CS 125 Algorithms & Complexity — Fall 2016

Problem Set 1

Due: 11:59pm, Friday, September 9th

See homework submission instructions at `http://seas.harvard.edu/~cs125/fall16/schedule.htm`

Problem 5 is worth one-third of this problem set, and problems 1-4 constitute the remaining two-thirds.

Problem 1

Indicate for each pair of expressions (A, B) in the table below the relationship between A and B. Your answer should be in the form of a table with a "yes" or "no" written in each box. For example, if A is $O(B)$, then you should put a "yes" in the first box. If the base of a logarithm is not specified, you should assume it is base-2.

A	B	O	o	Ω	ω	Θ
$\log_2 n$	$\log_3 n$					
$\log\log n$	$\sqrt{\log n}$					
$2^{\log^7 n}$	n^7					
$n^2 2^n$	3^n					
$n!$	n^n					
$\log(n!)$	$\log(n^n)$					
$(n^2)!$	n^n					
$(n!)^2$	n^n					

Problem 2

In many applications of sorting, the input is not just a list of numbers to be sorted, but rather a list of items, each of which has a *sort key* k_i (which is a number) and a *data payload* d_i (which comes from an arbitrary set). The task is to sort the items according to the sort key. (This is like sorting a spreadsheet by a particular column.) Formally, given an input $(k_1, d_1), \ldots, (k_n, d_n)$ where each $k_i \in \mathbb{N}$, a sorting algorithm should produce a sequence $(k'_1, d'_1), \ldots, (k'_n, d'_n)$ such that (1) $k'_1 \le k'_2 \le \cdots \le k'_n$, and (2) there is a permutation π of $\{1, \ldots, n\}$ such that for all i, $(k'_i, d'_i) = (k_{\pi(i)}, d_{\pi(i)})$.

(a) (3 points) Show how to extend counting sort to solve the above task, sorting in time $O(n + M)$ assuming all of the sort keys are in the range $[0, M)$. Your algorithm should work even if there are repetitions among the sort keys. You can assume that copying of data items d_i can be done in unit time.

(b) (3 points) Show how to ensure that your algorithm is *stable*, in the sense that it does not reorder items with the same sort key. Formally, if $k_i = k_j$ for some $i < j$, then $\pi(i) < \pi(j)$.

(c) (4 points) Another sorting algorithm that can work in $o(n \log n)$ time is Radix Sort. Radix sort works as follows, on numbers represented in binary.

   i. Start with the *last* b bits of the numbers. Use your version of counting sort from part (b) to the sort the numbers using the last b bits as the sort key.

   ii. Continue from right to left looking at the next b bits of the numbers, and sort based on those bits along using counting sort.

   iii. Continue this repeated sorting including through the first b bits.

   Argue that if you use $b = \log_2 n$ and you are sorting n numbers in the range $[0, n^j)$ for some constant j that the total time taken by radix sort is $O(n)$. (Here we assume, as we did in class, that our machine can manipulate numbers of $\log_2 n$ bits with unit cost operations – so that, for example, it can cope with an array of n numbers.) As part of your proof, explain why you need the intermediate sorting steps to be stable.

Problem 3

In class we showed how to speed up integer multiplication via a divide-and-conquer approach: equipartitioning the digits of each of x and y into two sets, then doing three recursive multiplications followed by some insertions and subtractions (Karatsuba's algorithm). The overall runtime was $O(n^{\log_2 3})$. In this problem we will develop a similar, but faster, approach.

In order to speed up integer multiplication, we will first take a slight detour. Let us first consider the problem of solving a system of n linear equations with n variables $x_0, \ldots, x_{n-1}$. Thus the input is $n^2 + n$ numbers $\{a_{i,j}\}$ for $0 \leq i \leq n-1$ and $0 \leq j \leq n$. These represent the n equations $a_{i,0}x_0 + \cdots + a_{i,n-1}x_{n-1} + a_{i,n} = 0$ for $0 \leq i \leq n-1$. Consider the following pseudocode for a function SOLVE(), which solves for the n variables assuming that there is a unique solution. The input is a doubly-indexed array `A` with `A[i][j]` representing $a_{i,j}$ above. Below, we sometimes abuse notation and think of `A[i]` as the vector $(a_{i,0}, a_{i,1}, \ldots, a_{i,n})$.

Algorithm SOLVE(A[0..n-1][0..n]): // coefficients for n equations, n variables

// base case, $n = 1$, corresponds to $a_{0,0}x_0 + a_{0,1} = 0$
if $n = 1$: **return** (-A[0][1]/A[0][0])

// make sure x_0 has coefficient 1 in the 0th equation
let i be the first index with A[i][0] $\neq 0$; swap A[i] with A[0]
A[0] $\leftarrow$ A[0]/A[0][0]

// zero out the coefficient of x_0 in every equation but the 0th one
for $i = 1, \ldots, n$: A[i] $\leftarrow$ A[i]$-$A[0]$\cdot$A[i][0]

// recursively solve $n-1$ equations in $n-1$ variables $x_1, \ldots, x_{n-1}$
$(x_1, \ldots, x_{n-1}) \leftarrow$ SOLVE(A[1..n-1][1..n])

$x_0 \leftarrow -$A[0][n] $- \sum_{j=1}^{n-1} x_j \cdot$A[0][j]
return $(x_0, \ldots, x_{n-1})$

(a) (2 points) Let $T(n)$ denote the worst case running time of SOLVE() on n equations over n variables. Assume all basic arithmetic operations (addition, subtraction, division, and multiplication) are constant time. Write a recurrence for $T(n)$ and solve it.

(b) (2 points) Now let us *not* assume arithmetic operations are unit cost. To implement SOLVE(), we maintain all intermediate computations explicitly as fractions, storing numerators and denominators. Suppose $a_{i,j}$ for $0 \le i, j < n$ are L-digit integers, and the $a_{i,n}$ are each R digits. Prove that there exists a function $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that if one carried out all arithmetic operations in SOLVE() *exactly* by storing fractions explicitly as (numerator, denominator) pairs, then no intermediate numerators or denominators of A[i][0..n-1] values or denominators of A[i][n] values would ever require more than $f(n, L)$ digits, and no intermediate numerators of A[i][n] values would ever require more than $f(n, L) \cdot R$ digits, for any A in any level of recursion. Here $\mathbb{N}$ is the set of natural numbers. Showing the existence of any such f is sufficient for full credit — you do not have to find an optimally slow-growing f. Conclude a bound on the running time of SOLVE() in terms of f, n, L, R.

(c) (4 points) In this problem part we will finally develop a method faster than Karatsuba's algorithm for integer multiplication. Suppose we want to multiply two n-digit positive integers w, y. If $n = 1$, we simply output the answer. Otherwise, we pad w, y with leading zeroes to make n a multiple of 3. Then we write $w = p_w(10^{n/3})$ and $y = p_y(10^{n/3})$, where $p_w(z)$ is the polynomial $w_{hi} \cdot z^2 + w_{mid} \cdot z + w_{lo}$, and similarly for p_y. Here each of w_{hi}, w_{mid}, w_{lo} have $n/3$ digits. For example, if $w = 140712$ then $w_{hi} = 14$, $w_{mid} = 7$, $w_{lo} = 12$. Show how to use p_w, p_y, and SOLVE() to develop an algorithm for integer multiplication faster than Karatsuba's algorithm, and prove a bound on the running time of your method. You may use the result of part (b) even if you didn't solve it. **Not for credit:** what if you tried to break w, y into $k > 3$ parts each?

You may take for granted the fact that for any $d \ge 1$, for any distinct reals $z_0, \ldots, z_d$, and for any (not necessarily distinct) $m_0, \ldots, m_d$, the set of $d+1$ linear equations

$m_j + \sum_{i=0}^{d} x_i z_j^i = 0$ has a unique solution. In other words, there is a unique degree-d polynomial interpolating given values $-m_j$ for any $d+1$ distinct evaluation points z_j.

Problem 4

It is known that every integer $n > 1$ can be uniquely factored as a product of primes. For example, $4 = 2 \times 2$, $6 = 2 \times 3$, and $90 = 2 \times 3 \times 3 \times 5$. Let $p(n)$ be the number of *distinct* prime divisors of n, so $p(6) = 2$ but $p(4) = 1$.

(a) (2 points) Show that $p(n) = O(\log n)$.

(b) (4 points) Show that $p(n) = O(\frac{\log n}{\log \log n})$.

(c) (4 points) It is a fact, which you may assume without proof, that there are $\Theta(t/\log t)$ primes between 1 and t. Use this fact to show that it is *not* true that $p(n) = o(\frac{\log n}{\log \log n})$.

Problem 5 (Programming Problem)

Solve "ZOO" on the programming server `https://cs125.seas.harvard.edu`.
(under "Problem Set 1").