CS 61A Challenge Problems:
Recursive Data Structures and Orders of Growth

Solutions at https://alextseng.net/teaching/cs61a/

Alex Tseng

---

1 Linked Lists

Consider the following basic definition of a linked list:

```
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)

five_links = Link(1, Link(2, Link(3, Link(4, Link(5)))))
ten_links = Link(1, Link(2, Link(3, Link(4, Link(5, Link(6, Link(7, Link(8, Link(9, Link(10))))))))))
```

(a) Write a function `rotate_left` that returns a new linked list, rotated one to the left. This means that the first element is now the last element.
`rotate_left(five_links) ---> Link(2, Link(3, Link(4, Link(5, Link(1)))))`

(b) Write a function `rotate_right` that returns a new linked list, rotated one to the right. This means that the last element is now the first element.
`rotate_right(five_links) ---> Link(5, Link(1, Link(2, Link(3, Link(4)))))`

(c) Write a function `filter_links` that goes through a linked list and filters out items that do not satisfy a given predicate function. `filter_links` returns a new linked list and leaves the old one untouched.
`filter_links(ten_links, lambda x:  x % 3 == 0) ---> Link(3, Link(6, Link(9)))`

(d) *Challenge* Write a function `delete_every_k`. This function takes in a linked list and some number `k` and deletes every `k`th element *in place*. You are guaranteed that `k` is at least 2. If we were to call this function on a linked list, then the function itself will return `None`, but the linked list will be modified such that the `k`th, `2k`th, `3k`th, etc. elements are removed.
`delete_every_k(ten_links, 4) ---> None`
`ten_links ---> Link(1, Link(2, Link(3, Link(5, Link(6, Link(7, Link(9, Link(10))))))))`
Hint: it may behoove you to write a `length` function that returns the length of the linked list.

Note: "in place" means the function does not create any new `Link` objects.

2 Trees

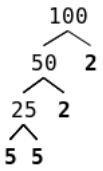
Consider the following basic definition of a tree:

```
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))
```

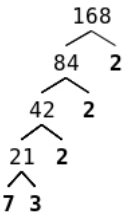
(a) Write a function `prime_factors_tree`. This function takes in a single positive integer `x` greater than 1, and returns a new tree that contains the prime factorization. Each subtree has either 2 or 0 branches.

The returned tree should follow these examples:

`prime_factors_tree(100)`:



`prime_factors_tree(168)`:



The order of the prime factors on the leaves is not too important, but try your best to reproduce the output here (prime factors increase in value as you go down and left). You may assume a function `is_prime` is already written for you.

(b) Now that we have our prime factors tree, we wish to actually find our prime factors. Write a function `get_leaves` that retrieves the entries of all the leaves and puts them in a list. Your function should work for all trees in general, and not just prime factorization trees, which always have either 0 or 2 branches. Order of the list does not matter.
`get_leaves(prime_factors_tree(168)) ---> [7, 3, 2, 2, 2]`

3 Orders of Growth

Θ notation is used to simplify functions. For example, if you have a very complicated runtime like $T(n) = 25n^2 + 2.6\pi n - 46.2 + \sqrt{10}$, it is much easier to display that as $\Theta(n^2)$. $\Theta(f(n))$ is a set of functions—an infinite set to be precise—that includes all of the functions that "simplify" to $\Theta(f(n))$. In 61B, you will learn how to rigorously prove that a function $g(n)$ is in $\Theta(f(n))$.

(a) Simplify the following runtimes as much as possible using Θ notation
$T(n) = n^2 + n \in \Theta(n^2)$

$T_1(n) = 2 \times 6^n \in$

$T_2(k) = 2^{k+5} \in$

$T_3(n) = 25 \in$

$T_4(n, m) = n^2 + mn \in$

$T_5(n) = 2^{\log n} \in$

(b) What are the runtimes (in Θ notation) of the following functions?

```
def foo(n):
    for i in range(n):
        for j in range(i, n):
            print("Help I'm stuck in a loop!")
```

```
def binary_search(sorted_list, item):
    if not sorted_list:
        return "Not here!"
    middle_index = int(len(sorted_list) / 2)
    if sorted_list[middle_index] == item:
        return "Found it!"
    if sorted_list[middle_index] > item:
        return binary_search(sorted_list[:middle_index], item)
    if sorted_list[middle_index] < item:
        return binary_search(sorted_list[middle_index + 1:], item)
```

```
from math import sqrt
def bar(x):  # This one's challenging!
    if x < 2:
        print("root")
    else:
        print("square")
        bar(sqrt(x))
```