

Homework 4

Due April 15

Problem 4.1

Give an algorithm for the following problem:

CONTIGUOUS SUBSEQUENCE

Input: A sequence of integers x_1, \dots, x_n .

Output: The contiguous subsequence x_i, \dots, x_j with the largest sum.

(Note that the integers may be positive or negative.)

Problem 4.2

Give the frequency array and Huffman tree for the following string:

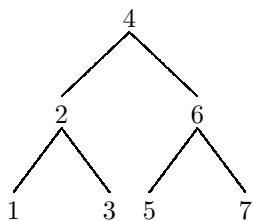
“whose wholly holy toes were in those holes”

Problem 4.3

- Design a greedy algorithm for making change after someone buys some candy costing x cents and the customer gives the clerk a one-dollar bill. Your algorithm should try to minimize the number of coins returned. Do not assume that the coins come in typical American denominations.
- Show that your greedy algorithm returns the minimum number of coins if the coins have denominations of 25 cents, 10 cents, 5 cents, and 1 cent.
- Give a set of denominations for which your algorithm does not return the minimum number of coins. Include an example where your algorithm fails.

Problem 4.4

- We start with an empty splay tree and perform the operations $\text{insert}(5)$, $\text{insert}(3)$, $\text{insert}(2)$, $\text{insert}(4)$, $\text{insert}(6)$, $\text{insert}(0)$, $\text{find}(3)$, $\text{insert}(8)$, $\text{find}(3)$, $\text{insert}(7)$. Draw the resulting splay tree.
- Take the following splay tree and perform the operations $\text{find}(1)$, $\text{find}(2)$, \dots , $\text{find}(7)$. What does the resulting tree look like?



- (c) In class, we defined $W(x)$ = number of nodes in tree rooted at x , so that $W(\text{leaf}) = 1$, and defined $H(x) = \log(W(x))$. We then defined the potential of a tree T to be

$$\phi(T) = \sum_{x \in T} H(x).$$

We argued in class that potentials were high for unbalanced trees and low for balanced trees. I claimed that among all trees with n nodes, the largest difference in potential was $O(n \log n)$. Show this in two steps.

- (i) Explain why the potential of any n -node tree is non-negative.
- (ii) Explain why the potential of the totally-unbalanced n -node tree (i.e., just a long line of nodes) is $O(n \log n)$.

Problem 4.5

In this exercise, you're going to try to address the "point in intervals" problem. You've got a bunch of intervals $J_i = [a_i, b_i]$ and a number p ; it's possible that p is in some interval J_i , i.e., that $a_i \leq p \leq b_i$; in fact, it's possible that p is in several of the intervals, because the intervals may overlap. Given a number p , you'd like to report, as fast as possible, the list of intervals that contain it (i.e., a list of pairs (a_i, b_i) defining all intervals in which p lies).

Example application: you're writing a calendar-manager program. The intervals are "calendar events" like "meet Sophie for Lunch, 12-1 Thursday Oct 5, 2008". You want to be able to ask, "What should I be doing right now?", i.e., "Which events on my calendar contain this precise moment?"

- (a) Suppose that you're given all the intervals (n of them), and are given time to build a data structure to contain them. The cost of building the data structure can be ignored, even if its exponential in the number of intervals (although it shouldn't be, unless you're doing something that we can't even imagine!). Having created this bunch-of-intervals structure, you're given a number p and must produce the list of intervals containing p . Find an algorithm/data structure that produces this list as fast as possible; describe its running time.

As a straw-man, you could simply keep a list of the intervals, sorted by their left endpoints. You then take p and walk through the list; if an interval contains p , you add it to the output; if it doesn't, you ignore it. And if the left-endpoint of the interval is greater than p , you can stop. This approach has $O(n)$ running time. You should aim to do better. Remember that you have all the time in the world to do pre-processing.

- (b) Now consider the somewhat harder problem: your "bunch-of-intervals" structure must support the addition of new intervals, and you don't get any pre-processing time. In other words, the input is a sequence of operations, each of which is either

`bunch.add(a, b)`

which adds the interval $[a, b]$ to the bunch, or

`bunch.findIntervals(p)`

which finds all intervals currently in the bunch that contain p . The output consists of the lists that the `findIntervals` calls produce.

You are to devise data structures and algorithms to make this efficient in two circumstances:

- (i) If you expect to do lots of `add` operations and few `findInterval` operations.
- (ii) If you expect to do few `add` operations and lots of `findInterval` operations.

The straw-man from the first part supports `add` in $O(n)$ (you have to walk the list to find the right place to insert the new interval) and `findIntervals` in $O(n)$. So it doesn't particularly favor either scenario (i). or ii. You may find a solution that does both operations in, say, $O(1)$ time, in which case you wouldn't favor either scenario either (although I'd be very surprised if you did so!). You might find one solution that does `add` in $O(1)$ but `findIntervals` in $O(n \log n)$, and another for which the opposite is true; these would be suited to scenarios i and ii, respectively.

You should also give big-O performance bounds for your operations.

Note: this is a very open-ended problem. I (jfh) don't know the literature on this problem, and have only a couple of vague ideas about what might work. I certainly can do better than the "list of intervals" straw man, but I don't know how *much* better. In this sense, you're facing a problem that a typical programmer might encounter: "I need a way to find containing intervals. How do I go about that?" It's possible that the speed of your algorithm for "find" will depend on the size of the output, k (i.e., the number of intervals containing p); if so, you might express the running time of your approach by saying "it runs in $O(n^2 \log n + k \log k)$." You might want to spend some time thinking about the best possible performance for your algorithm, so that once you've found something good, you don't keep looking to improve it. You should certainly stop looking around after you've spent an hour on this problem.