Homework 1

Due: Sept. 24, 2020 at 2:30 PM EDT (scratch work submission)
Oct. 1, 2020 at 2:30 PM EDT (final submission)

Instructions

Problems **5 through 7** on this homework cover material that will be discussed in lecture by September 24th.

Please submit evidence that you've thought about Problems **1 through 4** by the first due date (scratch work submission) in assignment **HW1: Scratch Work** on Gradescope. This can be an outline of your proofs, any thoughts on how you plan to approach the problems, etc. You are also free to submit completed problems instead of outlines, although you will not receive feedback on your work until after the final homework due date.

This homework must be typed in LaTeX and handed in via Gradescope.

Please ensure that your solutions are complete, concise, and communicated clearly. Use full sentences and plan your presentation before you write. Except in the rare cases where it is indicated otherwise, consider every problem as asking you to prove your result.

Problem 1

Order the following list of functions by their asymptotic (Big-O) growth, from slowest to fastest growth. I.e., if $f(n)$ precedes $g(n)$ in your ordering, then $f(n)$ should be $O(g(n))$. Also, group together functions that are Big-Theta (Θ) of one another. Briefly justify (a few sentences at most) these groupings

$$\begin{array}{cccccc} 6n\log n & 2^{100} & \log\log n & \log^2 n & 2^{2^n} & \lceil\sqrt{n}\,\rceil \\ n^{0.01} & 1/n & 4n^{3/2} & 3n^{0.5} & 5n & \lfloor 2n\log^2 n\rfloor \\ 2^n & n\log_4 n & 4^n & n^3 & n^2\log n & \sqrt{\log n} \end{array}$$

Hint: When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$

Problem 2

In the ***maximum subarray problem***, we are given an array of integers, $A = [a_1, a_2, ..., a_n]$, and are asked to find the largest subarray sum $s_{j,k}$, where

$$s_{j,k} = a_j + a_{j+1} + ... + a_k = \sum_{i=j}^{k} a_i,$$

for some indices j and k.

Note that each element of the array could have a positive, negative, or zero value. Thus, in the special case where all array elements are negative, the solution is 0, which corresponds to an empty subarray.

One of the solutions to the maximum subarray problem is the following algorithm, which we will call algorithm `MaxSubSlow`:

```
1: Input
2:     An n-element array, A, of numbers indexed from 1 to n
3: Output
4:     The maximum subarray sum of A
5: procedure MaxSubSlow(A)
6:     m ← 0                          ▷ Keeps track of the maximum found so far
7:     for j = 1 to n do
8:         for k = j to n do
9:             s ← 0                  ▷ Keeps track of the next partial sum we are computing
10:            for i = j to k do
11:                s ← s + A[i]
12:                if s > m then
13:                    m ← s
14:                end if
15:            end for
16:        end for
17:    end for
18:    return m
19: end procedure
```

This algorithm calculates the partial sum, $s_{j,k}$, of every possible subarray by adding up the values in the subarray $A[j : k]$. Moreover, for each such subarray sum, it compares that sum to a running maximum, and if the new value is greater than the old, it updates that maximum to the new value. In the end, this will be the maximum subarray sum.

a. Express the running time of algorithm MaxSubSlow as a function, $T(n)$, of its input size, n. Function $T(n)$ should count the number or steps in the pseudo-code executed.
b. Show that the running time of MaxSubSlow is $\Omega(n^3)$. That is, show that there exist constants $c > 0$ and $n_0 > 0$ such that $T(n) \geq cn^3$, for all $n \geq n_0$.

Example: To help you understand what we are looking for in the first question (a.), here is an example of expressing the running time of an algorithm as a function of the input size. Consider algorithm DoubleInput shown below. The running time of this algorithm is expressed by the

function $T(n) = 2n + 1$ because it makes an initial assignment to variable a and then updates the value of a twice for each of the n iterations of the **for** loop.

```
1: Input
2:     An integer, n
3: Output
4:     2n
5: procedure DoubleInput(n)
6:     a ← 0
7:     for i = 0 to n do
8:         a ← a − 1
9:         a ← a + 3
10:    end for
11:    return a
12: end procedure
```

Problem 3

You are at the store and want to buy some pasta. You are the kind of person who does not like to stand out, so you don't want to appear extravagant by buying expensive pasta nor do you want to appear stingy by buying cheap pasta. Your comfort zone is being right in the middle.

Thus, every week, you try to buy the package of pasta with *median* cost among all of the different packages of pasta on the shelves. However, each week, you notice that there are more and more different packages of pasta on the shelves, so you have to constantly recalculate which pasta you want to buy for the week.

You decide that the most efficient way to find the pasta you want is to keep track of all pasta prices by means of a data structure that maintains a *median* item in a collection of n items. Assume, for simplicity, that all items are distinct. Your data structure should use $O(n)$ space and support the following operations with the given time complexity:

- insert(x): Insert a new item, $O(\log n)$ time
- removeMedian(): Remove and return a median item, $O(\log n)$ time
- median(): Return (but do not remove) a median item, $O(1)$ time

Partial credit will be given for data structures that support operation median() in $O(\log n)$ time. Extra credit will be given for data structures that support a general collection of items where more than one item can have the same value.

Informally, a median of a set of elements is an element in the middle of the ordered sequence of elements. The formal definition is as follows. Recall that the elements of a set are distinct. Given a set with n elements, an element $x \in S$ is said to be a *median* of S if there are $\lfloor \frac{n}{2} \rfloor$ items in S less than or equal to x and $\lfloor \frac{n}{2} \rfloor$ items in S greater than or equal to x. If n is odd, the median is unique. If n is even, there are two medians.

Problem 4

Suppose there is an infinitely long, straight line, and people will sometimes step onto the line or step off of it, at locations corresponding to integers. People standing on the line will always be facing in the positive direction. Unfortunately, some of these people hold gravity-defying tomatoes that if thrown, will splatter on the next person on the line.

In particular, you are going to observe a series of events of the following type:

- enter(p):a person outside the line moves into unoccupied position p on the line;
- exit(p): the person standing at position p exits the line;
- throw(p): the person standing at position p throws a tomato in the positive direction, thus aimed at the next person in line, if any.

For example, imagine this series of events:

1. The line is empty.
2. Person A steps on the line at position 43.
3. Person B steps on the line at position 1.
4. Person C steps on the line at position 62.
5. Person B throws a tomato.

In this case, you should warn person A to duck!

Your objective is to design a data structure to model the above scenario so you can warn would-be victims before they get hit by a tomato. Each event type corresponds to a method of the data structure. Method throw(p) returns the occupied position immediately following p. In case p is not occupied or p is the last occupied position, throw(p) outputs an error message.

a. Describe how to modify a balanced binary search tree to realize the above data structure so that it uses $O(n)$ space and each of its methods (enter, exit, and throw) runs in $O(\log n)$ time, where n is the number of occupied positions.

The problem, however, gets more complex than this. In reality, each tomato is thrown at a different height, t, and only someone whose height is *strictly greater* than t will get hit. Thus, your new data structure should support methods for the following events:

- enter(p, h):a person with height h outside the line moves into unoccupied position p on the line;
- exit(p): the person standing at position p exits the line;
- throw(p, t): the person standing at position p throws a tomato in the positive direction at height t.

Method throw(p, t) returns the smallest position q such that $q > p$ and there is a person standing at q with height greater than t.

b. Describe how to modify a balanced binary search tree to realize the above more complicated data structure so that it uses $O(n)$ space and each of its methods runs in $O(\log n)$ time, where n is the number of occupied positions.

Hint: At each node, in addition to storing the position and height of a person, you should figure out what *additional* information to store.

Problem 5

Sports announcers are expected to keep talking during a broadcast of a sporting event even when there is nothing actually happening, such as during half-time. One common way to fill empty time is with sports trivia. Suppose, then, that you are going to be a sports announcer for the big game between the Bears and the Anteaters. To fill the empty time during half-time, you would like to say that this is the nth time that a game between the Bears and Anteaters has had a score of i-versus-j at half-time. The problem is that you don't know the values of i and j yet, of course, because the game hasn't happened yet, and once half-time arrives, you won't have time to look through the entire list of Bear-Anteater half-time scores to count the number of times the pair (i, j) appears.

a. Describe an efficient scheme for processing the list of Bear-Anteater half-time scores before the game so that you can quickly say, right at the start of half-time, how many times the pair (i, j) has occurred at similar moments in the past. The processing task should take time proportional to the number of previous games and the querying task should take constant time.

b. Justify the runtime and correctness of your scheme.

Problem 6

a. Rewrite the merge sort algorithm so that instead of dividing the input list into two equal-sized sublists at each step, it divides the list into **three** equal-sized sublists.

b. Analyze the runtime of this new algorithm

c. Is this algorithm asymptotically faster than the standard (split-in-two) mergesort algorithm seen in class?

Problem 7

Consider the following problem:

Input: a target integer N, and n numbers $a_1, \ldots, a_n$
Output: a pair of indexes i and j (between 1 and n) such that $a_i + a_j = N$ or report that no such indices exist

1. Give a deterministic $\Theta(n^2)$ algorithm for the problem, and prove its runtime and correctness.
2. Give a deterministic $\Theta(n \log n)$ algorithm for the problem, and prove its runtime and correctness.
3. Give a randomized algorithm that takes expected time $O(n)$, and prove its runtime and correctness.

Note: Express your runtimes in terms of n, the number of items in the list.