Solutions to CS 61A Challenge Problems: Midpoint Review

Other worksheets and solutions at https://alextseng.net/teaching/cs61a/

Alex Tseng

---

1 Recursion

(a) *Challenge* Define a function `parens` that takes in a positive integer `n` and returns the list of all possible nesting patterns of parentheses:
`parens(2) ---> ["()()", "(())"]`
`parens(3) ---> ["()()()", "()(())", "(())()", "(()())", "((()))"]`
Order in the list doesn't matter, but be wary of repeats.

```
def parens(n):
    if n == 1:
        return ["()"]
    else:
        parens_one_fewer = parens(n - 1)
        return ["()" + p for p in parens_one_fewer] + \
               [p + "()" for p in parens_one_fewer if (p + "()") != ("()" + p)] + \
               ["(" + p + ")" for p in parens_one_fewer]
```

The main part of this problem is noticing that if we can get all of the possible arrangements for `n-1`, then we can add that last `n`th set of parentheses in 3 ways: to the left, to the right, and surrounding the entire string. The trick is to realize that if we *define* our building process to always add to the outside, then we only need to consider these cases! This strict definition allows us to just use 3 list comprehensions to build up our answer for `n` from the answer for `n-1`. The other tricky part is worrying about repeats. Since we may be adding our new parentheses to both the left and right of the items, we must make sure that they result in different structures. In our code, we always add to the left. We add to the right only if it results from something different than adding to the left. This allows us to avoid repeats.

(b) *Challenge* Given a string `s`, write a function `perms` that returns a list of all the possible permutations of characters. You may assume all characters are unique:
`perms("61a") ---> ["61a", "6a1", "16a", "a61", "1a6", "a16"]`
Order in the list doesn't matter.

```
def perms(s):
    if not s:
        return [s]
    else:
        return [p[:i] + s[0] + p[i:] for i in range(len(s)) for p in perms(s[1:])]
```

Classic tree recursion. That list comprehension at the end may be a little confusing. It is a 2-dimensional one. Notice that we call `perms(s[1:])`. Taking the recursive leap of faith, this returns all of the permutations of `s` *without* the first character. Then how do we include the first character? We realize that for each permutation, we can insert that first character anywhere! So for one permutation `p`, we can insert that first character anywhere from the beginning to the end. This is what the first half of the list comprehension does. The second half of the list comprehension iterates through all the permutations for `s[1:]`. This gives us all the possible permutations including the first character of `s`.
Interesting note: using the ternary operator, we can actually write the body of this function in one line!
`return [s] if not s else <list comp>`

(c) *Challenge* A tarsier spies a delicious cricket at the top of a ladder with `n` rungs. Due to a tarsier's short arms and legs, it can only climb 1 or 2 rungs at a time. How many ways can the tarsier climb the

ladder to retrieve its reward? Define a function `climbs` to determine the answer:
`climbs(4) ---> 5 (1111, 12, 21, 22)`

```
def climbs(n):
    if n == 1:
        return 1
    if n == 2:
        return 2
    else:
        return climbs(n - 1) + climbs(n - 2)
```

Realize that at the current step, if we want to climb up the ladder, we can either take 1 step or 2 steps now. Each one gives us a different path. If we take 1 step first, then there are `climbs(n-1)` ways up the ladder after that. If we take 2 steps first, then there are `climbs(n-2)` ways up the ladder after that. Then we add them together to get all the possible ways. The tricky part is the base case. If we have 1 step left, we can just take 1 step and reach the top. If we have 2 steps left, we can either do two hops of 1 or one hop of 2. We do need to include both base cases or else we might be calling `climbs(0)`, which can complicate things.

(d) Recall Pascal's Triangle. Write a function `pascal` that takes in `i,j` and returns the entry of Pascal's Triangle in the `i`th row and `j`th entry in that row. `i` and `j` are both 0-indexed. It may help to look up the precise definition of the triangle:

```
pascal(4, 0) ---> 1
pascal(4, 2) ---> 6
pascal(4, 3) ---> 4
pascal(5, 3) ---> 10
```

```
def pascal(i, j):
    if j == 0:
        return 1
    if j == i:
        return 1
    else:
        return pascal(i - 1, j - 1) + pascal(i - 1, j)
```

This one is fairly straightforward if you are familiar with Pascal's Triangle. Our base cases are simple. If we are at the beginning of a row or at the end of a row, we return 1. Otherwise, we sum the items above it. Be careful with off-by-one errors. I recommend drawing out Pascal's Triangle as a list of lists so you don't get confused by the indexing.

2 Mutable Data and Functions

(a) Draw the environment diagram of the following:

```
def butter(fly):
    cater = 20
    pillar = 10
    def chrysalis(mystery):
        nonlocal cater
        cater = mystery(cater)
        pillar = mystery(fly)
        return [cater, pillar]
    return chrysalis
pupa = butter(4)
a = pupa(lambda x: x / 2)
b = pupa(lambda x: x / 2)
bugs = list([a, b]) + list(a)
```

Try the code out yourself on the environment diagram creator. Here is a permalink to one with the code already inputted: `https://goo.gl/NckLwp`
Pay attention to not only the various functions, but also the end result for `bugs`. It may behoove you to go back to the Mutable Data problems.

(b) *Challenge* Write a function to compute the `n`th Fibonacci number. The first time it is called on `n`, it should take $\Theta(n)$ time. All later calls to the function for the same `n` should take $\Theta(1)$ time.
Hint: Use higher order functions and memoization.

```
def fast_fibo():
    memo = {}
    def fibo(n):
        nonlocal memo
        if n in memo:
            return memo[n]
        else:
            prev = 0
            current = 1
            for _ in range(n):
                prev, current = current, prev + current
            memo[n] = current
            return current
    return fibo
```

This question has more to do with design. We want a function to compute Fibonacci numbers faster after we've already done them once. Then we simply need to keep track of the things we've already seen before. The solution here uses a nested function that keeps a nonlocal dictionary for this lookup of things we've seen before. If you wish to use mutable functions, this is an example of what it might look like. You could've also used objects.

3 Hierarchical Data Structures

As a reminder, here are the definitions of `Link` and `Tree`:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is Link.empty:
            rest = ""
        else:
            rest = ", " + repr(self.rest)
        return "Link({0}{1})".format(self.first, rest)

class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        self.branches = branches

    def is_leaf(self):
        return len(self.branches) == 0

    def __repr__(self):
        if self.branches:
            return "Tree({0}, {1})".format(self.entry, repr(self.branches))
        else:
            return "Tree({0})".format(repr(self.entry))
```

(a) Write a function `reverse_print` that prints out the elements of the linked list in reverse order. Do not create new Links or modify the existing list.

```
def reverse_print(links):
    if links.rest is Link.empty:
        print(links.first)
    else:
        reverse_print(links.rest)
        print(links.first)
```

Do not be fooled. This is a very simple program, but it will test if you truly understand recursion. Make sure you understand the *order* of the statements. What would happen if we swapped the last two lines? Answer: we'd simply be printing out the list in order.

(b) *Challenge* Write a function `k_from_end` that returns the `k`th element from the end of a linked list. A value of `k = 0` corresponds to the last element. Try your best to do this in $\Theta(n)$ time where n is the length of the list.
`k_from_end(Link(1, Link(2, Link(3, Link(4)))), 2) ---> 2`

```
def k_from_end(links, k):
    tortoise, hare = links, links
    for _ in range(k):
        hare = hare.rest
    while hare.rest is not Link.empty:
        tortoise, hare = tortoise.rest, hare.rest
    return tortoise.first
```

The hardest part is to do it in $\Theta(n)$ time. This means we need to iterate through the list just once (or at least a constant number of times). The trick is to have a tortoise and a hare that are pointers. They start out at the head of the list. Then the hare is given a head start of `k` Links. So now the hare is ahead of the tortoise by `k`. Now we increment the tortoise and the hare *at the same speed* so that the hare is *always* ahead of the tortoise by `k` Links. When the hare is at the very end, the tortoise is `k` away from the end.

(c) *Challenge* Write a function `reverse_k` that reverses every `k` elements of a linked list *in place*. Assume that the length of the list is a perfect multiple of `k`:

```
x = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6, Link(7, Link(8, Link(9)))))))))
y = reverse_k(x, 3)
y ---> Link(3, Link(2, Link(1, Link(6, Link(5, Link(4, Link(9, Link(8, Link(7)))))))))
```

Note that since `reverse_k` changes the first link, it will return a pointer to the new head of the list. This means that `x` will no longer point to the head of the list. Instead, `x` will still be pointing to the 1.

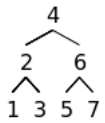
```
def reverse_k(links, k):
    if links is Link.empty or links.rest is Link.empty:
        return links
    prev, current = links, links.rest
    for _ in range(k - 1):
        temp = prev
        prev, current = current, current.rest
        prev.rest = temp
    links.rest = reverse_k(current, k)
    return prev
```

This question really should be marked as a double challenge! This combines both recursion and iteration. Our basic algorithm is that we iterate through the first `k` links. While iterating, we set the pointers backwards. This is why we have `prev` and `current`. After setting the pointers backwards, we then recurse on the rest of the list. Remember to assume the recursive leap of faith. The result of this recursion is set to the rest of the list. **You must draw out the example of 9 links. Run through the code exactly.**

(d) A binary search tree is a tree with two very special properties:

1. Each subtree (including the main one) has exactly 0 or 2 children.
2. For every subtree `t`, every element in `t`'s left branch is smaller than `t.entry`, and every element in `t`'s right branch is larger than `t.entry`.

Here is an example of a valid binary search tree. This example is particularly balanced, but it is not necessary that the depth is the same on each side.



Write a function `is_bst` that takes in a tree and returns whether or not it is a valid binary search tree. Hint: It may be helpful to keep track of the minimum and maximum allowable values for a subtree.

```
def bst_helper(t, minimum, maximum):
    if t.entry < minimum or t.entry > maximum:
        return False
    if not t.branches:
        return True
    if len(t.branches) != 2:
        return False
    return bst_helper(t.branches[0], minimum, t.entry) and \
           bst_helper(t.branches[1], t.entry, maximum)

def is_bst(t):
    return bst_helper(t, float("inf"), -float("inf"))
```

Trees are naturally recursive, so we definitely want to use recursion. We start by checking `t`'s entry to make sure it is within the correct bounds. If `t` has no children, we're done. Otherwise, we need to check that it has the correct number of children, and that each child is also a valid BST. To check this, we realize that if `t` has a certain minimum and maximum, then everything to the left of `t` must also follow the same minimum, but now the maximum has been lowered to `t.entry`! Similarly, for the right branch of `t`, it has the same maximum, but the minimum has been raised to `t.entry`. Once you realize this recurrence relation between `t` and its branches, this problem because very simple.