CS 61A Fall 2020

Structure and Interpretation of Computer Programs

Quiz 3

INSTRUCTIONS

- Please review this worksheet before the exam prep session. Coming prepared will help greatly, as the TA will be live solving without allocating much time for individual work.
- Either Sean or Derek will be on video live solving these questions. The other TA will be answering questions in the chat. It is in your best interest to come prepared with **specific** questions.
- This is not graded, and you do not need to turn this in to anyone.
- Fall 2020 students: the boxes below are an artifact from more typical semesters to simulate exam environments. Obviously this doesn't apply to this semester's exams, but we just kept the fields to keep our materials looking professional :) Feel free to ignore them.
- For multiple choice questions, fill in each option or choice completely.
  - ☐ means mark **all options** that apply
  - ◯ means mark a **single choice**

Last name	
First name	
Student ID number	
CalCentral email (_@berkeley.edu)	
Discussion Section	___ ___ ___
All the work on this exam is my own. **(please sign)**	

1. **Warm-Up**

   (a) A palindrome is a string that remains identical when reversed. Given a string `s`, return whether or not it is a palindrome.

```
def is_palindrome(s):
    """
    >>> is_palindrome("tenet")
    True
    >>> is_palindrome("tenets")
    False
    >>> is_palindrome("")
    True
    >>> is_palindrome("a")
    True
    >>> is_palindrome("ab")
    False
    """
    if _____:
        return True
    return _____
```

s[1:len(s)-1]

len(s) <= 1

s[0] == s[-1] and is_palindrome(s[1:-1])

s[len(s)-1]

BASE CASE

→ When are you 100% sure it's a palindrome without having to do any more work?

2. **Great Pals**

(a) A *substring* of `s` is a sequence of consecutive letters within `s`. Given a string `s`, 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`.**

```
def greatest_pal(s):
    """
    >>> greatest_pal("tenet")
    'tenet'
    >>> greatest_pal("tenets")
    'tenet'
    >>> greatest_pal("stennet")
    'tennet'
    >>> greatest_pal("abc")
    'a'
    >>> greatest_pal("")
    ''
    """
    def helper(a, b, c):
        if _____ > _____:
            return _____
        elif _____ > _____:
            return _____
        elif _____ and _____:
            _____
        return _____
    return helper(1, 0, "")
```

a: starts at 1
b: starts at 0
c: string, starts at ""

while length <= len(s):
    while start <= len(s) - length

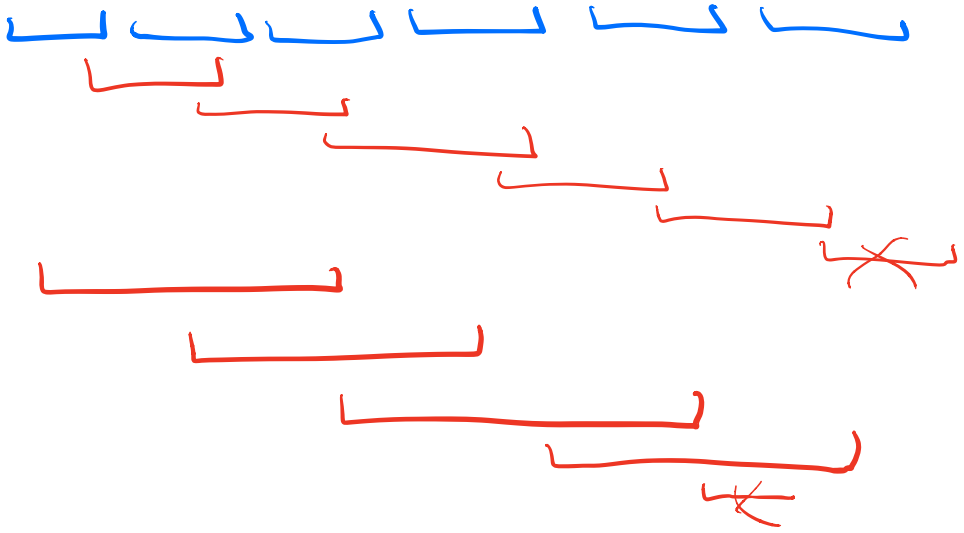
length — start index — what we've seen (longest palindrome so far)

exclude len(s)

b : b+a

b+a = len(c)

a
len(s)
c
b + a
len(s)
helper(a, b+1, c)
is_palindrome(s[b:b+a]) and a > len(c)
c = s[b:b+a]
helper(a, b+1, c)



3. **Take Two**

(a) A *substring* of `s` is a sequence of consecutive letters within `s`. Given a string `s`, 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`.**

```
def greatest_pal_simpler(s):
    """
    >>> greatest_pal_simpler("tenet")
    'tenet'
    >>> greatest_pal_simpler("tenets")
    'tenet'
    >>> greatest_pal_simpler("stennet")
    'tennet'
    >>> greatest_pal_simpler("abc")
    'a'
    >>> greatest_pal_simpler("")
    ''
    """
    def helper(a, b):
        if _____a == 0_____:
            return _____""_____
        elif _____a+b > len(s)_____:
            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)_____
```

4. **Wait, it's all palindromes?**

   (a) 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`.**

```
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("")
    ''
    """
    if _____:
        return _____
    for _____:
        if _____:
            return _____
    return s
```

s[-i-1]

len(s) <= 1

s

i in range(len(s)//2)

s[i] != s[len(s)-i-1]

max(greatest_pal_two(s[1:]), greatest_pal_two(s[:-1]), key=len)

>>> max(-2, 1, key=lambda x: x**2)
-2

5. **All-Ys Has Been**

(a) 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. `is_pal` should take a string and return whether it is a palindrome.

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

```
Y = lambda f: (lambda x: x(x))(lambda x: f(lambda z: x(x)(z)))

fib_maker = lambda f: lambda r: ____r if r<=1 else f(r-1) + f(r-2)_____

is_pal_maker = lambda f: lambda r: ____True if len(r)<=1 else r[0]==r[-1] and f(s[1:-1])

fib = Y(fib_maker)
is_pal = Y(is_pal_maker)

assert fib(0) == 0
assert fib(1) == 1
assert fib(2) == 1
assert fib(3) == 2
assert fib(4) == 3
assert fib(5) == 5

assert is_pal('tenet')
assert not is_pal('tenets')
assert not is_pal('ab')
assert is_pal('')
assert is_pal('a')
```


calls g on itself
(Since we're calling this on call_f_on_inner
so we have call_f_on_inner(call_f_on_inner))

```
def Y(f):
    def call_on_self(g):
        return g(g)
    def call_f_on_inner(x):
        def call_x_on_self_then_z(z):
            return x(x)(z)
        return f(call_x_on_self_then_z)
    return call_on_self(call_f_on_inner)
```

```
Y = lambda f: (lambda x: x(x))(lambda x: f(lambda z: x(x)(z)))
```

call_f_on_self_then_z

fib = Y(fib_maker)

fib_maker = λf: λr: ____?

is the input value (n for fibonacci)

fib_maker, e.g.

x bound to call_f_on_inner

```
def Y(f):
    def call_f_on_inner(x):
        def call_x_on_self_then_z(z):
            return x(x)(z)
        return f(call_x_on_self_then_z)
    return call_f_on_inner(call_f_on_inner)
```

f should return a fn of one argument

↳ Notice the first arg of f goes to the outer layer of our lambda fn.

this is call_x_on_self_then_z

So we have fib_maker = lambda f: lambda r: ?

How can we use call_x_on_self_then_z?

Well, calling it on some z applies x to itself then calls the result on z.

↳ But we know x is just call_f_on_inner.

↳ So really calling call_f_on_self_then_z(z) calls

call_f_on_inner(call_f_on_inner)(z)

f(call_x_on_self_then_z)(z)

So within our λ function, we could write

f = lambda z: <fib_maker>(call_f_on_inner)(z)

f = <fib_maker>(call_f_on_inner)

BUT this is exactly what Y(fib_maker) returns!

So f is Y(fib_maker). If we have a solid base case, we can argue that f is the function we are trying to define!

Let's say fib_maker doesn't reference r; so we have a base case.

Then call_x_on_self_then_z never gets called. So no recursion; we can ignore that fn. and we have a solid base case.

So f refers to the fn. we want to create, and we can use it for recursion!

6. **Longest Palindromic Subsequence**

(a) A *subsequence* of a string is another string which contains possibly non-consecutive characters of the string in the same order. For example, “abcd", “acd" and “bd" are all subsequences of “abcd" (although there are others).

Given a string `s`, return the longest palindromic subsequence of `s`. If there are multiple options for palindromic subsequences, you may return any; the doctests follow the rule that all chosen letters are as far left as possible. You may not use functions from any other problem.

```
def gp_subseq(s):
    """
    >>> gp_subseq("")
    ''
    >>> gp_subseq("abc")
    'a'
    >>> gp_subseq("admrdak") # adrda is also acceptable
    'admda'
    """
    if _____:
        return _____
    elif _____:
        return _____
    return _____
```

s[1:-1] ⟺ s[1:len(s)-1]

a b c d e

1 len(s-1)

len(s) <= 1

s

s[0] + gp_subseq(s[1:-1]) + s[-1]

max(gp_subseq(s[1:]), gp_subseq(s[:-1]), key=len)

"m" "r"