CS-UY 1134: Data Structures and Algorithms
Lab 4: Review

Friday, June 21, 2019

1 Instructions

This lab will review several of the topics that have been covered so far in the course. These topics are fair game for the first midterm exam.

Try to solve all of the problems by the end of the lab time. As added practice, it can be a good idea to try writing your code out on pen and paper first, as you will have to do this on the exam.

When you type up your code, write some test cases. Try multiple test cases for each function you have to implement, and consider edge cases carefully.

1.1 What to submit

If you believe you have solved all the problems, let a TA or the professor know and they will check your work. This is a good chance to get feedback on problems similar to what you might see on the exam.

If you don't finish by the end of the lab, you may submit something on NYU Classes by Sunday June 16, 11:55 PM.

Either when a TA has checked your solutions, or by the Sunday submission deadline, submit your solutions to the "Lab 4: Review" assignment on NYU Classes.

2 Warm-up Written Problems (Don't Submit)

The following are some warm-up problems that you should be able to answer on an exam. Solve these *without* typing code into the Python interpreter (though you may wish to use the Python shell to check your answers). You do not have to submit anything for these problems, but you should do them for practice, as similar questions may be on the exam.

1. What would the following line of code print out if entered in the Python shell?

```
[x*5 for x in range(10,20,2) if x*5 % 6 != 0]
```

2. Determine the output of the following code segments.

   (a)
   ```
   import copy
   lst = ['a', [1, 2], ['b', [2, 'c']], 'd']
   lst_copy = copy.copy(lst)
   lst_copy[0] = 'A'
   lst_copy[1][1] += 10
   lst_copy[2][1][0] *= 4
   print(lst)
   print(lst_copy)
   ```

   (b)
   ```
   import copy
   lst = ['a', [1, 2], ['b', [2, 'c']], 'd']
   lst_deepcopy = copy.deepcopy(lst)
   lst_deepcopy[0] = 'A'
   lst_deepcopy[1][1] += 10
   lst_deepcopy[2][1][0] *= 4
   print(lst)
   print(lst_deepcopy)
   ```

3 Written Problems (Submit These)

Please answer the following questions and submit as a ".pdf" file. You may either type your answers or handwrite them and take a picture, but in either case, please submit as a PDF file. If you handwrite, make sure you write clearly and neatly.

3. Analyze the running time complexity (using Big-Oh or Big-Theta) of the following Python functions. Make sure your answer is *tight*: That is, if the code runs in time $O(n)$, do **not** say that it is $O(n^2)$.

   (a)
   ```
   def foo1(lst):
       i = len(lst) ** 2
       for j in range(len(lst)-1):
           lst[j], lst[j+1] = lst[j+1], lst[j]
           if j + i <= j:
               break
           i //= 2
   ```

   (b)
   ```
   def foo2(lst):
       n = len(lst)
       for i in range(n-1):
           j = i+1
           while j > 1:
               lst[j] = 3*j - i
               if lst[j] > lst[j-1] / 2:
                   j -= 1
   ```

```
        else:
            j -= 3
```

4. Below are two algorithms for recursively finding the sum of all the elements of a list of integers. Make sure you understand how each algorithm works and the recursive idea in each implementation. Then, answer the following question for each:

   (a) Let `lst = [1, 2, 3, 4, 5, 6, 7, 8]`. Trace the execution when given the list `lst` (initially passing in `low = 0` and `high = 7`). Write down all outputs in the order they are output, and the return values of the functions.

   (b) Draw the recursion tree when given a list of n elements. Include the cost of each call. Use the recursion tree to conclude the total asymptotic running time of each function.

   Finally, use your answers to the above questions to determine which function is asymptotically faster.

```
def sum_list1(lst, low, high):
    print("sum_list1, low=", low, ", high=", high, sep='')
    if low == high:
        return lst[low]
    else:
        sum_of_rest = sum_list1(lst, low+1, high)
        total_sum = lst[low] + sum_of_rest
        return total_sum

def sum_list2(lst, low, high):
    print("sum_list2 low=", low, ", high=", high, sep='')
    if low == high:
        return lst[low]
    else:
        mid = (low + high) // 2
        left_sum = sum_list2(lst, low, mid)
        right_sum = sum_list2(lst, mid+1, high)
        total_sum = left_sum + right_sum
        return total_sum
```

4 Coding Problems

Solve the following coding problems and submit as a ".py" file. Try writing the code out on paper before tying up your solution.

5. Below is an implementation of a function `random_str(n)`, which takes a number `n` and *returns a string* containing n random letters separated

by a space. Each letter is a randomly chosen letter from `"a"` to `"z"` (all lowercase). For example, `random_str(10)` could return `"f m q o n r g f z g"`.

```
import random
def random_str(n):
    letters = "abcdefghijklmnopqrstuvwxyz"
    s = ""
    for i in range(1,n+1):
        curr = random.choice(letters)
        s = s + curr + ' '
    return s
```

(a) Analyze the asymptotic running time of the above implementation. In your analysis:

i. Assume that the + operator for strings *creates a new string* instance containing the result of the concatenation. This runs in *linear* time; that is in time proportional to the length of the newly created string.

ii. Assume that calling the function `random.choice(letters)` takes time $\Theta(1)$ (that is, it is constant time).

(b) There is a string method called `join` that takes an iterable (such as a list) and returns a concatenation of all the elements of the list, separated by the string object on which `join` is called.

For example, the call `'-'.join(["abc", "cat", "dog"]` would return `"abc-cat-dog"`.

If any of the values in the given list or iterable is not a string, `join` raises a TypeError.

The method `join` is guaranteed to run in linear time; that is, in time proportional to the length of the string that is returned.

Using the `join` method, give an implementation of `random_str` that runs in *linear* time; that is, in time $\Theta(n)$.

6. Implement a method `powers(base, n)` that creates a generator, that when iterated over, yields n *powers* of `base`. For example, the code:

```
for val in powers(2, 10):
    print(val, end=' ')
```

should print: `2 4 8 16 32 64 128 256 1024` and the code

```
for val in powers(3, 5):
    print(val, end=' ')
```

should print: `3 9 27 81 243`

7. Write a **recursive** method `is_palindrome` that checks if a given string is a palindrome. It should take three parameters: a string `input_str` to check, and two indices `low` and `high` which indicate the range of indices in the string to be checked. The function should return a boolean indicating whether the substring of `input_str` between the indices `low` and `high` form a palindrome.

   For example, if `input_str` is `"racecar"`, the function should return `True`; but if `input_str` is `"recursion"`, the function should return `False`. (In both examples, calling the function initially with `low = 0` and `high = len(input_str)`)

8. In this problem, we will implement a function `partition(lst)`. This will *partition* a given list using `lst[0]` as what's called a *pivot*. The function should *modify* the list in place so that all items *less than* the pivot occur *before* all items that are *greater than* the pivot. Further, the pivot element should be placed after the elements *less than* the pivot and *before* the elements that are greater than the pivot. As an example, if `lst = [42, 17, 81, 77, 68, 22, 55, 10, 90]`, calling `partition(lst)` should produce output that looks like:

```
[22,  17,  10,    42,    68,  77,  55,  81,  90]
{   < pivot  }  {pivot}  {       > pivot       }
```

   The order of the elements less than the pivot, and the order of the elements greater than the pivot doesn't matter. Your function should run in time $\Theta(n)$ for a list of length n. Further, your function should only use $\Theta(1)$ additional space.