CS 61A
Fall 2018

Orders of Growth & Linked Lists

Discussion 7: October 10, 2018 Solutions

1 Orders of Growth

When we talk about the efficiency of a function, we are often interested in the following: as the size of the input grows, how does the runtime of the function change? And what do we mean by "runtime"?

- `square(1)` requires one primitive operation: `*` (multiplication). `square(100)` also requires one. No matter what input `n` we pass into `square`, it always takes one operation.

input	function call	return value	number of operations
1	square(1)	$1 \cdot 1$	1
2	square(2)	$2 \cdot 2$	1
⋮	⋮	⋮	⋮
100	square(100)	$100 \cdot 100$	1
⋮	⋮	⋮	⋮
n	square(n)	$n \cdot n$	1

- `factorial(1)` requires one multiplication, but `factorial(100)` requires 100 multiplications. As we increase the input size of n, the runtime (number of operations) increases linearly proportional to the input.

input	function call	return value	number of operations
1	factorial(1)	$1 \cdot 1$	1
2	factorial(2)	$2 \cdot 1 \cdot 1$	2
⋮	⋮	⋮	⋮
100	factorial(100)	$100 \cdot 99 \cdots 1 \cdot 1$	100
⋮	⋮	⋮	⋮
n	factorial(n)	$n \cdot (n-1) \cdots 1 \cdot 1$	n

For expressing complexity, we use what is called big Θ (Theta) notation. For example, if we say the running time of a function `foo` is in $\Theta(n^2)$, we mean that the running time of the process will grow proportionally with the square of the size of the input as it becomes very large.

- **Ignore lower order terms:** If a function requires $n^3 + 3n^2 + 5n + 10$ operations with a given input n, then the runtime of this function is in $\Theta(n^3)$. As n gets larger, the lower order terms (10, $5n$, and $3n^2$) all become insignificant compared to n^3.
- **Ignore constants:** If a function requires $5n$ operations with a given input n, then the runtime of this function is in $\Theta(n)$. We are only concerned with how the runtime grows asymptotically with the input, and since 5n is still asymptotically linear; the constant factor does not make a difference in runtime analysis.

Kinds of Growth

Here are some common orders of growth, ranked from no growth to fastest growth:

- $\Theta(1)$ — constant time takes the same amount of time regardless of input size
- $\Theta(\log n)$ — logarithmic time
- $\Theta(n)$ — linear time
- $\Theta(n \log n)$ — linearithmic time
- $\Theta(n^2)$, $\Theta(n^3)$, etc. — polynomial time
- $\Theta(2^n)$, $\Theta(3^n)$, etc. — exponential time (considered "intractable"; these are really, really horrible)

In addition, some programs will never terminate if they get stuck in an infinite loop.

Questions

What is the order of growth for the following functions?

1.1
```
def sum_of_factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n) + sum_of_factorial(n - 1)
```

$\Theta(n^2)$, we will call factorial n times with arguments $n, n-1, n-2, ..., 0$. The sum from 0 to n is approximately n^2.
Video walkthrough

1.2
```
def bonk(n):
    total = 0
    while n >= 2:
        total += n
        n = n / 2
    return total
```

$\Theta(\log(n))$, because our while loop iterates at most $\log(n)$ times, due to n being halved in every iteration.
Video walkthrough

1.3
```
def mod_7(n):
    if n % 7 == 0:
        return 0
    else:
        return 1 + mod_7(n - 1)
```

$\Theta(1)$, since at worst it will require 6 recursive calls to reach the base case. So this is $\Theta(6)$, which can be reduced to $\Theta(1)$.

2 Linked Lists

There are many different implementations of sequences in Python. Today, we'll explore the linked list implementation.

A linked list is either an empty linked list, or a Link object containing a `first` value and the `rest` of the linked list.

To check if a linked list is an empty linked list, compare it agains the class attribute `Link.empty`:

```
if link is Link.empty:
    print('This linked list is empty!')
else:
    print('This linked list is not empty!')
```

Implementation

```
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:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(repr(self.first), rest_str)

    @property
    def second(self):
        return self.rest.first

    @second.setter
    def second(self, value):
        self.rest.first = value

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

Questions

2.1 Write a function that takes in a Python list of linked lists and multiplies them element-wise. It should return a new linked list.

If not all of the `Link` objects are of equal length, return a linked list whose length is that of the shortest linked list given. You may assume the `Link` objects are shallow linked lists, and that `lst_of_lnks` contains at least one linked list.

```
def multiply_lnks(lst_of_lnks):
    """
    >>> a = Link(2, Link(3, Link(5)))
    >>> b = Link(6, Link(4, Link(2)))
    >>> c = Link(4, Link(1, Link(0, Link(2))))
    >>> p = multiply_lnks([a, b, c])
    >>> p.first
    48
    >>> p.rest.first
    12
    >>> p.rest.rest.rest
    ()
    """
```

Recursive solution:

```
    product = 1
    for lnk in lst_of_lnks:
        if lnk is Link.empty:
            return Link.empty
        product *= lnk.first
    lst_of_lnks_rests = [lnk.rest for lnk in lst_of_lnks]
    return Link(product, multiply_lnks(lst_of_lnks_rests))
```

For our base case, if we detect that any of the lists in the list of `Links` is empty, we can return the empty linked list as we're not going to multiply anything.

Otherwise, we compute the product of all the firsts in our list of `Links`. Then, the subproblem we use here is the rest of all the linked lists in our list of Links. Remember that the result of calling `multiply_lnks` will be a linked list! We'll use the product we've built so far as the first item in the returned `Link`, and then the result of the recursive call as the rest of that `Link`.

Iterative solution:

```
    import operator
    from functools import reduce
    def prod(factors):
         return reduce(operator.mul, factors, 1)

    head = Link.empty
    tail = head
    while Link.empty not in lst_of_lnks:
```

```
            all_prod = prod([l.first for l in lst_of_lnks])
            if head is Link.empty:
                head = Link(all_prod)
                tail = head
            else:
                tail.rest = Link(all_prod)
                tail = tail.rest
            lst_of_lnks = [l.rest for l in lst_of_lnks]
        return head
```

The iterative solution is a bit more involved than the recursive solution. Instead of building the list "backwards" as in the recursive solution (because of the order that the recursive calls result in, the last item in our list will be finished first), we'll build the resulting linked list as we go along.

We use `head` and `tail` to track the front and end of the new linked list we're creating. Our stopping condition for the loop is if any of the `Link`s in our list of `Link`s runs out of items.

Finally, there's some special handling for the first item. We need to update both head and tail in that case. Otherwise, we just append to the end of our list using tail, and update tail.

2.2 Write a function that takes a sorted linked list of integers and mutates it so that all duplicates are removed.

```
def remove_duplicates(lnk):
    """
    >>> lnk = Link(1, Link(1, Link(1, Link(1, Link(5)))))
    >>> remove_duplicates(lnk)
    >>> lnk
    Link(1, Link(5))
    """
```

Recursive solution:

```
    if lnk is Link.empty or lnk.rest is Link.empty:
        return
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
        remove_duplicates(lnk)
    else:
        remove_duplicates(lnk.rest)
```

For a list of one or no items, there are no duplicates to remove.

Now consider two possible cases:

- If there is a duplicate of the first item, we will find that the first and second items in the list will have the same values (that is, `lnk.first == lnk.rest.first`). We can confidently state this because we were told that the input linked list

is in sorted order, so duplicates are adjacent to each other. We'll remove the second item from the list.

Finally, it's tempting to recurse on the remainder of the list (`lnk.rest`), but remember that there could still be more duplicates of the first item in the rest of the list! So we have to recurse on `lnk` instead. Remember that we have removed an item from the list, so the list is one element smaller than before. Normally, recursing on the same list wouldn't be a valid subproblem.

- Otherwise, there is no duplicate of the first item. We can safely recurse on the remainder of the list.

Iterative solution:

```
while lnk is not Link.empty and lnk.rest is not Link.empty:
    if lnk.first == lnk.rest.first:
        lnk.rest = lnk.rest.rest
    else:
        lnk = lnk.rest
```

The loop condition guarantees that we have at least one item left to consider with `lnk`.

For each item in the linked list, we pause and remove all adjacent items that have the same value. Once we see that `lnk.first != lnk.rest.first`, we can safely advance to the next item. Once again, this takes advantage of the property that our input linked list is sorted.

3 Midterm Review

3.1 Write a function that takes a list and returns a new list that keeps only the even-indexed elements of lst and multiplies them by their corresponding index.

```
def even_weighted(lst):
    """
    >>> x = [1, 2, 3, 4, 5, 6]
    >>> even_weighted(x)
    [0, 6, 20]
    """

    return [__________________________________________________]


    return [i * lst[i] for i in range(len(lst)) if i % 2 == 0]
```

Alternatively, we can take advantage of the step size for range to make sure we only consider even numbered indices:

```
    return [i * lst[i] for i in range(0, len(lst), 2)]
```

The key point to note is that instead of iterating over each element in the list, we must instead iterate over the indices of the list. Otherwise, there's no way to tell if we should keep a given element.

One way of solving these problems is to try and write your solution as a for loop first, and then transform it into a list comprehension. The for loop solution might look something like this:

```
result = []
for i in range(len(lst)):
    if i % 2 == 0:
        result.append(i * lst[i])
return result
```

Video walkthrough

3.2 The **quicksort** sorting algorithm is an efficient and commonly used algorithm to order the elements of a list. We choose one element of the list to be the **pivot** element and partition the remaining elements into two lists: one of elements less than the pivot and one of elements greater than the pivot. We recursively sort the two lists, which gives us a sorted list of all the elements less than the pivot and all the elements greater than the pivot, which we can then combine with the pivot for a completely sorted list.

First, implement the `quicksort_list` function. Choose the first element of the list as the pivot. You may assume that all elements are distinct.

Note: in computer science, "sorting" refers to placing elements in order from least to greatest, not putting things in categories

```
def quicksort_list(lst):
```

```
    """
    >>> quicksort_list([3, 1, 4])
    [1, 3, 4]
    """

    if ____________________________________________________:

        ____________________________________________________

    pivot = lst[0]

    less = _____________________________________________

    greater = _____________________________________________

    return _____________________________________________


def quicksort_list(lst):
    if len(lst) <= 1:
        return lst
    pivot = lst[0]
    less = [e for e in lst[1:] if e < pivot]
    greater = [e for e in lst[1:] if e > pivot]
    return quicksort_list(less) + [pivot] +
           quicksort_list(greater)
```

A list with zero or no elements is already sorted. Otherwise, we follow the procedure outline in the description.

We pick a "pivot" to remove from the list. Then, construct `less` and `greater` lists that represent items less than and greater than the pivot. Notice that both of these lists are guaranteed to be smaller than our original list, so we are guaranteed that these are valid subproblems for our recursive call.

The sorted version of the items less than the pivot will all be less than the pivot, so adding pivot to the end of that list maintains a sorted list over all the smaller elements plus the pivot. Finally, the sorted version of the items greater than pivot will be all be greater than the items less than the pivot plus the pivot, so adding that to the end will ensure our list is still sorted. After concatenating all these lists together, we have a sorted version of our original list.

3.3 Write a function that takes in a list and returns the maximum product that can be formed using nonconsecutive elements of the list. The input list will contain only numbers greater than or equal to 1.

```
def max_product(lst):
    """Return the maximum product that can be formed using lst
    without using any consecutive numbers
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """


    if lst == []:
        return 1
    elif len(lst) == 1: # Base case optional
        return lst[0]
    else:
        return max(max_product(lst[1:]), lst[0]*max_product(lst[2:]))
```

At each step, we choose if we want to include the current number in our product or not:

- If we include the current number, we cannot use the adjacent number.
- If we don't use the current number, we try the adjacent number (and obviously ignore the current number).

The recursive calls represent these two alternate realities. Finally, we pick the one that gives us the largest product.
Video walkthrough

3.4 Complete `redundant_map`, which takes a tree `t` and a function `f`, and applies `f` to each node (2^d) times, where d is the depth of the node. The root has a depth of 0. It should mutate the existing tree rather than creating a new tree.

```
def redundant_map(t, f):
  """
  >>> double = lambda x: x*2
  >>> tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
  >>> redundant_map(tree, double)
  >>> print_levels(tree)
  [2] # 1 * 2 ^ (1) ; Apply double one time
  [4, 8] # 1 * 2 ^ (2), 2 * 2 ^ (2) ; Apply double two times
  [16] # 1 * 2 ^ (2 ^ 2) ; Apply double four times
  [256] # 1 * 2 ^ (2 ^ 3) ; Apply double eight times
  """
  t.label = _____________________________________________
```

```
  new_f = ____________________________________________________

  ____________________________________________________________

        ____________________________________________________
```

```
def redundant_map(t, f):
  """
  >>> double = lambda x: x*2
  >>> tree = Tree(1, [Tree(1), Tree(2, [Tree(1, [Tree(1)])])])
  >>> redundant_map(tree, double)
  >>> print_levels(tree)
  [2]
  [4, 8]
  [16]
  [256]
  """
  t.label = f(t.label)
  new_f = lambda x: f(f(x))
  for branch in t.branches:
    redundant_map(branch, new_f)
```

Every time we recurse, we transform our map function into one that is one level deeper in terms of calls to input function `f`. To see why this will achieve the result we want, let's look at what happens to some input function `f`.

- The first call to `redundant_map` will call `f` once.
- This means on the second call to `redundant_map`, we pass in a function `g` that causes the original `f` to be called two times.
- On the third call to `redundant_map`, we pass in a function `h` that causes `g` to be called two times. Remember that `g` calls original `f` twice, so `h` will end up calling original `f` four times.

Therefore, each level will have double the calls to `f` as the previous level, which matches the requirements.