Solutions to Assignment 2

1. (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 (which is only a small modification of FFT) to retrieve PQ in 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 $1 \le i \le K$ 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.

   (c) There is a "small" omission in the statement of the problem, namely, we assume that all polynomials are of degree at least 1, i.e., none of them is just a constant i.e., of degree 0. 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 $2^m \le K$ 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

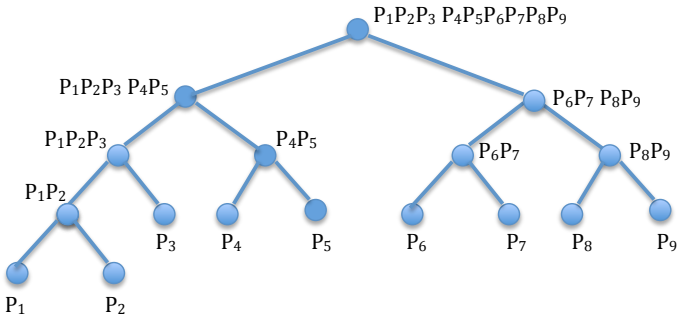

Figure 1: Here $K = 9$; thus, $m = \lfloor \log_2 K \rfloor = 3$ and we add two children to $K - 2^m = 9 - 2^3 = 1$ leaf of a perfect binary tree with 8 leaves. Thus obtained tree has 9 leaves.

also $O((d_1 + d_2) \log S$, because clearly $\log(d_1 + d_2) \leq S$. 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.

2. We start by initializing an array A of length M with all zeros. We then go through all of the N coins one by one, and if the current coin is of value v_i we increment the count n_i of number of coins of value v_i, stored in the cell v_i of the array. We then construct construct the polynomial $P(x) = \sum_{i=1}^{M} A[i]x^i$. Clearly, this polynomial is equal to the polynomial $x^{v_1} + \cdots + x^{v_N}$ because the coefficients in $P(x)$ in front of all powers not equal to some v_i are zero. Then, we can find the square of P, that is, $P^2(x)$ in $O(M \log M)$ time using FFT. Note that a power x^t will be present in $P^2(x)$ (i.e., the coefficient corresponding to the power t will be non zero) just in case there are two values v_i and v_j (not necessarily distinct) such that $v_i + v_j = t$. However, we must rule out the possibility that such a power comes ONLY from a single coin of value $v_i = t/2$ because we draw two coins without replacement so if there is a single coin of value $t/2$ and no other two coins whose values add up to t, then power t does not correspond to a possible outcome of drawing TWO coins simultaneously (i.e., without drawing one coin, returning it and drawing again). But this is easy to rule out - this can happen only if the coefficient in front of power x^t is equal 1; all other options produce the coefficient of at least 2 (because $x^m \cdot x^k + x^k \cdot x^m = 2x^{k+m} = 2x^t$ if $k + m = t$, or, for multiple coins of value $t/2$ the coefficient in front of x^t is at least $2 \cdot 2 = 4$). Thus, the total number of sums obtainable by drawing two coins without replacement is equal to the number of powers x^i in $P^2(x)$ with the coefficient in front of x^i larger than 1.

3. (a) When $n = 1$ we have

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

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

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

by the Inductive Hypothesis. Hence

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

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 $O(\log n)$.

4. Initially we allocate everything to Alice, and then choose which items to move to Bob. Observe that if $N = A + B$, we move items in non-increasing order of $B[i] - A[i]$ to Bob: these are the items that will give us the largest gain.

Now suppose $N < A + B$. Let the number of items such that $B[i] - A[i] > 0$ be p and let n be the number of items such that $B[i] - A[i] \leq 0$. If $p > B$, give

Bob B many items with B many largest (positive) values of $B[i] - A[i]$ and give the the rest to Alice. Otherwise, if $n > A$ give Alice A many items with the smallest (i.e., the most negative) values of $B[i] - A[i]$. If neither of the two options holds, then we must have $p < B$ and $n < A$ in which case we can give all items with $B[i] - A[i]$ positive to Bob and all items with $B[i] - A[i]$ negative to Alice.

The runtime of this algorithm is dominated by the sort, which takes $O(N \log N)$.

5. (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.