CS 170 Algorithms HW 4
Fall 2013 Satish Rao

1. (10 pts) Problem 2.33

a) Let $M_{ij} \neq 0$. It is sufficient to bound from above $\Pr[(Mv)_i = 0]$, as $\Pr[(Mv)_i = 0] \geq \Pr[Mv = 0]$ (since $Mv = 0$ implies that $(Mv)_i = 0$). Now, $(Mv)_i = \sum_{t=1}^{n} M_{it} v_t$, so $(Mv)_i$ is 0 if and only if:

$$M_{ij} v_j = \sum_{\substack{t=1 \\ t\neq j}}^{n} M_{it} v_t$$

Consider now any fixed assignment of values to all the v_t's but v_j. If, under this assignment, the right hand side is 0, v_j has to be 0. Similarly, if the right hand side is non-zero, v_j cannot be 0. In either case, v_j can take at most one of the two values $\{0,1\}$ for the equation to hold. We therefore see that for any vector $x \in \{0,1\}^{n-1}$, the conditional probability $\Pr[(Mv)_i = 0 | (v_1, v_2, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n) = x]$ is at most $\frac{1}{2}$. By expanding $\Pr[(Mv)_i = 0]$ in terms of these conditional probabilities (and remembering that $\Pr[(v_1, v_2, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n) = x] = \frac{1}{2^{n-1}}$, because v is chosen uniformly at random from $\{0,1\}^n$), we then have

$$\begin{aligned}\Pr[Mv = 0] \leq \Pr[(Mv)_i = 0] &= \sum_{x\in\{0,1\}^{n-1}} \Pr[(Mv)_i = 0 | (v_1, v_2, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n) = x] \\ &\qquad \cdot \Pr[(v_1, v_2, \ldots, v_{j-1}, v_{j+1}, \ldots, v_n) = x] \\ &\leq \sum_{x\in\{0,1\}^{n-1}} \left(\frac{1}{2} \cdot \frac{1}{2^{n-1}}\right) = \frac{1}{2}.\end{aligned}$$

b) If $AB \neq C$, the difference $M = AB - C$ is a non-zero matrix and, by part a):

$$\Pr[ABv = Cv] = \Pr[Mv = 0] \leq \frac{1}{2}$$

The randomized test for checking whether $AB = C$ is to compare ABv with Cv for a random v constructed as in a). To compute ABv, it is possible to use the associativity of matrix multiplication and compute first Bv and then $A(Bv)$. This algorithm performs 3 matrix-vector multiplications, each of which takes time $O(n^2)$, while the final comparison takes time $O(n)$. Hence, the total running time of the randomized algorithm is $O(n^2)$.

2. (10 pts) Fast Fourier Transform

(a) Our four inputs will be $(x_1, x_2, x_3, x_4) = (\omega^0, \omega^1, \omega^2, \omega^3) = (1, i, -1, -i)$.

(b) $A_e(x) = 1 - x$ and $A_o(x) = 2 + 3x$ fit the bill.

(c) The x-values are 1 and -1. *(Note: these are the squares of the values from part (a).)*

3. (10 pts) Fast Fourier Troubles

(a) Ben can recalculate $A(\omega^i)$ in linear time by rewriting $A(x)$:

$$\begin{aligned} A(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\dots)) \end{aligned}$$

Calculating ω^i takes at most n multiplications. With the above representation of $A(x)$, evaluating $A(x)$ at ω^i take $O(n)$ multiplications and $O(n)$ additions. Thus, this algorithm takes $O(n)$ time.

Alternative solution: Another solution is to use the fact that $\sum_{k=0}^{n-1} \omega^{jk} = 0$ for $1 \le j \le n-1$. We therefore have

$$\sum_{\substack{0\le k\le n-1 \\ k\ne i}} A(\omega^k) = \sum_{\substack{0\le k\le n-1 \\ k\ne i}} \sum_{j=0}^{d} a_j\omega^{kj} \tag{1}$$

$$= \sum_{j=0}^{d} a_j \sum_{\substack{0\le k\le n-1 \\ k\ne i}} \omega^{kj} \tag{2}$$

$$= (n-1)a_0 + \sum_{j=1}^{d} a_j \left(\left(\sum_{k=0}^{n-1} \omega^{kj} \right) - \omega^{ij} \right) \tag{3}$$

$$= na_0 - a_0 - \sum_{j=1}^{d} a_j\omega^{ij} = na_0 - A(\omega^i). \tag{4}$$

Thus, Ben can merely take the sum $S = \sum_{\substack{0\le k\le n-1 \\ k\ne i}} A(\omega^k)$ of his remaining entries, and calculate $A(\omega^i) = na_0 - S$. Calculating S takes $O(n)$ additions, and then finding $A(\omega^i)$ through the above equation takes one multiplication and one subtraction, giving a total running time of $O(n)$ additions and $O(1)$ multiplications.

(b) **SOLUTION 1:** The Fourier Transform and the inverse Fourier Transform have a nice symmetry. Calculating c_j is equivalent to evaluating the polynomial $\frac{1}{n}\sum_{i=0}^{n} A(\omega^i)B(\omega^i)x^i$ at $x = \omega^{-j}$. Ben already has $A(\omega^j)B(\omega^j)$ values for $0 \le j \le n$, so he can use the method of polynomial evaluation from part a to calculate c_j in $O(n)$ time.

SOLUTION 2: We have the value $C(1)$, and all the other coefficients, so we can take $C(1)$ and subtract all the c_js to get the missing one. This takes $O(n)$ time.

SOLUTION 3: Notice that $c_j = \sum_{k=0}^{j} a_kb_{j-k}$. This takes $O(n)$ additions and multiplications so this takes $O(n)$ time.

SOLUTION 4: Since we have the coefficients of C except one, we have $C(x) + c_jx^j = c_0 + c_1x + c_{j-1}x^{j-1} + c_{j+1}x^{j+1} + \ldots + c_dx^d$. Thus, we can choose an $\omega^i, i \ne j$, and compute the right hand side using Horner's rule, then subtract the known value of $C(\omega^i)$, and divide by $\omega^{i \cdot j}$. Notice each operation takes at most $O(n)$ time, and combining them takes constant time, so we have the value of c_j in $O(n)$ time.

4. (10 pts) Fast Pattern Matching

(a) For each of $i \in \{1,2,\ldots,n-m\}$ starting points in s_2, check if the substring $s_1[i : i+m]$ differs from s_2 in at most k positions. The check takes $O(m)$ time at each of $O(n)$ starting points, so the time complexity is $O(nm)$.

(b) We extend on the idea from the previous algorithm. We wish to compute the "score" of s_1 along each position in s_2. To do this, we make three very important observations:

- Use convolution: This allows us to compute the score the pattern s_1 along each position s_2 in $O(n \log n)$ time.
- Change the alphabet from $\{0,1\}$ to $\{-1,1\}$. The reason for this is that matches will be scored as $(-1)^2 = 1^2 = 1$, but not equal to mismatches which are $(-1)1 = -1$.
- Notice that convlution does a flip and shift operation. However, we don't want the flip. We only want the shift. Thus, the solution is just to flip s_1 beforehand.

Using these three obervations, we now sketch out our solution. First, we define pattern s'_1 of length n so that $s'_1(i) = -1$ if $s_1(m-i) = 0$, $s'_1(i) = 1$ if $s_1(m-i) = 1$ (the $m-i$ reverses the string for us) and $s'_1(i) = 0$ if $i > m$ (padding). Similarly, set s'_2 to be s_2 , with all the zero bits replaced by -1. Now we seek to compute the convolution

$$c(i) = \sum_{k=1}^{i} s'_1(i) \cdot s'_2(i-k)$$

We can compute this convolution via an FFT, a point-wise product and an inverse FFT. Notice that the only interesting region of c is between $m-1 \le i \le n-1$ so we only need to check those indices (this is because any thing outside of this interval will try to match s_1 outside the boundary of s_2). Notice that if we have k matches, our sum at $c(i)$ will be $m-2k$, thus, if in the resulting solution we get any $c(i) \ge m-2k$ for $i \ge k$, then we have found a solution where the matched pattern is $s_2[i-m+1 : i]$.

The running time of this algorithm, $O(n \log n)$, is dominated by the FFT and inverse FFT steps, which each take $O(n \log n)$ time. The point-wise product and search for $c(i) \ge m-2k$ each take $O(n)$ time.

Note: Some students assumed that s_1 must be entirely contained in s_2 to be considered for a match (i.e. no padding). Since the problem was not clear about this point, these solutions were accepted.

5. (15 pts) Problem 3.7

(a) Let us identify the sets V_1 and V_2 with the colors red and blue. We perform a DFS on the graph and color alternate levels of the DFS tree as red and blue (clearly they must have different colors). Then the graph is bipartite iff there is no monochromatic edge. This can be checked during DFS itself as such an edge must be a back-edge, since tree edges are never monochromatic by construction and DFS on undirected graphs produces only tree and back edges.

This algorithm consists of a slightly modified DFS, which takes linear time. Thus, our algorithm also takes linear time ($O(|V|+|E|)$).

(b) The “only if” part is trivial since an odd cycle cannot be colored by two colors. To prove the “if” direction, consider the run of the above algorithm on a graph which is not bipartite. Let u and v be two vertices such that (u,v) is a monochromatic back-edge and u is an ancestor of v. The path length from u to v in the tree must be even, since they have the same color. This path, along with the back-edge, gives an odd cycle.

(c) If a graph has exactly one odd cycle, it can be colored by 3 colors. To obtain a 3-coloring, delete one edge from the odd cycle. The resulting graph has no odd cycles and can be 2-colored. We now add back the deleted edge and assign a new (third) color to one of its end points.

6. (5 pts) **Problem 3.11** The graph has a cycle containing $e = (u,v)$ if and only if u and v are in the same connected component in the graph obtained by deleting e. Thus, we remove the edge $e = (u,v)$ from G and then run DFS starting at node v. If node u is reached at any point in the DFS, return true. Otherwise, return false.

Removing the edge e from an adjacency matrix takes constant time and DFS takes linear time. Thus, this algorithm takes linear time.