CS-UY 1134: Data Structures and Algorithms
Lab 3: Asymptotic Analysis

Friday, June 14, 2019

1 Instructions

This lab will involve solving some practice problems with an emphasis on achieving certain asymptotic running times. The goal is to provide practice thinking about asymptotic analysis as well as practicing solving problems *efficiently*.

Try to implement all the functions described by the end of the lab time. Try to plan out your approach ahead of time, and work out a solution with pen and paper first, before writing code.

As added practice, it can be a good idea to try writing your code out on pen and paper first. In addition to being a useful approach for planning out a solution, it is good practice for exams, where you will have to write some code on paper.

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.

Note that a significant portion of this lab will be based on acheiving the desired asymptotic running times. You will <u>not</u> get full credit if your solution does not have the proper asymptotic running time.

1.1 What to submit

If you believe you have working solutions to all the problems, have a TA or the professor check your solution.

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

Either when a TA has checked your code, or by the Sunday submission deadline, submit a single ".py" file to the "Lab 3: Asymptotic Analysis" assignment on NYU Classes. The ".py" file should contain your implementations of all the methods described below. Place any test code you wrote in a "main" function.

2 The Problems

2.1 Find First 1

Write a function `first1(lst)` that takes as input a list of 0's and 1's that is sorted (so that all 0's appear before all 1's) and returns the index of the first 1 in the list.

For example, if `lst` is the list `[0, 0, 0, 0, 1, 1]`, `first1(lst)` should return 4.

Your solution should run in *logarithmic* time: that is, $O(\log_2(n))$, where n is the length of the list.

2.2 Compute e

In this problem, we will compute the mathematical constant e approximately. The constant e has many uses and appears in many areas of mathematics. It's value is approximately 2.71828, though it is transcendental (and thus also irrational), like π. We can compute e using the following formula (which can be derived from the Taylor Series expansion for e^x):

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots = \sum_{k=0}^{\infty} \frac{1}{k!}$$

Write the function `e_approximation(n)` that takes a number n and computes an approximation of e by computing and adding the terms of this sum, up to the term $\frac{1}{n!}$. That is, you will compute

$$\sum_{k=0}^{n} \frac{1}{k!} \approx e$$

Your code should run in *linear time*: that is, $O(n)$.

2.3 Two-Sum

Write a function `two_sum(sorted_lst, target)` that takes as input a sorted list (`sorted_lst`) and a target number (`target`), and tries to find two numbers in the list that sum to `target`. The function should return two indices in the list as a tuple, corresponding to indices of the two elements that, when added, equal `target`. If no such pair of numbers exists, return `None`.

For example, calling `two_sum` with `sorted_lst = [-3,-2,0,5,17]` and `target = 2` should return `(0,3)`, because `sorted_list[0] + sorted_list[3] = -3 + 5 = 2 = target`.

Your solution should run in *linear* time: that is, $O(n)$, where n is the length of `lst`.

2.4 Separate Negatives and Positives

Write a function `split_neg_pos(lst)` that takes a list of numbers and rearranges the elements of the list so that all the negative numbers appear before all the positive numbers. This should *mutate* the list that was passed in, rather than return a new list.

For example, if `lst` is the list `[-7, 5, -3, 4, 2]`, after calling the function, `lst` *could* be `[-3, -7, 4, 2, 5]` (though the ordering among the negative numbers and the ordering among the positive numbers doesn't matter, so there could be other correct outputs).

You may only use $\Theta(1)$ additional space; this means you cannot use any additional collection of elements (such as a list) of non-constant size to solve the problem.

Your solution should run in *linear* time: that is, $O(n)$, where n is the length of the list `lst`.

2.5 Zeroes to End

Write a function `move_zeros(lst)` that takes a list `lst` of numbers and moves all 0's to the end of the list, preserving the relative order of all nonzero elements. This should *mutate* the list that was passed in, rather than return a new list.

For example, if `lst` is the list `[1,0,2,0,0,3,4]`, the function should *modify* the list`lst` to be `[1, 2, 3, 4, 0, 0, 0]`

You may only use $\Theta(1)$ additional space; this means you cannot use any additional collection of elements (such as a list) of non-constant size to solve the problem.

Your solution should run in *linear* time: that is, $O(n)$, where n is the length of the list `lst`.

2.6 Recursion Practice: Min

Write a **recursive** function `find_min(lst)` that, given a list, returns the minimum element in the list. For example, given the list `my_lst = [5, -1, 9, 6, 0]`, calling `find_min(my_lst)` should return -1.

Your implementation *must* be **recursive**. There is no requirement for running time on this problem, but think about the asymptotic running time of your implementation (it will depend on the length of the list that is passed in).