EFFICIENCY, LINKED LISTS, AND MIDTERM REVIEW Solutions

COMPUTER SCIENCE MENTORS

October 26, 2020 - October 29, 2020

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')
   ```

   Linear. This is a for loop that will run `n` times.

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

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

      Linear. The loop runs `n/2` times, but the runtime still scales linearly proportionally to `n`.

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

      Quadratic. The number of times the loop runs is proportional to n^2.

      iii. To `range(10000000)`?

      Constant. No matter the size of `n`, we will run the loop the same number of times.

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
```

Logarithmic. Notice that with each pass through the while loop, the value of `n` is halved. Since we are halving till 0, this would be a logarithmic runtime.

CSM 61A Fall 2020: Jason Chang and Jade Singh, with
Catherine Cang, Alina Trinh, Ethan Chiu, Jamie Ip, Kenneth Chi, and
Aidan Tong, Aman Shah, Bridget Cheng, Cindy Lin, Ethan Yeh, Ivan Penev, Ivy Li, James Fletcher, Jemmy Zhou, Jennifer Huang, Jessica Yu, Joshua Baum, Kevin Moy, Kunal Agarwal, Lauren Meier, Matthew Guo, Mihira Patel, Nikhita Anumukonda, Owen Gong, Richard Roggenkemper, Sean Sananikone, Uma Unni

2 Linked Lists

Linked lists consists of a series of links which have two attributes: `first` and `rest`. `First` 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.). `Rest`, on the other hand, needs to be a pointer to another link or `Link.empty`, which is just an empty linked list represented traditionally by an empty tuple (but it does not have to be so you should 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 with a while loop as well. 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)))
```

```
+---+---+   +---+---+   +---+---+
| 1 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+
```

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

```
1
```

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

```
+---+---+   +---+---+   +---+---+
| 5 | --|->| 2 | --|->| 3 | / |
+---+---+   +---+---+   +---+---+
```

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

5

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

2

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

Error: tuple object has no attribute rest (Link.empty has no rest)

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

   +---+---+  +---+---+  +---+---+
+->| 5 | --|->| 2 | --|->| 3 | --|--+
|  +---+---+  +---+---+  +---+---+  |
|                                   |
+-----------------------------------+

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

2

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

"Link(1, Link(2, Link(3)))"

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

Link(1, Link(2, Link(3)))

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

'<1 2 3>'

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

<<1> 2 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 ______________________________________:
        ________________________________________
    ____________________________________________
```

```
    if lst is Link.empty
        return Link.empty
    elif lst.rest is Link.empty:
        return Link(lst.first)
    return Link(lst.first, skip(lst.rest.rest))
```

Base cases:

- When the linked list is empty, we want to return a new Link.empty.
- If there is only one element in the linked list (aka the next element is empty), we want to return a new linked list with that single element.

Recursive case:

All other longer linked lists can be reduced down to either a single element or empty linked list depending on whether it has odd or even length. Therefore, we want to keep the first element, and recurse on the element after the next (skipping the immediate next element with `lst.rest.rest`). To build a new linked list, we can add new links to the end of the linked list by calling skip recursively inside the `rest` argument of the `Link` constructor.

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))
    """
```

```
def skip(lst): # Recursively
    if lst is Link.empty or lst.rest is Link.empty:
        return
    lst.rest = lst.rest.rest
    skip(lst.rest)

def skip(lst): # Iteratively
    while lst is not Link.empty and lst.rest is not Link.empty
      :
        lst.rest = lst.rest.rest
        lst = lst.rest
```

Because this problem is mutative, we should never be creating a new list - we should never have `Link(x)`, or the creation of a new Link instance, anywhere in our code! Instead, we'll be reassigning `lst.rest`.

In order to skip a node, we can assign `lst.rest = lst.rest.rest`. If we have lst assigned to a link list that looks like the following:

`1 -> 2 -> 3 -> 4 -> 5`

Setting `lst.rest = lst.rest.rest` will take the arrow that points form 1 to 2 and change it to point from 1 to 3. We can see this by evaluating `lst.rest.rest`. `lst.rest` is the arrow that comes from 1, and `lst.rest.rest` is the link with 3.

Once we've created the following list:

`1 -> 3 -> 4 -> 5`

we just need to call skip on the rest of the list. If we call skip on the list that starts at 3, we'll skip over the link with 4 and set the pointer from 3 to point to the link with 5. This is the behavior that we want! Therefore, our recursive call is `skip(lst.rest)`, since `lst.rest` is now the link that contains 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
    """

    if s is Link.empty:
        return False
    slow, fast = s, s.rest
    while fast is not Link.empty:
        if fast.rest is Link.empty:
            return False
        elif fast is slow or fast.rest is slow:
            return True
        slow, fast = slow.rest, fast.rest.rest
    return False
```

3 Midterm Review

1. Draw the box-and-pointer diagram.

```
>>> violet = [7, 77, 17]
>>> violet.append([violet.pop(1)])

>>> dash = violet * 2
>>> jack = dash[3:5]
>>> jackjack = jack.extend(jack)

>>> helen = list(violet)
>>> helen += [jackjack]
>>> helen[2].append(violet)
```

https://goo.gl/EAmZBW

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 _______________________________________

    if n == 0:
        return [[]]
    if len(lst) == n:
        return [lst]
    with_first = [[lst[0]] + x for x in subsets(lst[1:], n -
       1)]
    without_first = subsets(lst[1:], n)
    return with_first + without_first
```

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 _____________________________
```

```
def num_elems(lst):
    count = 0
    for elem in lst:
        if type(elem) is list:
            for c in num_elems(elem):
                yield c
            count += c
        else:
            count += 1
    yield count
```

`count` refers to the number of elements in the current list `lst` (including the number of elements inside any nested list). Determine the value of `count` by looping through each element of the current list `lst`. If we have an element `elem` which is of type **`list`**, we want to yield the number of elements in each nested list of `elem` before finally yielding the total number of elements in `elem`. We can do this with a recursive call to `num_elems`. Thus, we yield all the values that need to be yielded using the inner for loop. The last number yielded by this inner loop is the total number of elements in `elem`, which we want to increase `count` by. Otherwise, if `elem` is not a list, then we can simply increase `count` by 1. Finally, yield the total count of the list.

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 ____________________:

             ________________________________

     ________________________________________
```

```
     def helper(t, seen_so_far):
         if t.label in seen_so_far:
           t.label = -1
         else:
             seen_so_far = seen_so_far + [t.label]
         for b in t.branches:
             helper(b, seen_so_far)
     helper(t, [])
```

5. Write a function that returns true only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

```
def contains_n(elem, n, t):
    """
    >>> t1 = Tree(1, [Tree(1, [Tree(2)])])
    >>> contains_n(1, 2, t1)
    True
    >>> contains_n(2, 2, t1)
    False
    >>> contains_n(2, 1, t1)
    True
    >>> t2 = Tree(1, [Tree(2), Tree(1, [Tree(1), Tree(2)])])
    >>> contains_n(1, 3, t2)
    True
    >>> contains_n(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ___________________________________________:

        return _____________________________________

    elif ___________________________________________:

        return _____________________________________

    else:

        return _____________________________________
```

```
    if n == 0:
        return True
    elif t.is_leaf():
        return n == 1 and t.label == elem
    elif t.label == elem:
        return True in [contains_n(elem, n - 1, b) for b in
          t.branches]
    else:
        return True in [contains_n(elem, n, b) for b in
          t.branches]
```

Base cases: The simplest case we have is when `n == 0`, or when we want at least 0 instances of `elem` in `t`. In this case, we always return `True`. The other simple case we consider is when the tree is only a leaf — there is nothing left to recurse on. In that case, we simply check to see that both `n == 1` and that `t.label == elem`, meaning that we have one element left to satisfy, and the leaf label satisfies the final element we are looking for. If we have more elements to search for (ie. n ¿ 1), then we will not satisfy that many elements at the leaf node; conversely, if we have fewer (ie. `n == 0`), then the case would already be covered by the first base case.

Recursive cases: If the current node isn't a leaf, then there's two different cases we should consider. Either the label of the current node is equal to `elem` or the label is not equal to `elem`. For the former, we would have to search for `n` more `elems` in each branch of `t` and return `True` if any of the branches contain `n` elems. For the latter, we would have `(n - 1)` elements remaining, so we would search for `(n - 1)` more `elems` in each branch of `t` and return `True` if any of the branches contain `(n - 1)` `elems`. Since there is not room to do a for loop, we can use a list comprehension to recursively call the function on each branch. Thus, our two list comprehension statements would be `[contains_n(elem, n, b)for b in t.branches]` and `[contains_n(elem, n - 1, b)for b in t.branches]`. To determine if any of the branches contain either `n` elems or `(n - 1)` elems, we can check if there's a `True` element in the respective lists.