NYU, Tandon School of Engineering

CS-UY 1134 Data Structures and Algorithms — Summer 2023

Homework #2
Due by Friday 7/21, 11:55pm

Submission instructions:

1. You should turn in 6 files:
   - 2 '.pdf' files, one for question 1-2 and another for question 3
     Name your file: 'YourNetID_hw2_q1to2.pdf' and 'YourNetID_hw2_q3.pdf
   - 4 '.py' files, one for each question 4-7.
     Name your files: 'YourNetID_hw2_q4.py' and 'YourNetID_hw2_q5.py', etc.

Note: your netID follows an abc123 pattern, not N12345678.

2. You should submit your homework via Gradescope. For Gradescope's autograding feature to work:
   - Name all classes, functions and methods **exactly as they are in the assignment specifications**.
   - Make sure there are **no print statements** in your code. If you have tester code, please put it in a "main" function and **do not call it**.

Question 1:

Use the definitions of O and of Θ in order to show the following:

a. $5n^3 + 2n^2 + 3n = O(n^3)$

b. $\sqrt{7n^2 + 2n - 8} = \Theta(n)$

c. Show that if $d(n)=O(f(n))$ and $e(n)=O(g(n))$, then the product $d(n)e(n)$ is $O(f(n)g(n))$.

Question 2:

Give a θ characterization, in terms of *n*, of the running time of the following four functions:

```
def example1(lst):
 """Return the sum of the prefix sums of sequence S."""
      n = len(lst)
      total = 0
      for j in range(n):
            for k in range(1+j):
                  total += lst[k]
      return total


def example2(lst):
"""Return the sum of the prefix sums of sequence S."""
      n = len(lst)
      prefix = 0
      total = 0
      for j in range(n):
            prefix += lst[j]
            total += prefix
      return total


def example3(n):
    i = 1
    sum = 0
    while (i < n*n):
        i *= 2
        sum += i
    return sum
```

Question 3:

You are given 2 implementations for a recursive algorithm that calculates the sum of all the elements in a list (of integers):

```
def sum_lst1(lst):
  if (len(lst) == 1):
      return lst[0]
  else:
      rest = sum_lst1(lst[1:])
      sum = lst[0] + rest
      return sum


def sum_lst2(lst, low, high):
  if (low == high):
      return lst[low]
  else:
      rest = sum_lst2(lst, low + 1, high)
      sum = lst[low] + rest
      return sum
```

Note: The implementations differ in the parameters we pass to these functions:

- In the first version we pass only the list (all the elements in the list have to be taken in to account for the result).
- In the second version, in addition to the list, we pass two indices: low and high (low ≤ high), which indicate the range of indices of the elements that should to be considered.
  The initial values (for the first call) passed to low and high would represent the range of the entire list.

1) Make sure you understand the recursive idea of each implementation.
2) Analyze the running time of the implementations above. For each version:
   i) Draw the recursion tree that represents the execution process of the function, and the local-cost of each call.
   ii) Conclude the total (asymptotic) running time of the function.
3) Which version is asymptotically faster?

Question 4:

Implement the function **`def`** `split_parity(lst)`. That takes `lst`, a list of integers. When called, the function changes the order of numbers in `lst` so that all the odd numbers will appear first, and all the even numbers will appear last. Note that the inner order of the odd numbers and the inner order of the even numbers don't matter.

For example, if `lst` is a list containing [1, 2, 3, 4], after calling `split_parity`, `lst` could look like: [3, 1, 2, 4].

Implementation requirements:

1. You are **not allowed** to use an auxiliary list (a temporary local list).
2. Pay attention to the running time of your implementation. An efficient implementation would run in a linear time. That is, if n is the length of `lst`, the running time should be $\Theta(n)$.

Question 5:

Implement the function **`def`** `findChange(lst01)`. This function is given `lst01`, a list of integers containing a sequence of 0s followed by a sequence of 1s.
When called, it returns the index of the first 1 in `lst01`.

For example, if `lst01` is a list containing `[0, 0, 0, 0, 0, 1, 1, 1]`, calling `findChange(lst01)` will return 5.

Note: Pay attention to the running time of your function. If `lst01` is a list of size n, an efficient implementation would run in logarithmic time (that is $\Theta(log_2(n))$).

<u>Question 6:</u>

Give a **recursive** implement to the following functions:

a) `def count_lowercase(s, low, high):`

The function is given a string `s`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should return the number of lowercase letters at the positions *low, low+1, …, high* in `s`.

b) `def is_number_of_lowercase_even(s, low, high):`

The function is given a string `s`, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should return `True` if there are even number of lowercase letters at the positions *low, low+1, …, high* in `s`, or `False` otherwise.

<u>Question 7:</u>

Give a **recursive** implement to the following function:

`def split_by_sign(lst, low, high)`

The function is given a list `lst` of non-zero integers, and two indices: `low` and `high` (`low` ≤ `high`), which indicate the range of indices that need to be considered.

The function should reorder the elements in `lst`, so that all the negative numbers would come before all the positive numbers.

<u>Note</u>: The order in which the negative elements are at the end, and the order in which the positive are at the end, doesn't matter, as long as all he negative are before all the positive.