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

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

Alex Tseng

---

1 Linked Lists

For all of these linked list problems, it is ESSENTIAL that you **draw out the box-and-pointer diagrams**. It can be extremely difficult to understand the solutions (let alone solve the problem) without the visual aid. 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)))))`

```
def rotate_left(link_lst):
    if link_lst.rest is Link.empty:
        return link_lst
    second = link_lst.rest
    current = link_lst
    while current.rest is not Link.empty:
        current = current.rest
    current.rest = link_lst
    link_lst.rest = Link.empty
    return second
```

First we check if the list only has one element. If so, then we simply return it as is. Otherwise, we rotate. We start by saving the second item, since it will be the new first. We then traverse the list until we get to the last element. When we reach there, we make the last element point to the original first element, and set that element's tail to the empty link. Then we return the new head.

(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)))))`

```
def rotate_right(link_lst):  # Move last element to first
    if link_lst.rest is Link.empty:
        return link_lst
```

```
    previous = link_lst
    current = previous.rest
    while current.rest is not Link.empty:
        previous = previous.rest
        current = current.rest
    # At this point, current is the last item, and previous is the second to last item
    previous.rest = Link.empty
    current.rest = link_lst
    return current
```

First we check if the list only has one element. If so, then we simply return it as is. Otherwise, we rotate. Now we need to find the last element and the second to last element. This is what the while loop and the `previous` and `current` pointers are doing. Once we find these links, we make the second to last element the new last element by setting its tail to be the empty link. And we place the old last element at the head by setting its tail to the old first element. We then return the new head.

(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)))`

```
def filter_links(link_lst, pred):
    if link_lst.rest is Link.empty:
        return link_lst if pred(link_lst.first) else Link.empty
    if pred(link_lst.first):
        return Link(link_lst.first, filter_links(link_lst.rest, pred))
    else:
        return filter_links(link_lst.rest, pred)
```

It is much easier to use recursion for this problem. First we check if the list has only one item. If so, we return the list itself if it passes our predicate, and the empty list if otherwise. You may not have seen this ternary operator before. It is commonly used when assigning or returning a value that depends on a simple boolean value. In the case that the list is longer, we check if the first element passes our test. If it does, we create a new link that includes the first element and the recursive call to the rest of the list. Otherwise, we only need to return the recurse call.

(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.

```
def length(link_lst):
    if link_lst is Link.empty:
        return 0
    return 1 + length(link_lst.rest)

def delete_every_k(link_lst, k):
    if length(link_lst) < k:
        return
    current = link_lst
    for _ in range(k-2):
        current = current.rest
    # Delete the item after current
    delete_every_k(current.rest.rest, k)
    current.rest = current.rest.rest
```

Again, we use recursion. Our base case is if the length of our list is less than `k`. If so, then we don't need to do any work at all. We break out of our function, returning nothing. Otherwise, we traverse the list and find the `k`th element from the start. This loop is why we assume `k` is no smaller than 2. We delete this element by recursing on the list *after* the `k`th link, and then setting the `k-1`th link's tail to the recursive result. Again, draw box-and-pointer diagrams for this.

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

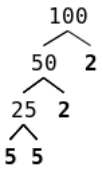
2 Trees

Consider the following basic definition of a tree:

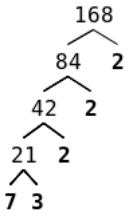
```
class Tree:
    def __init__(self, entry, branches=()):
        self.entry = entry
        self.branches = branches

    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.

```
def prime_factors_tree(x):
    if is_prime(x):
        return Tree(x)
    for i in range(2, int(sqrt(x)) + 1):
        if x % i == 0:
            return Tree(x, branches=(prime_factors_tree(x / i), Tree(i)))
```

Trees are naturally recursive, as are linked lists, so we use recursion again. If `x` is prime itself, then we simply return a lone tree with `x` as its entry and no branches. Otherwise, `x` is composite (not prime) and has prime factors. You may be wondering why we only iterate from 2 to the square root (inclusive). There is a theorem that states that every composite number has a prime factor that does not exceed its square root. This means we will definitely find a prime factor in this loop. Since we iterate through the

numbers in increasing order, we must find the prime factors in increasing order. Once we find a prime factor, we simply return a new tree with `x` as its root, and its children being the prime factor we found and the result of recursing on whatever is left. If we put the branches in a different order, then our tree would be flipped. If you're a little confused still, try walking through the code with `x = 100`.

(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]
```

```
def get_leaves(tree):
    if tree.is_leaf():
        return [tree.entry]
    leaves = []
    for branch in tree.branches:
        leaves.extend(get_leaves(branch))
    return leaves
```

We use recursion (Hopefully this is not a surprise anymore). If the tree is a leaf (the method to check this is conveniently already written for you), then we simply return a list with that entry in it. Otherwise, we must iterate through all of the branches, and find all of the leaves in those branches. If you need a refresher on the `extend` method, it is a bound method of lists and will simply concatenate two lists together. If we used `append`, then we would be appending lists into lists, and get a nested list instead of a nice 1-dimensional list like we want.

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
$T(n) = n^2 + n \in \Theta(n^2)$

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

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

$T_3(n) = 25 \in \Theta(1)$

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

$T_5(n) = 2^{\log_b n} \in \Theta(n^{\frac{1}{\log_2 b}})$
T_1: Constant factors in front of the function don't matter. However, the base of an exponential order of growth (c^n) matters a lot. 3^n grows much much much faster than 2^n.

T_2: We can rewrite 2^{k+5} to $2^5 2^k = 32 \times 2^k$. Then just like in T_1, we can ignore the constant factor in front. What if the function had been $T_2(n) = 2^{5k}$? Then we would not be able to take away the constant factor, because that is equivalent to $(2^5)^k = 32^k$, which grows astronomically faster than 2^k. Also note that just because we use k here, it doesn't mean anything different. n and k are just variables in functions.

T_3: Any constant function can be simplified to $\Theta(1)$.

T_4: This is a trick question! It may seem like you can get rid of the mn term because n^2 looks like it will dominate the linear mn term. However, this is incorrect. m is a variable of the function, too. If n is much larger than m, then yes, this function simplifies down to $\Theta(n^2)$. But if m is much larger than n, then in fact, this function would simplify down to $\Theta(m)$! Because we don't know which is larger, we need to keep both terms.

T_5: We'll have to use some logarithm tricks. Review your logarithm laws, because you will see them everywhere in computer science. Let us rewrite $2^{\log_b n}$ using the change of base formula: $2^{\log_b n} = 2^{\frac{\log_2 n}{\log_2 b}} = (2^{\log_2 n})^{\frac{1}{\log_2 b}} = n^{\frac{1}{\log_2 b}} \in \Theta(n^{\frac{1}{\log_2 b}})$. As ugly as $\frac{1}{\log_2 b}$ is, it is a simple constant that depends on b, so we can't get rid of it. So we see that depending on the base b that we started out with, we get a different polynomial runtime. In the case that $b = 2$, then we end up with linear $\Theta(n)$. The larger b is, in fact, the better our runtime will be.

(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))
```

`foo`: $\Theta(n^2)$. We see that we have a set of nested loops. We iterate from 0 to $n-1$. And for each of those iterations, we iterate through i to $n-1$. Then the real work that we're doing (printing out the message) is simply the number of times the inner loop is run. We're printing out the message $n + (n-1) + (n-2) + \cdots + 2 + 1$ times. Convince yourself of this fact by running through this code a few iterations and seeing what i and j are each time. Any time you see this sort of sum, you know it is $\Theta(n^2)$ time. It may be a little confusing at first because it seems like you're never actually adding n to itself n times, but we can actually rewrite this as $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. Then it becomes clear that this runtime really is in $\Theta(n^2)$.

`binary_search`: $\Theta(\log n)$. Make sure you know what the code is doing first before analyzing the runtime. In this case, it can be extremely difficult to just look purely at the code to find the order of growth.

The name of the function gives a hint. This function looks for an item in a sorted list. We look at the middle element, and if it is there, we have found it. Otherwise, we look in the right half or the left half, if the middle element is too small or too large (respectively). Notice that in one run of the function (without recursing), we do a constant amount of work. We find the middle index and do a simple comparison. This does not depend on the size of the list. But what about recursing? Each time we split the list in half, until we get down to the empty list. This means that if there are k recursive calls, then our runtime is in $\Theta(k)$, since each call we do constant work. How many calls are there? This is equivalent to asking how many times do we need to cut something in half until we get down to 1 or 0. The answer is $\log_2 n$ (This is equivalent to solving $n(\frac{1}{2})^k = 1$ for k). So our final runtime is $\Theta(\log n)$.

`bar`: $\Theta(\log \log x)$. The trick to this function is to note the overall structure. We're doing some simple recursion. Each level of recursion we do constant work by printing out something. So we only need to know how many levels of recursion there are. Each time we take the square root of x until we reach something that is less than 2. Then we're trying to solve this equation: $x^{(\frac{1}{2})^k} = 2$. We start with x, and we take the square root repeatedly, and we want to know how many times until we get down to 2. Again, we'll have to use our logarithm and exponential rules. $x^{(\frac{1}{2})^k} = x^{\frac{1}{2^k}} = 2$. Take the log (base 2) of both sides (It doesn't have to be base 2, but it makes our analysis easier): $\frac{1}{2^k} \log x = \log 2 = 1$, or $2^k = \log x$. Then we simply take the log again to find that $k = \log \log x$. This is the number of levels of recursion. So our final runtime is in $\Theta(\log \log x)$.