CSE 417

Algorithms & Computational Complexity
Assignment #4
Due: Friday, 2/4/22

1. Prof. Flashbulb proposed the following greedy algorithm for solving the interval partitioning problem (aka interval coloring, aka scheduling all intervals; section 4.1, p 122): apply the greedy algorithm for interval scheduling from the earlier part of section 4.1 (the algorithm on page 118); give all of those intervals one color, remove them from consideration, and recursively apply the same method to the resulting reduced set of intervals (using new colors). Show that this method does *not* give an optimal solution to the problem by giving a counterexample. (Keep it simple; 4 to 6 intervals should suffice.) Explain why your counter example is a counter example.

2. Suppose you are designing a new Computer Science building. It has one very long corridor, along which all faculty offices are located, say at positions $x_1 < x_2 < \cdots < x_n$ (arbitrary real numbers, for simplicity). Of course, faculty will die if their offices are not located within 100.0 feet of a combined espresso cart/WiFi hotspot.

   (a) Briefly describe a simple greedy algorithm which, given the x_i, will find cart/hotspot locations $g_1 < g_2 < \cdots < g_m$ satisfying the above constraint (i.e., for each $1 \le i \le n$ there is a $1 \le j \le m$ such that $|x_i - g_j| \le 100$) while minimizing the total number m of carts/hotspots.

   (b) Explain why your algorithm is correct. In particular, argue that any solution that differs from the one your algorithm specifies can be replaced by one with no additional carts/hotspots that also satisfies the constraints and is more similar to your algorithm's solution.

3. A quick look at the algorithm on page 124 for the interval partitioning problem suggests an $O(n^2)$ time bound: an outer loop for $j = 1, ..., n$, and an implicit inner loop "for each interval I_i" that also seems to need order n time.

   (a) The algorithm on pg 124 appears to assume that d (the *depth* of the problem instance; pg 123) is known in advance. However, the very simple variant of it given on slide 29 (approximately) avoids this assumption, and determines d while computing the partition. (How? Basically, assume the depth is 1 until you are faced with 2 overlapping intervals; assume it is 2 until you see 3, etc., and partition accordingly.) Give a slightly more detailed description of an implementation of the later algorithm that will take only $O(dn)$ time. (For your timing analysis, ignore the $\Theta(n \log n)$ time for the initial step that sorts the intervals by start time.)

   (b) If d is small, the above method typically will be much faster than the simple (n^2) analysis suggests. Show, however, that it is not faster in the worst case. I.e., show that there are instances where it takes time $\Omega(n^2)$. [Note that to show a lower bound like this, it's not good enough to show one bad example; you really need to show that there are infinitely many bad examples, for larger and larger values of n.]

   (c) Give a more clever implementation of the algorithm whose running time is $O(n \log n)$, (even if depth is $\Omega(\log n)$) and justify this time bound. [Hint: use a better data structure.]

4. The interval partition algorithm sorts intervals by start time. At first glance, it seems possible that sorting by *end* time would work just as well. Indeed, most of the statements made in the correctness proof (approximately slide 31 in the greedy slides) remain correct when applied to this modified algorithm (after changing "start time" to "end time" and some other obvious things in a few places).

   (a) Give a counterexample to the correctness of the modified algorithm and explain why it is a counterexample. (Keep it simple; 4 to 6 intervals should suffice.)

   (b) Explain where the proof goes wrong and why. E.g., which statement(s) on slide 31 are false for the modified algorithm (even after fixing the obvious issues like "start time" vs "end time") and why are these statements important?

5. (a) Simulate the Huffman tree algorithm on the input alphabet a, b, c, d, e with frequencies .15, .30, .50, .02, .03, resp. Show the intermediate "trees" built by the algorithm, as well as the final tree.

(b) What will be the total length in bits of the compressed text built from a sequence of 100 letters having the above frequencies using the Huffman code you calculated?

(c) Suppose I tell you that the frequencies of a and b are .15, .30, resp, as above, but I don't tell you the other frequencies. I can quickly tell (e.g., with*out* running the full Huffman algorithm) that the Huffman code for this data is *not* the one shown in figure 4.16 (a) (pg 168). Explain how.

6. The *Fibonacci numbers* are defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ with the initial conditions $F_1 = F_2 = 1$. Thus the first few Fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, 21$.

   (a) Construct a Huffman code for the case of 8 characters, with the i^{th} character having frequency proportional to F_i (i.e., $F_i/D, D = \sum_{i=1}^{8} F_i$). Show the first few steps of the tree construction and show the final tree. Label leaves and internal nodes with relative frequencies (i.e., frequency times D) as in the examples from lecture.

   (b) What is the average number of bits per character attained by this code?

   (c) Describe the Huffman code in the general case where there are n characters, and the i^{th} character has frequency proportional to F_i.