Solutions to Assignment 2

1. **[20 marks]** You are given two polynomials,

$$P_A(x) = A_0 + A_3x^3 + A_6x^6$$

and

$$P_B(x) = B_0 + B_3x^3 + B_6x^6 + B_9x^9$$

where all $A_i's$ and all $B_j's$ are large numbers. Multiply these two polynomials using only 6 large number multiplications.

Solution Substitute $y = x^3$ to obtain $P_A^*(y) = A_0 + A_3y + A_6y^2$ and $P_B^*(y) = B_0 + B_3y + B_6y^2 + B_9y^3$; their product $P_C^*(y) = P_A^*(y)\,P_B^*(y)$ is of degree 5 and is thus uniquely determined by six values of y. Compute $P_A^*(y)$ and $P_B^*(y)$ for $y = -2, -1, 0, 1, 2, 3$ or for $y = -3, -2, -1, 0, 1, 2$ and multiply these values using 6 large number multiplications. Let the product polynomial be $P_C^*(y) = C_0 + C_1y + C_2y^2 + C_3y^3 + C_4y^4 + C_5y^5$ where $C_j's$ are some unknown coefficients. Set 6 linear equations of the form $P_C^*(y_k) = P_A^*(y_k)\,P_B^*(y_k)$ where $y_k \in \{-2, -1, 0, 1, 2, 3\}$ or $y_k \in \{-3, -2, -1, 0, 1, 2\}$, and the right side are the 6 obtained products. The solutions of this system produces the coefficients of $P_C^*(y)$; now replace back y with x^3 to obtain $P_C(x) = P_A(x)\,P_B(x)$.

2. (a) **[5 marks]** Multiply two complex numbers $(a + ib)$ and $(c + id)$ (where a, b, c, d are all real numbers) using only 3 real number multiplications.

   (a) **[5 marks]** Find $(a + ib)^2$ using only two multiplications of real numbers.

   (b) **[10 marks]** Find the product $(a+ib)^2(c+id)^2$ using only five real number multiplications.

   Solution

   (a) $(a + ib)(c + id) = ac - bd + i(ad + bc)$; thus, we compute ac, bd and $(a + b)(c + d) = ac + bc + ad + bd$. We now obtain $ac - bd$ and $ad + bc = (a + b)(c + d) - ac - bd$ without any additional multiplications.

   (a) $(a + ib)^2 = a^2 - b^2 + 2iab = (a + b)(a - b) + 2i\,ab$ which requires only two multiplications.

   (b) $(a + ib)^2(c + id)^2 = ((a + ib)(c + id))^2$; by (a) only 3 multiplications are needed to obtain $(a + ib)(c + id)$ and by (b) only 2 additional multiplications to square the result.

3. (a) **[2 marks]** *Revision:* Describe how to multiply two n-degree polynomials together in $O(n \log n)$ time, using the Fast Fourier Transform (FFT). You do not need to explain how FFT works – you may treat it as a black box.

(b) In this part we will use the Fast Fourier Transform (FFT) algorithm described in class to multiply multiple polynomials together (not just two). Suppose you have K polynomials $P_1, \ldots, P_K$ so that

$$\text{degree}(P_1) + \cdots + \text{degree}(P_K) = S > 1.$$

(i) **[6 marks]** Show that you can find the product of these K polynomials in $O(KS \log S)$ time.
Hint: How many points do you need to uniquely determine an S-degree polynomial?

(ii) **[12 marks]** Show that you can find the product of these K polynomials in $O(S \log S \log K)$ time.
Hint: consider using divide-and-conquer; a tree which you used in the previous assignment might be helpful here as well. Also, remember that if x, y, z are all positive, then $\log(x + y) < \log(x + y + z)$

Solution

(a) Call our two polynomials P and Q. Since both have degree (at most) n, their product PQ will have degree at most $2n$. Thus, it suffices to know the value of PQ at $2n + 1$ distinct points to uniquely determine it.

We use FFT to evaluate P and Q at values which are the roots of unity of order which is the smallest power of two no less than $2n + 1$; this can be done in $O(n \log n)$ time. This converts the polynomials from coefficient form to value form. We then multiply the value of P with the value of Q at each point to obtain the value form of PQ (at these points). We then use the Inverse FFT to retrieve PQ in the coefficient form in $O(n \log n)$, as required.

(b) We obtain the products $\Pi(i) = P_1(x) \cdot P_2(x) \ldots \cdot P_i(x)$ for all $1 \leq i \leq K$ by a simple recursion. Initially, $\Pi(1) = P_1(x)$, and for all $i < K$ we clearly have $\Pi_{i+1} = \Pi(i) \cdot P_{i+1}(x)$. At each stage, the degree of the partial product $\Pi(i)$ and of polynomial $P_{i+1}(x)$ are both less than S, so each multiplication, if performed using fast evaluation of convolution (via the FFT) is bounded by the same constant multiple of $S \log S$. We perform K such multiplications, so our total time complexity is $O(KS \log S)$, as required.

Alternatively, pad all polynomials to a degree which is the smallest power of 2 larger than S and find the FFT of all of these polynomials, which will be K sequences of fewer than $2S$ many values. This requites taking K FFTs each in time $\Theta(S \log S)$ so in total $\Theta(KS \log S)$ many operations. Now multiply point-wise all these K sequences, using $(K - 1)S$ multiplications to get a single sequence of length S. Take the Inverse FFT of that sequence in time $\Theta(S \log S)$. Thus, in total the complexity is $\Theta(K\, S \log S)$.

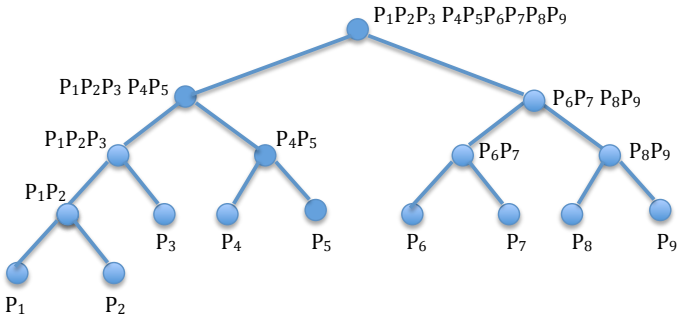

Figure 1: Here $K = 9$; thus, $m = \lfloor \log_2 K \rfloor = 3$ and we add two children to $K - 2^m = 9 - 2^3 = 1$ leaf of a perfect binary tree with 8 leaves. Thus obtained tree has 9 leaves.

(c) The easiest way is, just as in the Celebrity Problem, to organize polynomials and their intermediate products into a complete binary tree, a trick which is useful in many situations which involve recursively operating on pairs of objects. To remind you of the construction of a complete binary tree, we first compute $m = \lfloor \log_2 K \rfloor$ and construct a perfect binary tree with $2^m \leq K$ leaves (i.e., a tree in which each node except the leaves has precisely two children and all the leaves are at the same depth). If $2^m < K$ add two children to each of the leftmost $K - 2^m$ leaves of such a perfect binary tree. In this way you obtain $2(K - 2^m) + (2^m - (K - 2^m)) = 2K - 2^{m+1} + 2^m - K + 2^m = K$ leaves exactly, but each leaf now has its pair, and the depth of each leaf is either $\lfloor \log_2 n \rfloor$ or $\lfloor \log_2 n \rfloor + 1$, see the picture on the next page. Each leaf is now assigned one of the polynomials and the inner nodes of the tree represent partial products of polynomials corresponding to the two children. Note that, with the possible exception of the deepest level $\lfloor \log_2 n \rfloor + 1$ of the tree (which in the example on the picture contains only two polynomials), the sum of the degrees of polynomials on each level is equal to the sum of the degrees of all K polynomials $P_i(x)$, i.e. is equal to S. Let d_1 and d_2 be the degrees of two polynomials corresponding to the children of a node at some level k; then the product polynomial is of degree $d_1 + d_2$ and thus it can be evaluated (using the FFT to compute the convolution of the sequences of the coefficients) in time $O((d_1 + d_2) \log(d_1 + d_2))$ which is also $O((d_1 + d_2) \log S)$, because clearly $\log(d_1 + d_2) \leq \log S$. Adding up such bounds for all products on level k we get the bound $O(S \log S)$, because the degrees of all polynomials at each of the levels add up precisely to S. Since we have $\lfloor \log K \rfloor + 1$ levels we get that the total amount of work is $O(\log K \cdot S \cdot \log S)$, as required.

4. Let us define the Fibonacci numbers as $F_0 = 0$, $F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. Thus, the Fibonacci sequence looks as follows: 0, 1, 1, 2, 3, 5,

8, 13, 21, ...

(a) **[5 marks]** Show, by induction or otherwise, that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

for all integers $n \geq 1$.

(b) **[15 marks]** Give an algorithm that finds F_n in $O(\log n)$ time.

Solution

(a) When $n = 1$ we have

$$\begin{aligned} \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 \end{aligned}$$

so our claim is true for $n = 1$.
Let $k \geq 1$ be an integer, and suppose our claim holds for $n = k$ (Inductive Hypothesis). Then

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \end{aligned}$$

by the Inductive Hypothesis. Hence

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1} &= \begin{pmatrix} F_{k+1} + F_k & F_{k+1} \\ F_k + F_{k-1} & F_k \end{pmatrix} \\ &= \begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix} \end{aligned}$$

by the definition of the Fibonacci numbers. Hence, our claim is also true for $n = k + 1$, so by induction it is true for all integers $n \geq 1$.

(b) Let

$$G = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

and suppose we know G^x. Then, we can compute G^{2x} by simply squaring G^x, which takes $O(1)$ time. Since it is enough to compute G^n, we can do so by first computing G^{2^t} for all $t \leq \lfloor \log_2 n \rfloor$: we can do this in $O(\log n)$ steps by repeatedly squaring G.
Then, we can consider the base 2 representation of n: this tells us precisely which G^{2^t} matrices we should multiply together to obtain G^n.

As an alternative (and equivalent) solution, we can proceed by divide and conquer. To compute G^n: if n is even, recursively compute $G^{n/2}$ and square it in $O(1)$. If n is odd, recursively compute $G^{(n-1)/2}$, square it and then multiply by another G: this last step also occurs in $O(1)$. Since there are $O(\log n)$ steps of the recursion only, this algorithm runs in $O(\log n)$.

5. Your army consists of a line of N giants, each with a certain height. You must designate precisely $L \leq N$ of them to be leaders. Leaders must be spaced out across the line; specifically, every pair of leaders must have at least $K \geq 0$ giants standing in between them. Given N, L, K and the heights $H[1..N]$ of the giants in the order that they stand in the line as input, find the *maximum* height of the *shortest* leader among all valid choices of L leaders. We call this the *optimisation* version of the problem.

   For instance, suppose $N = 10, L = 3, K = 2$ and $H = [1, 10, 4, 2, 3, 7, 12, 8, 7, 2]$. Then among the 10 giants, you must choose 3 leaders so that each pair of leaders has at least 2 giants standing in between them. The best choice of leaders has heights 10, 7 and 7, with the shortest leader having height 7. This is the best possible for this case.

   (a) [**8 marks**] In the *decision* version of this problem, we are given an additional integer T as input. Our task is to decide if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than T.

   Give an algorithm that solves the decision version of this problem in $O(N)$ time.

   (b) [**12 marks**] Hence, show that you can solve the optimisation version of this problem in $O(N \log N)$ time.

(a) Notice that for the decision variant, we only care for each giant whether its height is at least T, or less than T: the actual value doesn't matter. Call a giant *eligible* if their height as at least T.

We sweep from left to right, taking the first eligible giant we can, then skipping the next K giants and repeating. We return `true` if the total number of giants we obtain from this process is at least L, or `false` otherwise.

This algorithm is clearly $O(N)$.

(b) Observe that the optimisation problem corresponds to finding the largest value of T for which the answer to the decision problem is `true`.

Suppose our decision algorithm returns `true` for some T. Then clearly it will return true for all smaller values of T as well: since every giant that is eligible for this T will also be eligible for smaller T. Hence, we can say that our decision problem is *monotonic* in T.

Thus, we can use binary search to work out the maximum value of T where our decision problem returns `true`. Note that it suffices to check only heights of giants as candidate answers: the answer won't change between them. Thus, we can sort our heights in $O(N \log N)$ and binary search over these values, deciding whether to go higher or lower based on a run of our decision problem. Since there are $O(\log N)$ iterations in the binary search, each taking $O(N)$ to resolve, our algorithm is $O(N \log N)$ overall.