling to 8, then incrementing again to 9, so one path is [3, 4, 8, 9].

```
def paths(x, y):
    """Return a list of ways to reach y from x by repeated
    incrementing or doubling.
    >>> paths(3, 5)
    [[3, 4, 5]]
    >>> sorted(paths(3, 6))
    [[3, 4, 5, 6], [3, 6]]
    >>> sorted(paths(3, 9))
    [[3, 4, 5, 6, 7, 8, 9], [3, 4, 8, 9], [3, 6, 7, 8, 9]]
    >>> paths(3, 3) # No calls is a valid path
    [[3]]
    >>> paths(5, 3) # There is no valid path from x to y
    []
    """
    if ___x == y___________________________________
        return ___[[y]]  (or  [[x]] )___________________
    elif ___x > y___________________________
        return ___[]______________________________
    else:
        a = ___paths(x+1, y)____________________
        b = ___paths(x*2, y)____________________
        return [[x]+path for path in a+b]____
```

paths (3,9)

increment ↙ ↘ double

a = paths (4,9)   b= paths (6,9)

[[4,5,6,7,8,9], [4,8,9]]   [[6,7,8,9]]

x+1 to y   x *2 to y

a+b

[[ ], [ ], [ ]]

[x]+ ↓

[a+b]

[[a], [b]] + [[c]]

↓

[[[a], [b], [c]]]

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

Trees

Q3: Widest Level

Write a function that takes a `Tree` object and returns the elements at the depth with the most elements.

In this problem, you may find it helpful to use the second optional argument to `sum`, which provides a starting value. All items in the sequence to be summed will be concatenated to the starting value. By default, start will default to 0, which allows you to sum a sequence of numbers. We provide an example of sum starting with a list, which allows you to concatenate items in a list.

```
def widest_level(t):
    """
    >>> sum([[1], [2]], [])
    [1, 2]
    >>> t = Tree(3, [Tree(1, [Tree(1), Tree(5)]),
    ...              Tree(4, [Tree(9, [Tree(2)])])])
    >>> widest_level(t)
    [1, 5, 9]
    """
    levels = []
    x = [t]
    while _____________:
        _____________
        __________ = sum(_______________________________, [])
    return max(levels, key=_________________________________)
```

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

Q4: Level Mutation Link

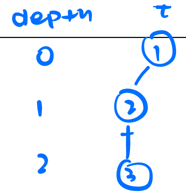
As a reminder, the depth of a node is how far away the node is from the root. We define this as the number of edges between the root to the node. As there are no edges between the root and itself, the root has depth 0.

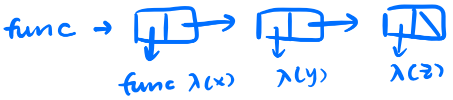
Given a tree `t` and a linked list of one-argument functions `funcs`, write a function that will mutate the labels of `t` using the function from `funcs` at the corresponding depth. For example, the label at the root node (with a depth of 0) will be mutated using the function at `funcs.first`. Assume all of the functions in `funcs` will be able to take in a label value and return a valid label value.

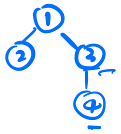
If `t` is a leaf and there are more than 1 functions in `funcs`, all of the remaining functions should be applied in order to the label of `t`. (See the doctests for an example.) If `funcs` is empty, the tree should remain unmodified.

```
def level_mutation_link(t, funcs):
    """Mutates t using the functions in the linked list funcs.

    >>> t = Tree(1, [Tree(2, [Tree(3)])])
    >>> funcs = Link(lambda x: x + 1, Link(lambda y: y * 5, Link(lambda z: z ** 2)))
    >>> level_mutation_link(t, funcs)
    >>> t
    Tree(2, [Tree(10, [Tree(9)])])
    >>> t2 = Tree(1, [Tree(2), Tree(3, [Tree(4)])])
    >>> level_mutation_link(t2, funcs)
    >>> t2
    Tree(2, [Tree(~~100~~10), Tree(15, [Tree(16)])])
    >>> t3 = Tree(1, [Tree(2)])
    >>> level_mutation_link(t3, funcs)
    >>> t3
    Tree(2, [Tree(100)])
    """ funcs == Link.empty
    if funcs is Link.empty:
        return
    t.label = funcs.first(t.label)
    remaining = funcs.rest
                                             !=
    if t.is_leaf() and remaining is not:Link.empty
        while remaining is not Link.empty
            t.label = remaining.first(t.label)
            remaining = remaining.rest
    for b in t.branches:
        level_mutation_link(b, remaining )
```







(2 * 5) ** 2 → 100

Lists and Mutability

Q5: Shuffle

Define a function `shuffle` that takes a sequence with an even number of elements (cards) and creates a new list that interleaves the elements of the first half with the elements of the second half.

To interleave two sequences `s0` and `s1` is to create a new sequence such that the new sequence contains (in this order) the first element of `s0`, the first element of `s1`, the second element of `s0`, the second element of `s1`, and so on.

> **Note:** If you're running into an issue where the special heart / diamond / spades / clubs symbols are erroring in the doctests, feel free to copy paste the below doctests into your file as these don't use the special characters and should not give an "illegal multibyte sequence" error.

```
def card(n):
    """Return the playing card numeral as a string for a positive n <= 13."""
    assert type(n) == int and n > 0 and n <= 13, "Bad card n"
    specials = {1: 'A', 11: 'J', 12: 'Q', 13: 'K'}
    return specials.get(n, str(n))

def shuffle(cards):
    """Return a shuffled list that interleaves the two halves of cards.

    >>> shuffle(range(6))
    [0, 3, 1, 4, 2, 5]
    >>> suits = ['H', 'D', 'S', 'C']
    >>> cards = [card(n) + suit for n in range(1,14) for suit in suits]
    >>> cards[:12]
    ['AH', 'AD', 'AS', 'AC', '2H', '2D', '2S', '2C', '3H', '3D', '3S', '3C']
    >>> cards[26:30]
    ['7S', '7C', '8H', '8D']
    >>> shuffle(cards)[:12]
    ['AH', '7S', 'AD', '7C', 'AS', '8H', 'AC', '8D', '2H', '8S', '2D', '8C']
    >>> shuffle(shuffle(cards))[:12]
    ['AH', '4D', '7S', '10C', 'AD', '4S', '7C', 'JH', 'AS', '4C', '8H', 'JD']
    >>> cards[:12]  # Should not be changed
    ['AH', 'AD', 'AS', 'AC', '2H', '2D', '2S', '2C', '3H', '3D', '3S', '3C']
    """
    assert len(cards) % 2 == 0, 'len(cards) must be even'
    half = _______________
    shuffled = []
    for i in _____________:
        _________________
        _________________
    return shuffled
```

Efficiency

Q6: Bonk

Describe the order of growth of the function below.

```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

Sum = sum + n

sum += 1

constant

# of loops * # of operations in each loop

* log n   constant

n

$\frac{n}{2}$

$\frac{n}{2^2}$

⋮

$\frac{n}{2^x} = 1$

$\Rightarrow x = \log_2 n$

Choose one of:

- Constant
- [✓ Logarithmic]
- Linear
- Quadratic
- Exponential
- None of these

Q7: Pow

Write the following function so it runs in (log k) time.

> **Hint:** This can be done using a procedure called repeated squaring.

```
def lgk_pow(n,k):
    """Computes n^k.

    >>> lgk_pow(2, 3)
    8
    >>> lgk_pow(4, 2)
    16
    >>> a = lgk_pow(2, 100000) # make sure you have log time
    """
    "*** YOUR CODE HERE ***"
```

Generators

Q8: Yield, Fibonacci!

Implement `fibs`, a generator function that takes a one-argument pure function `f` and yields all Fibonacci numbers `x` for which `f(x)` returns a true value. The Fibonacci numbers begin with 0 and then 1. Each subsequent Fibonacci number is the sum of the previous two. Yield the Fibonacci numbers in order.

```
def fibs(f):
    """Yield all Fibonacci numbers x for which f(x) is a true value.
    >>> odds = fibs(lambda x: x % 2 == 1)
    >>> [next(odds) for i in range(10)]
    [1, 1, 3, 5, 13, 21, 55, 89, 233, 377]
    >>> bigs = fibs(lambda x: x > 20)
    >>> [next(bigs) for i in range(10)]
    [21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
    >>> evens = fibs(lambda x: x % 2 == 0)
    >>> [next(evens) for i in range(10)]
    [0, 2, 8, 34, 144, 610, 2584, 10946, 46368, 196418]
    """
    n, m = 0, 1
    while _________:
        if _________:
            _________
        n, m = _________, _________
```

Q9: Partitions

Tree-recursive generator functions have a similar structure to regular tree-recursive functions. They are useful for iterating over all possibilities. Instead of building a list of results and returning it, just `yield` each result.

You'll need to identify a *recursive decomposition*: how to express the answer in terms of recursive calls that are simpler. Ask yourself what will be yielded by a recursive call, then how to use those results.

Definition. For positive integers `n` and `m`, a *partition* of `n` using parts up to size `m` is an addition expression of positive integers up to `m` in non-decreasing order that sums to `n`.

Implement `partition_gen`, a generator functon that takes positive `n` and `m`. It yields the partitions of `n` using parts up to size `m` as strings.

Reminder: For the `partitions` function we studied in lecture (video), the recursive decomposition was to enumerate all ways of partitioning `n` using at least one `m` and then to enumerate all ways with no `m` (only `m-1` and lower).

```
def partition_gen(n, m):
    """Yield the partitions of n using parts up to size m.

    >>> for partition in sorted(partition_gen(6, 4)):
    ...     print(partition)
    1 + 1 + 1 + 1 + 1 + 1
    1 + 1 + 1 + 1 + 2
    1 + 1 + 1 + 3
    1 + 1 + 2 + 2
    1 + 1 + 4
    1 + 2 + 3
    2 + 2 + 2
    2 + 4
    3 + 3
    """
    assert n > 0 and m > 0
    if n == m:
        yield ____
    if n - m > 0:
        "*** YOUR CODE HERE ***"




    if m > 1:
        "*** YOUR CODE HERE ***"
```

OOP

Q10: Mint

A mint is a place where coins are made. In this question, you'll implement a `Mint` class that can output a `Coin` with the correct year and worth.

- Each `Mint` *instance* has a `year` stamp. The `update` method sets the `year` stamp of the instance to the `present_year` *class attribute* of the `Mint` *class.*
- The `create` method takes a subclass of `Coin` (*not* an instance!), then creates and returns an *instance* of that class stamped with the `mint`'s year (which may be different from `Mint.present_year` if it has not been updated.)
- A `Coin`'s `worth` method returns the `cents` value of the coin plus one extra cent for each year of age *beyond 50*. A coin's age can be determined by subtracting the coin's year from the `present_year` class attribute of the `Mint` class.

```
class Mint:
    """A mint creates coins by stamping on years.

    The update method sets the mint's stamp to Mint.present_year.

    >>> mint = Mint()
    >>> mint.year
    2023
    >>> dime = mint.create(Dime)
    >>> dime.year
    2023
    >>> Mint.present_year = 2103  # Time passes
    >>> nickel = mint.create(Nickel)
    >>> nickel.year     # The mint has not updated its stamp yet
    2023
    >>> nickel.worth()  # 5 cents + (80 - 50 years)
    35
    >>> mint.update()   # The mint's year is updated to 2102
    >>> Mint.present_year = 2178     # More time passes
    >>> mint.create(Dime).worth()    # 10 cents + (75 - 50 years)
    35
    >>> Mint().create(Dime).worth()  # A new mint has the current year
    10
    >>> dime.worth()     # 10 cents + (155 - 50 years)
    115
    >>> Dime.cents = 20  # Upgrade all dimes!
    >>> dime.worth()     # 20 cents + (155 - 50 years)
    125
    """
    present_year = 2023

    def __init__(self):
        self.update()

    def create(self, coin):
        "*** YOUR CODE HERE ***"




    def update(self):
        "*** YOUR CODE HERE ***"
```

```
class Coin:
    cents = None # will be provided by subclasses, but not by Coin itself

    def __init__(self, year):
        self.year = year

    def worth(self):
        "*** YOUR CODE HERE ***"



class Nickel(Coin):
    cents = 5

class Dime(Coin):
    cents = 10
```

Linked Lists

Q11: Every Other

Implement `every_other`, which takes a linked list `s`. It mutates `s` such that all of the odd-indexed elements (using 0-based indexing) are removed from the list. For example:

```
>>> s = Link('a', Link('b', Link('c', Link('d'))))
>>> every_other(s)
>>> s.first
'a'
>>> s.rest.first
'c'
>>> s.rest.rest is Link.empty
True
```

If `s` contains fewer than two elements, `s` remains unchanged.

Do not return anything! `every_other` should mutate the original list.

```
def every_other(s):
    """Mutates a linked list so that all the odd-indiced elements are removed
    (using 0-based indexing).

    >>> s = Link(1, Link(2, Link(3, Link(4))))
    >>> every_other(s)
    >>> s
    Link(1, Link(3))
    >>> odd_length = Link(5, Link(3, Link(1)))
    >>> every_other(odd_length)
    >>> odd_length
    Link(5, Link(1))
    >>> singleton = Link(4)
    >>> every_other(singleton)
    >>> singleton
    Link(4)
    """
    "*** YOUR CODE HERE ***"
```

Q12: Insert

Implement a function `insert` that takes a `Link`, a `value`, and an `index`, and inserts the `value` into the `Link` at the given `index`. You can assume the linked list already has at least one element. Do not return anything – `insert` should mutate the linked list.

> **Note**: If the index is out of bounds, you should raise an `IndexError` with: raise IndexError('Out of bounds!')

```
def insert(link, value, index):
    """Insert a value into a Link at the given index.

    >>> link = Link(1, Link(2, Link(3)))
    >>> print(link)
    <1 2 3>
    >>> other_link = link
    >>> insert(link, 9001, 0)
    >>> print(link)
    <9001 1 2 3>
    >>> link is other_link # Make sure you are using mutation! Don't create a new linked
    list.
    True
    >>> insert(link, 100, 2)
    >>> print(link)
    <9001 1 100 2 3>
    >>> insert(link, 4, 5)
    Traceback (most recent call last):
        ...
    IndexError: Out of bounds!
    """
    "*** YOUR CODE HERE ***"
```