Exam Prep 3: Recursion, Tree Recursion

Students from past semesters wanted more content and structured time to prepare for exams. Exam Prep sections are a way to solidify your understanding of the week's materials. The problems are typically designed to be a bridge between discussion/lab/homework difficulty and exam difficulty.

Reminder: There is nothing to turn in and there is no credit given for attending Exam Prep Sections.

We try to make these problems **exam level** , so you are not expected to be able to solve them coming straight from lecture without additional practice. To get the most out of Exam Prep, we recommend you **try these problems first on your own** before coming to the Exam Prep section, where we will explain how to solve these problems while giving tips and advice for the exam. Do not worry if you struggle with these problems, **it is okay to struggle while learning**.

You can work with anyone you want, including sharing solutions. We just ask you don't spoil the problems for anyone else in the class. Thanks!

You may only put code where there are underscores for the codewriting questions.

You can test your functions on their doctests by clicking the red test tube in the top right corner after clicking Run in 61A Code. Passing the doctests is not necessarily enough to get the problem fully correct. You must fully solve the stated problem.

We recommending reading sections 1.7 from the textbook for these problems. We also recommend watching the February 1 and February 3 lectures on recursion and tree recursion, respectively.

Test your work! For example, for `is_palindrome` , you can type `test(is_palindrome)` in the python interpreter you get once you click Run in 61A Code to verify if you pass the doctests or not.

IMPORTANT: you may not use any string operations other than indexing, `len` and slicing. Specifically, you may not call `reversed` or index with a negative step size.

Here is a brief survey of string operations useful to this worksheet. Given a string `s = 'abcdefg'`,

- `len(s)` evaluates to the length of `s`, in this case `6`
- `s[0]` evaluates to the leftmost character of `s`, in this case `'a'`
- `s[-1]`, or alternatively `s[len(s) - 1]`, evaluates to the rightmost character of `s`, in this case `g`
- `s[a:b]` evaluates to a slice of `s` from position `a` to just before position `b`, zero-indexed for both. So `s[2:5]` would give `cde`
- `s[1:]` evaluates to a slice of `s` from position `1` until the end, in this case `bcdefg`
- `s[:-1]`, or alternatively `s[:len(s)-1]`, evaluates to a slice of `s` from the beginning until one position before the end, in this case `abcdef`
- Equality can be used as normal. For example, `s == 'abcdefg'` evaluates to `True`, and `s == 'egg'` evaluates to `False`.

Q1: 'Tis it?

Difficulty: ☆

A palindrome is a string that remains identical when reversed. Given a string `s`, `is_palindrome` should return whether or not `s` is a palindrome.

IMPORTANT: Please use the template for this problem; if you have spare time, try to solve the problem using iteration without the template.

```
1   def is_palindrome(s):
2       """
3       >>> is_palindrome("tenet")
4       True
5       >>> is_palindrome("tenets")
6       False
7       >>> is_palindrome("raincar")
8       False
9       >>> is_palindrome("")
10      True
11      >>> is_palindrome("a")
12      True
13      >>> is_palindrome("ab")
14      False
15      """
16      if s == s[::-1]:
17          return True
18      return False
19
20
```

Q2: Greatest Pals

Difficulty: ☆☆

A *substring* of `s` is a sequence of consecutive letters within `s`. Given a string `s`, `greatest_pal` should return the longest palindromic substring of `s`. If there are multiple palindromic substrings of greatest length, then return the leftmost one. **You may use `is_palindrome`.**

IMPORTANT: For this problem, each starter code template is just a suggestion. We recommend that you use the first, but feel free to modify it, try one of the other two or write your own if you'd like to. Comment out the other versions of the function to run doctests.

```
    """
    >>> greatest_pal("tenet")
    'tenet'
    >>> greatest_pal("tenets")
    'tenet'
    >>> greatest_pal("stennet")
    'tennet'
    >>> greatest_pal("25 racecars")
    'racecar'
    >>> greatest_pal("abc")
    'a'
    >>> greatest_pal("")
    ''
    """
    if is_palindrome(s):
        return s
    left, right = greatest_pal(s[:-1]), greatest_pal(s[1:])
    if len(left) >= len(right):
        return left
    return right

def greatest_pal(s):
    """
```

```
        >>> greatest_pal("tenet")
        'tenet'
        >>> greatest_pal("tenets")
        'tenet'
        >>> greatest_pal("stennet")
        'tennet'
        >>> greatest_pal("25 racecars")
        'racecar'
        >>> greatest_pal("abc")
        'a'
        >>> greatest_pal("")
        ''
        """
        def helper(a, b, c):
            if a > len(s):
                return c
            elif b > len(s) - a:
                return helper(a+1, 0, c)
            elif is_palindrome(s[b:b+a]) and a > len(c):
                c = s[b:b+a]
            return helper(a, b+1, c)
        return helper(1, 0, "")

    def greatest_pal(s):
        """
        >>> greatest_pal("tenet")
        'tenet'
        >>> greatest_pal("tenets")
        'tenet'
        >>> greatest_pal("stennet")
        'tennet'
        >>> greatest_pal("25 racecars")
        'racecar'
        >>> greatest_pal("abc")
        'a'
        >>> greatest_pal("")
        ''
        """
```

```
        def helper(a, b):
            if b > len(s) - a:
                return helper(a-1, 0)
            elif is_palindrome(s[b:b+a]):
                return s[b:b+a]
            return helper(a, b+1)
        return helper(len(s), 0)
```

Q3: Wait, It's All Palindromes?

Difficulty: ☆☆☆

Given a string `s`, return the longest palindromic substring of `s`. If there are multiple palindromes of greatest length, then return the leftmost one. **You may not use `is_palindrome`.**

Hint: Given equivalent values `a` and `b`, `max(a, b)` will evaluate to `a`. You may also find the `key` argument to `max` helpful.

```
def greatest_pal_two(s):
    """
    >>> greatest_pal_two("tenet")
    'tenet'
    >>> greatest_pal_two("tenets")
    'tenet'
    >>> greatest_pal_two("stennet")
    'tennet'
    >>> greatest_pal_two("abc")
    'a'
    >>> greatest_pal_two("")
    ''
    """
    for i in range(len(s) // 2):
        if s[i] != s[-(i+1)]:
            return max(greatest_pal_two(s[:-1]), greatest_pal_two(s[1:]), key=len)
    return s

```

Just for Fun

This is a challenge problem and not reflective of exam difficulty. We will not be going over this problem in examprep section, but we will be releasing solutions.

Q4: All-Ys Has Been

Difficulty: 😨

Given mystery function `Y`, complete `fib` and `is_pal` so that the given doctests work correctly. When `Y` is called on `fib`, it should return a function which takes a positive integer `n` and returns the `n` th Fibonacci number.

Similarly, when `Y` is called on `is_pal_maker` it should return a function `is_pal` that takes a string `s` and returns whether `s` is a palindrome.

Hint: You may use the ternary operator `if <bool-exp> <a> else <b>`, which evaluates to `<a>` if `<bool-exp>` is truthy and evaluates to `<b>` if `<bool-exp>` is false-y.

```
3   is_pal_maker = lambda f: lambda r: True if len(r) <= 1 else r[0] == r[-1] and f(r[1:-1])
4
5   fib = Y(fib_maker)
6   is_pal = Y(is_pal_maker)
7
8   # This code sets up doctests for fib and is_pal. Run test(fib) and test(is_pal) to check your implemen
9
10  fib.__name__ = 'fib'
11  fib.__doc__="""Given n, returns the nth Fibonacci nuimber.
12
13  >>> fib(0)
14  0
15  >>> fib(1)
16  1
17  >>> fib(2)
18  1
19  >>> fib(3)
20  2
21  >>> fib(4)
22  3
23  >>> fib(5)
```

```
    5
    """

    is_pal.__name__ = 'is_pal'
    is_pal.__doc__="""Returns whether or not an input string s is a palindrome.

    >>> is_pal('tenet')
    True
    >>> is_pal('tenets')
    False
    >>> is_pal('ab')
    False
    >>> is_pal('')
    True
    >>> is_pal('a')
    True
    """

```