CS 598TMC, Fall 2020
Timothy Chan

Homework 1 (due Sep 30 Wednesday 10:00am (CT))

Instructions: You may work individually or in groups of at most 3; submit one set of solutions per group. Always acknowledge discussions you have with other people and any sources you have used (although most homework problems should be doable without using outside sources). In any case, *solutions must be written in your own words.*

1. [*25 pts*] Consider the following version of the 3SUM problem: given three sets of numbers A, B, and C with $|A|+|B|+|C| = n$, we want to decide whether there exist elements $a \in A$, $b \in B$, and $c \in C$ such that $c = a + b$.

   In class, we have shown that in the special case when $A, B \subseteq [u]$ (where $[u]$ denotes $\{1, \ldots, u\}$), then the problem can be solved in $O(u \log u)$ time by FFT.

   Give an algorithm for a slightly more general case when $A \subseteq [u]$ but B and C can be arbitrary sets of integers. Express the running time as a function of n and u. The time bound should be truly subquadratic in n if u is truly subquadratic in n.

   [Hint: divide into intervals of length u, and consider "light" vs. "heavy" intervals (intervals with $\le d$ vs. $> d$ elements of $B \cup C$, for some parameter d). For the light case, first give a 3SUM algorithm that solves the problem in $\widetilde{O}(|A| + |B||C|)$ time...]

2. [*25 pts*] We are given a pattern string $p_1 \cdots p_m \in \Sigma^*$ of length m and a text string $t_1 \cdots t_n \in \Sigma^*$ of length n. Here, the alphabet is $\Sigma = \{1, \ldots, \sigma\}$. Consider the following "strange" string matching problem: decide whether there exists an index i such that $p_1 \cdots p_m \preceq t_{i+1} \cdots t_{i+m}$, where "$p_1 \cdots p_m \preceq t_{i+1} \cdots t_{i+m}$" means $p_1 \le t_{i+1}$, *and* $p_2 \le t_{i+2}$, ..., *and* $p_m \le t_{i+m}$.

   (Note that comparisons of symbols make sense since we have specified that the alphabet symbols are integers in $\{1, \ldots, \sigma\}$.)

   (a) First describe an $O(\sigma n \log n)$-time algorithm for this problem.

   (b) Describe an $\widetilde{O}(n\sqrt{m})$-time algorithm for this problem.

3. [*25 pts*] We are given a collection of n strings $s^{(1)}, \ldots, s^{(n)} \in (\Sigma \cup \{\texttt{?}\})^*$ with the "don't care" symbol `?`. All strings are of length ℓ.

   We want to determine whether there exist two strings $s^{(i)}$ and $s^{(j)}$ ($i \ne j$) that *match*, where two strings $a_1 \cdots a_\ell$ and $b_1 \cdots b_\ell$ in $(\Sigma \cup \{\texttt{?}\})^*$ are said to match iff for all k, we have $a_k = b_k$ or $a_k = \texttt{?}$ or $b_k = \texttt{?}$.

   The trivial algorithm runs in $O(\ell n^2)$ time. Describe a faster algorithm.

   [Hint: use the idea from Clifford and Clifford's algorithm, but replace convolution with matrix multiplication...]

4. [*25 pts*] Let n be a power of 2. Redefine $[n] = \{0, 1, \ldots, n-1\}$ (in this problem, it will be more convenient to have indices of matrices start at 0).

   Given three binary strings $u = u_1 \cdots u_\ell, v = v_1 \cdots v_\ell, w = w_1 \cdots w_\ell \in \{0,1\}^*$, the triple (u, v, w) is said to be *bad* iff there exists an index h such that $u_h = v_h = w_h = 0$. For example, $(01001, 10101, 01000)$ is bad, because the 4th symbols in the three strings 01001 and 10101 and 01000 are all 0's.

   Similarly, given three numbers $i, j, k \in [n]$, the triple (i, j, k) is said to be *bad* iff $(\phi(i), \phi(j), \phi(k))$ is bad where $\phi(i)$ denotes the binary representation of i (which is viewed as a string of length exactly $\log_2 n$). For example, $(9, 21, 8)$ is bad, because $(01001, 10101, 01000)$ is bad. Note that for $n = 2$, the only bad triple is $(0, 0, 0)$.

   We now introduce a "funny" variant of matrix multiplication: given two $n \times n$ matrices $A = (a_{ij})_{i,j\in[n]}$ and $B = (b_{ij})_{i,j\in[n]}$, define the *funny product* $C = A \star B = (c_{ij})_{i,j\in[n]}$ by the formula:

$$c_{ij} \;=\; \sum_{k\in[n]:\ (i,j,k)\ \text{not bad}} a_{ik} b_{kj}.$$

   The funny product $\star$ is like standard matrix multiplication, but with some terms missing. For example, for $n = 2$, we have

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \star \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{01}b_{10} & a_{00}b_{01} + a_{01}b_{11} \\ a_{10}b_{00} + a_{11}b_{10} & a_{10}b_{01} + a_{11}b_{11} \end{pmatrix},$$

   which differs from the standard matrix product in that the upper left output entry is missing the term $a_{00}b_{00}$ (since $(0, 0, 0)$ is bad).

   Prof. K came up with a variant of Strassen's formulas that computes the funny product for $n = 2$ using 6 multiplications, amazingly![1]

   (a) For arbitrary n (that is a power of 2), we can compute the funny product $A * B$ by divide-and-conquer, i.e., dividing each of A and B into $(n/2) \times (n/2)$ submatrices, and using Prof. K's formulas (where elements are $(n/2) \times (n/2)$ submatrices and multiplication of elements are funny products, computed recursively). You don't need to justify correctness of this divide-and-conquer algorithm.

[1] It is not important to know the precise formulas, but for those who are interested, here they are (which you can verify, assuming that I didn't make a mistake):

$$\begin{aligned}
p_1 &= (a_{10} + a_{11})(b_{10} + b_{11}) \\
p_2 &= a_{01}b_{10} \\
p_3 &= (a_{01} + a_{11})(b_{01} - b_{11}) \\
p_4 &= (a_{10} + a_{01} + a_{11})(b_{10} - b_{01} + b_{11}) \\
p_5 &= (a_{00} + a_{10} + a_{01} + a_{11})b_{01} \\
p_6 &= a_{10}(b_{00} - b_{10} + b_{01} - b_{11}) \\
c_{00} &= p_2 \\
c_{01} &= p_5 - p_2 + p_4 \\
c_{10} &= p_6 - p_2 + p_4 + p_3 \\
c_{11} &= p_1 + p_2 - p_4 - p_3
\end{aligned}$$

Analyze the running time $T_\star(n)$ of this divide-and-conquer algorithm (which should be better than $O(n^{2.81})$ from Strassen's original algorithm).

(b) Unfortunately, the funny product doesn't seem to help solve the standard matrix multiplication prolem. But as you will see, it is useful in solving the *Boolean* matrix multiplication problem: given $n \times n$ Boolean matrices $A = (a_{ij})_{i,j\in[n]}$ and $B = (b_{ij})_{i,j\in[n]}$ with $a_{ij}, b_{ij} \in \{0,1\}$, compute $C = A \bullet B = (c_{ij})_{i,j\in[n]}$ where $c_{ij} = \bigvee_{k\in[n]} a_{ik} \wedge b_{kj}$.

Take random permutations $\pi, \pi', \pi'' : [n] \to [n]$. Consider a fixed $i, j \in [n]$. Prove that

(i) if $\bigvee_{k\in[n]} a_{ik} \wedge b_{kj}$ is false, then $\sum_{k\in[n]:\ (\pi(i),\pi'(j),\pi''(k))\ \text{not bad}} a_{ik}b_{kj}$ is always zero;

(ii) if $\bigvee_{k\in[n]} a_{ik} \wedge b_{kj}$ is true, then $\sum_{k\in[n]:\ (\pi(i),\pi'(j),\pi''(k))\ \text{not bad}} a_{ik}b_{kj}$ is nonzero with probability at least $(7/8)^{\log_2 n}$.

Thus, by randomly permuting the rows and columns of the matrices, we can solve the Boolean matrix multiplication problem by a randomized algorithm in $O(T_\star(n))$ time where each output entry is correct with probability at least $(7/8)^{\log_2 n}$.

(c) Unfortunately, the correctness probability from part (b) is low (the algorithm is wrong most of the time!). Describe how to reduce the chance of error, so that the resulting algorithm computes the entire output matrix is correct with probability at least $1 - 1/n^{100}$. [Hint: repeat $\widetilde{O}((8/7)^{\log_2 n})$ times...]

(d) Putting parts (a)–(c) together, what final time bound do you get for Boolean matrix multiplication? (It should hopefully be better than $O(n^{2.81})$.)