CSE 417
Algorithms & Computational Complexity
Assignment #5 (rev. a)
Due: Friday, 2/15/19

Turnin Instructions: Gradescope again this week; use your @uw email and gradescope password as before.

1. [10 points] Given algorithms with (exactly) the run times listed below, how much slower is each if (i) the input size is increased by 1, or (ii) input size is doubled?

   (a) $n \log_2 n$

   (b) n^2

   (c) $100n^2$

   (d) n^3

   (e) 2^n

2. [10 points] Some friends collect butterflies. One day they return with n butterflies, and they believe that each belongs to one of two different species, called A and B for simplicity. They'd like to divide the n specimens into two groups—those that belong to A, and those that belong to B—but the species look similar, making it very hard to directly label any one specimen. They adopt the following approach.

   They carefully study each pair of specimens i and j side-by-side; if they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

   Given the collection of n specimens, as well as a collection of m judgments (either "same" or "different" for the pairs that were not declared to be ambiguous), they'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B. More concretely, we'll declare the m judgments to be *consistent* if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same," it is the case that i and j have the same label; and for each pair (i, j) labeled "different," it is the case that i and j have opposite labels.

   Give an algorithm with running time $O(m + n)$ that determines whether the m judgments are consistent.

3. [10 points] 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., $F_i / \sum_{i=1}^{8} F_i$).

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

4. [10 points] KT Chapter 5, problem 3, page 246. (.jpg image) Clearly write out a pseudocode version of your (recursive!) algorithm, highlighting the base case(s), etc., very carefully state exactly what your recurrence is counting, then write the corresponding recurrence relation, as outlined in my "Code to Recurrence" section of the D&C slides. You may use the "master recurrence" to solve it (end of the "Recurrences" section of the slides); clearly indicate what the constants a, b, c, d and k given in the master recurrence are for this problem. The emphasis this time is on analysis, but do include a brief correctness argument.

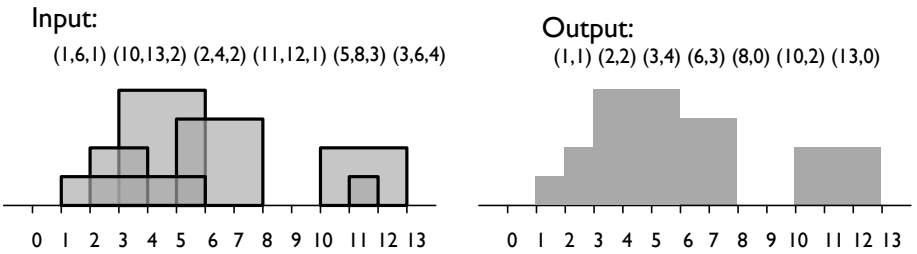
5. [10 points] Give a fast divide and conquer algorithm for the following *silhouette problem*:

   Input: an unordered list of triples (x_l, x_r, h), where $0 < x_l < x_r$ and $0 < h$.

   Problem: Each triple describes a rectangle of height h whose base is on the x axis, and whose left and right sides are at positions x_l and x_r, respectively. Imagine that each rectangle is opaque, and that the whole collection is illuminated from behind. The problem is to calculate the silhouette of the set of rectangles — a description of

the visible edges in the resulting scene. (This is a very simple case of what's called *hidden line elimination* in computer graphics.)

Output: A list of pairs (x_i, y_i), ordered by increasing x_i, of the height y_i at each point x_i where the height of the silhouette changes. (See example below.)



Give a brief English description of your algorithm, plus pseudocode for it. Give a recurrence relation for its running time and solve it. (As in the lecture slides, very carefully state exactly what your recurrence is counting. If the recurrence follows one of the forms solved in class or in the text, just state that and give the solution; otherwise, justify your solution.)

6. **Extra Credit:** For two $n \times n$ matrices of real numbers $A = (a_{i,j})$ and $B = (b_{i,j})$, their product $C = A * B$ is defined by $c_{i,j} = \sum_{1 \le k \le n} a_{i,k} \cdot b_{k,j}$. The obvious algorithm based on this definition takes time $O(n^3)$: $O(n)$ steps to compute each of the $O(n^2)$ entries $c_{i,j}$. To be more precise, focusing just on scalar multiplications (i.e., multiplications of two real numbers), it performs exactly n^3 multiplications. Is there a better way? Having just read the Divide & Conquer chapter, you might think of this:

   MMult (A, B) {
   View A as a 2×2 matrix $(A_{i,j})$ for $1 \le i, j \le 2$, where $A_{1,1}$ is the $n/2 \times n/2$ matrix consisting of the first $n/2$ rows by $n/2$ columns of A, etc. View B, C similarly.
   Calculate:

   $$C_{1,1} = A_{1,1} * B_{1,1} + A_{1,2} * B_{2,1}$$
   $$C_{1,2} = A_{1,1} * B_{1,2} + A_{1,2} * B_{2,2}$$
   $$C_{2,1} = A_{2,1} * B_{1,1} + A_{2,2} * B_{2,1}$$
   $$C_{2,2} = A_{2,1} * B_{1,2} + A_{2,2} * B_{2,2}$$

   Return C
   }

   Here "*" denotes matrix multiplication and "+" just means to add corresponding pairs of matrix elements. If you do the algebra, you will find that this correctly computes $C = A * B$. (No need to turn this in.) In this problem you will look at 4 different versions of MMult, based on how that "*" is computed. You may assume that n is a power of 2.

   (a) **Version 1:** Suppose the eight "*"'s in MMult are directly computed by the "obvious" algorithm suggested in the first paragraph (not by recursively applying MMult). How many scalar multiplications are performed by the algorithm MMult when multiplying two $n \times n$ matrices? (Give an exact count, not a big-O bound.)

   (b) **Version 2:** Suppose instead that "*" in this algorithm is performed by *recursively* calling MMult (unless $n = 1$, in which case A and B are scalars and you just directly multiply them and return their product). Write a recurrence relation for the number of scalar multiplications performed by this algorithm when multiplying two $n \times n$ matrices.

   (c) Solve the above recurrence using the "master recurrence" (at the end of the "Recurrences" section of the D&C slides).

   (d) Solve the recurrence directly, using one of the methods outlined in the text (5.1-5.2) or lecture ("Recurrences" section of the D&C slides, excluding the "master recurrence"); i.e., fun with algebra...

   (e) **Version 3:** In 1969, V. Strassen published a matrix multiplication method akin to MMult above, but using 7 multiplications of $n/2 \times n/2$ matrices instead of the 8 used above (somewhat like the trick in Karatsuba's

algorithm for integer multiplication). Repeat step (a) assuming that the body of MMult, version 1, is replaced by this more clever method (but "*" is again computed by the direct n^3 method). Compare the time bound for this method to that of the n^3 method directly applied to the $n \times n$ problem.

(f) **Version 4:** Repeat steps (b), and (c) (but not (d)) assuming the fancy 7-multiply method is used recursively.

(g) Strassen's method requires more additions than MMult above (15 or so, instead of 4, in the "top level" code shown above). If we count the total number of additions instead of multiplications, is Strassen's method (as in version 4) still better that the "obvious" algorithm? Does your answer depend on n?

Revision History:

Rev a: fixed typos, minor rewording, removed "draft" label. — 2/10/19.