EFFICIENCY, LINKED LISTS, AND MIDTERM REVIEW

COMPUTER SCIENCE MENTORS

March 8, 2021 - March 11, 2021

1 Efficiency

An order of growth (OOG) characterizes the runtime **efficiency** of a program as its input becomes extremely large. Common runtimes, in increasing order of time, are: constant, logarithmic, linear, quadratic, and exponential.

Examples:

Constant time means that no matter the size of the input, the runtime of your program is consistent. In the function `f` below, no matter what you pass in for `n`, the runtime is the same.

```
def f(n):
    return 1 + 2
```

A common example of a linear OOG involves a single for/while loop. In the example below, as `n` gets larger, the amount of time to run the function grows proportionally.

```
def f(n):
    while n > 0:
        print(n)
        n -= 1
```

An example of a quadratic runtime involves nested for loops. If you increment the value of `n` by only 1, an additional `n` amount of work is being done, since the inner for loop will run one more time. This means that the runtime is proportional to n^2.

```
def f(n):
    for i in range(n):
```

```
    for j in range(n):
        print(i*j)
```

1. What is the order of growth for `foo`?

   (a)
   ```
   def foo(n):
       for i in range(n):
           print('hello')
   ```

   (b) What's the order of growth of `foo` if we change `range(n)`:

      i. To `range(n/2)`?

      ii. To `range(n**2 + 5)`?

      iii. To `range(10000000)`?

2. What is the order of growth for `belgian_waffle`?

```
def belgian_waffle(n):
    total = 0
    while n > 0:
        total += 1
        n = n // 2
    return total
```

2 Linked Lists

Linked lists consists of a series of links which have two attributes: `first` and `rest`. The `first` attribute contains some sort of value that is usually what you want to end up storing in the list (these can be integers, strings, lists etc.). The `rest` attribute, on the other hand, is a pointer to another link or `Link.empty`, which is just an empty linked list represented traditionally by an empty tuple (but not necessarily, so never assume that it is represented by an empty tuple otherwise you may break an abstraction barrier!).

Because each link contains another link or `Link.empty`, linked lists lend themselves to recursion (just like trees). Consider the following example, in which we double every value in linked list. We mutate the current link and then recursively double the rest.

```
def double_values(link):
     if link is not Link.empty:
        link.first *= 2 # we mutate the value inside of the link
        double_val(link.rest) # we mutate the values in the rest
                              # of the linked list
    # if the link is empty then do nothing
```

However, unlike with trees, we can also solve many linked list questions using iteration. Take the following example where we have written `double_values` using a while loop instead of using recursion:

```
def double_values_iter(link):
    while link is not Link.empty:
        link.first *= 2
        link = link.rest # Note that this does not mutate
                         # the original linked list;
                         # it changes what link the variable
                         # link is pointing to
```

For each of the following problems, assume linked lists are defined as follows:

```
class Link:
    empty = ()
    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

To check if a `Link` is empty, compare it against the class attribute `Link.empty`:

```
if link is Link.empty:
    print('This linked list is empty!')
```

1. What will Python output? Draw box-and-pointer diagrams to help determine this.

```
>>> a = Link(1, Link(2, Link(3)))
>>> a.first
```

```
>>> a.first = 5
>>> a.first
```

```
>>> a.rest.first
```

```
>>> a.rest.rest.rest.rest.first
```

```
>>> a.rest.rest.rest = a
>>> a.rest.rest.rest.rest.first
```

```
>>> repr(Link(1, Link(2, Link(3, Link.empty))))
```

```
>>> Link(1, Link(2, Link(3, Link.empty)))
```

```
>>> str(Link(1, Link(2, Link(3))))
```

```
>>> print(Link(Link(1), Link(2, Link(3))))
```

2. Write a function `skip`, which takes in a `Link` and returns a new `Link` with every other element skipped.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> a
    Link(1, Link(2, Link(3, Link(4))))
    >>> b = skip(a)
    >>> b
    Link(1, Link(3))
    >>> a
    Link(1, Link(2, Link(3, Link(4)))) # Original is unchanged
    """
    if ____________________________________________:

        ____________________________________________

    elif __________________________________________:

        ____________________________________________

    ________________________________________________
```

3. Now write function `skip` by mutating the original list, instead of returning a new list. Do NOT call the `Link` constructor.

```
def skip(lst):
    """
    >>> a = Link(1, Link(2, Link(3, Link(4))))
    >>> skip(a)
    >>> a
    Link(1, Link(3))
    """
```

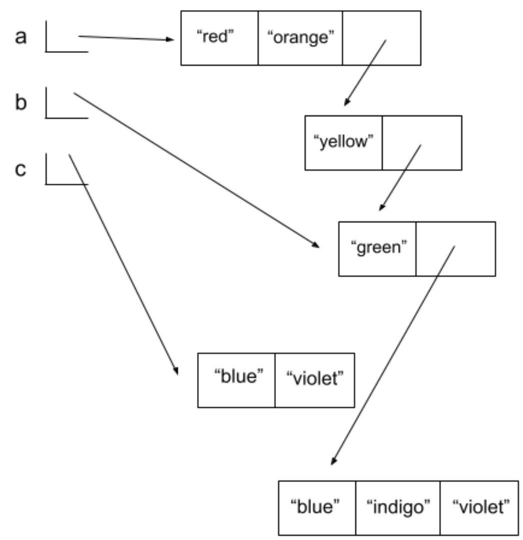
4. **(Optional)** Write `has_cycle` which takes in a `Link` and returns `True` if and only if there is a cycle in the `Link`.

```
def has_cycle(s):
    """
    >>> has_cycle(Link.empty)
    False
    >>> a = Link(1, Link(2, Link(3)))
    >>> has_cycle(a)
    False
    >>> a.rest.rest.rest = a
    >>> has_cycle(a)
    True
    """
```

3 Midterm Review

1. Fill in each blank in the code example below so that its environment diagram is the following. Do not write any new colors (use only references to pre-existing list elements).

```
a = ["red", "red"]
b = ["orange", "green", "
   indigo"]
a[1] = _________
c = ["yellow", "green"]
a.append(_______)
c = ["blue", "violet"]
a[___][___] = __________
b.pop(_____)
b[1] = __________ +
   __________ + __________
```



2. Implement `subsets`, which takes in a list of values and an integer `n` and returns all subsets of the list of size exactly `n` in any order. You may not need to use all the lines provided.

```
def subsets(lst, n):
    """
    >>> three_subsets = subsets(list(range(5)), 3)
    >>> for subset in sorted(three_subsets):
    ...     print(subset)
    [0, 1, 2]
    [0, 1, 3]
    [0, 1, 4]
    [0, 2, 3]
    [0, 2, 4]
    [0, 3, 4]
    [1, 2, 3]
    [1, 2, 4]
    [1, 3, 4]
    [2, 3, 4]
    """
    if n == 0:
        ________________________________

    if ________________________________:

        ________________________________

    ______________________________________________

    ______________________________________________

    return ______________________________________
```

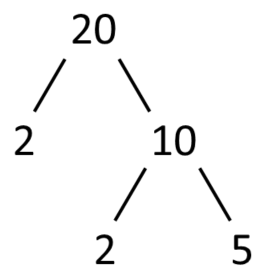
3. Write a generator function `num_elems` that takes in a possibly nested list of numbers `lst` and yields the number of elements in each nested list before finally yielding the total number of elements (including the elements of nested lists) in `lst`. For a nested list, yield the size of the inner list before the outer, and if you have multiple nested lists, yield their sizes from left to right.

```
def num_elems(lst):
    """
    >>> list(num_elems([3, 3, 2, 1]))
    [4]
    >>> list(num_elems([1, 3, 5, [1, [3, 5, [5, 7]]]]))
    [2, 4, 5, 8]
    """

    count = ________________

    for ______________________________:

        if ___________________________:

            for _____________________________________:

                yield ______________________________

            ___________________________________

        else:

            ___________________________________

    yield ______________________________
```

4. Define `delete_path_duplicates`, which takes in `t`, a tree with non-negative labels. If there are any duplicate labels on any path from root to leaf, the function should mutate the label of the occurrences deeper in the tree (i.e. farther from the root) to be the value `-1`.

```
def delete_path_duplicates(t):
    """
    >>> t = Tree(1, [Tree(2, [Tree(1), Tree(1)])])
    >>> delete_path_duplicates(t)
    >>> t
    Tree(1, [Tree(2, [Tree(-1), Tree(-1)])])
    >>> t2 = Tree(1, [Tree(2), Tree(2, [Tree(2, [Tree(1, Tree
       (5))])])])
    >>> delete_path_duplicates(t2)
    >>> t2
    Tree(1, [Tree(2), Tree(2, [Tree(-1, [Tree(-1, [Tree(5)])])
       ])])
    """
     def helper(_______________, _______________):

         if ________________________________:

             ________________________________

         else:

             ________________________________

         for _______ in ____________________:

             ________________________________

     ________________________________________
```

5. Define the function `factor_tree` which takes in a positive integer `n` and returns a factor tree for `n`. In a factor tree, multiplying the leaves together is the prime factorization of the root, `n`. See below for an example of a factor tree for `n = 20`.




```
def factor_tree(n):
    """
    >>> factor_tree(20)
    Tree(20, [Tree(2), Tree(10, [Tree(2), Tree(5)])])
    >>> factor_tree(1)
    Tree(1)

    for i in ______________________:

        if __________________________:

            return Tree(_____, ________________________________)

    ___________________________________
```