COMP3121/9101 19T1 — Assignment 2 (UNSW)

Wenfei Guo , z5135080

1. First, observe that it is enough to be able to multiply any degree 2 polynomial by a degree 3 polynomial using 6 such multiplications: both A and B are really polynomials with those respective degrees in x^3.

   Since the result is a polynomial of degree 5, we can uniquely determine it by determining its values at 6 points. For this we can choose {−3, -2, ...,2} or alternatively, the 6th roots of unity.

   We evaluate A at these 6 points and B at these 6 points: these are only multiplications of a large number my a (constant size) scalar, so these operations are cheap.

   We then multiply the results pointwise: these require precisely 6 large number multiplications. We can then determine the coefficients from these values by setting up a system of linear equations.

   Solving this is done by inverting a constant matrix (as described in the course), so this inversion can even be done by hand, offline and requires no computation. We then multiply the matrix by the vector formed by the pointwise multiplications, which again only multiplies these results by scalars, to give the final polynomial.

2.
   (a) (a + i * b) * (c + i * d)

   Let a1 = ac-bd, a2 = ad+bc
   K1 = ac, K2 = bd, K3 = (a+b)(c+d)
   a1 = K1-K2
   a2 = K3 – K2 – K1
   (a + i * b) * (c + i * d)
   = (a * c - b * d) + i * (a * d + b * c)
   = (a1) + i*(a2)
   = (K1-K2) + i*(K3 – K2 – K1)
   a,b,c,d are all real numbers,
   so K1, K2, K3 are all 3 real number multiplications

(b) (a + i * b)^2

Let K1 = ab, K2 = (a+b)(a-b)

A1= a^2-b^2 = k2
A2 = 2ab = K1 + K1

(a + i * b)^2
= a^2-b^2 + 2abi
= (a+b)(a-b) + 2abi
= K2 + (K1+K1)i
a,b are all real numbers,
so K1, K2 are all 2 real number multiplications

(c) (a + ib)^2 * (c + id)^2
(a + ib)^2 * (c + id)^2 is equal to [(*a*+*ib*)(*c*+*id*)]^2, and can be further converted to [(ac - bd)+(ad + bc)i]^2.

To solve the (ac - bd)+(ad + bc)I part, we can apple the same method as in part(a).
Three multiplications are needed for it, (ac, bd, and (a+b)(c+d)).
Let K1 = ac, K2 = bd, K3 = (a+b)(c+d).
We can find the real part (ac - bd) = (K1-K2) (K3 – K2 – K1), imaginary part (ad + bc) = (K3 – K2 – K1).

To solve [(ac - bd)+(ad + bc)i]^2, we can apple the same method as in part(b).
Because we already have K1 = ac, K2 = bd, K3 = (a+b)(c+d). The function can be expressed as [(K1-K2) + i*(K3 – K2 – K1)]^2, and be converted into [(K1-K2)+(K3 – K2 – K1)] * [(K1-K2) -(K3 – K2 – K1)]+ [2*(K1-K2)*(K3 – K2 – K1)]i.

We apply K4= [(K1-K2), and K5 = (K3 – K2 – K1) to make it easier. We can find
the real part [(K1-K2)+(K3 – K2 – K1)] * [(K1-K2) -(K3 – K2 – K1)]
= (K4+K5) (K4-K5),
imaginary part 2*(K1-K2)*(K3 – K2 – K1) = K4*K5 + K4*K5.

Let M = (K4+K5) (K4-K5), N = K4*K5,
We have K4^2-K5^2 = M, K4*K5 + K4*K5 = N+N

Therefore , we need five multiplications in total.

3. (a)

First, convert the two n-degree polynomials which are chosen to be multiplied in value representation at 2n + 1 distinct points x0,x1,...,x2n. Using FFT to evaluate A and B 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(nlogn) time.

(i.e. PA(x) = A0 + A1x^1+ A2x^2+…+ Anx^n can be convert into
PA(x) <-> {(x0, PA(x0)), (x1, PA(x1)), …, (x2n, PA(x2n))} ;
PB(x) = B0 + B1x^1+ B2x^2+…+ Bnx^n can be convert into
PB(x) <-> {(x0, PB(x0)), (x1, PB(x1)), …, (x2n, PB(x2n))}.)

Then, multiply them point by point using 2n + 1 multiplications in such a form.

(i.e. PA(x) * PB(x) =
{(x0, PA(x0)), (x1, PA(x1)), …, (x2n, PA(x2n))} * {(x0, PB(x0)), (x1, PB(x1)), …, (x2n, PB(x2n))}
= {(x0, PA(x0) PB(x0)), (x1, PA(x1) PB(x1)), …, (x2n, PA(x2n) PB(x2n))}.
We say PC(x0) = PA(x0)PB(x0), PC(x1) = PA(x1)PB(x1) … PC(x2n) = PA(x2n)PB(x2n).
)

Finally, convert such value representation of PC(x) to its coefficient form using IFFT which takes O(nlogn). (IFFT is the Inverse FFT).
(i.e. PC(x) = C0 + C1x^1+ C2x^2+…+ Cn2x^{2n})

(b)

(i) We can get the total products $\prod(i)$ = P1(x)·P2(x)...·Pi(x) for all 1 ≤ i ≤ K (i is integer) by a simple recursion.

From the production equation we can firstly obtain $\prod(1)$ = P1(x), and for all the i < K (i is integer) we clearly can get $\prod(i+1)$= $\prod(i)$* Pi+1(x).

At each stage, the degree of $\prod(i)$and Pi+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 logS.

Because the degree of $\prod(i)$and Pi+1(x) in all stages are less than S, if using Fast Fourier Transform(FFT) to compute polynomial in each multiplication takes O(SlogS)

We have K times such multiplications in total, therefore the total time complexity is O(KSlogS).

(ii) We use **divide-and-conquer** to show that we can obtain the product of these K polynomials in O(S logS logK) time.

Firstly, we assume that all polynomials are of degree at least 1 which means we do not consider polynomials with constant part.

We can use the method we applied in Celebrity Problem, to do divide and conquer multiplying two polynomials at a time with all polynomials being leaves of a complete binary tree.

We initially compute c = [log2 K] and construct a perfect binary tree with 2^c ≤ K leaves (perfect binary tree refers to a tree in which each node except the leaves has precisely two children and all the leaves are at the same depth).

If 2c < K add two children to each of the leftmost K −2c leaves of such a perfect binary tree.

We can get 2(K−2^c)+(2^c−(K−2^c)) = 2K−2^(c+1)+2^c−K+2^c = K leaves by using this method and known the depth of each leaf is either [log 2 n] or [log2 n]+1, also all leaves have its own pair.

Due to the structure of binary search tree, each leaf is assigned one of the polynomials and the inner nodes of the tree represent partial products of polynomials corresponding to the two children.

Note that, for the potential deepest level [log2 n]+ 1 of the tree, the sum of the degrees of polynomials on each level is equal to the sum of the degrees of Pi(x)(i.e. i is integer and 1<= i <=K) which is S.

Let a1 and a2 be the degrees of two polynomials corresponding to the children of a node at some level k.

The product polynomial of degree(Pa1) and degree(Pa2) is of degree a1+a2 degree(Pa1+a2). Therefore we can multiply these two polynomials together in O((a1+a2)log(a1+a2) time, using the Fast Fourier Transform (FFT) to compute.

We can convert O((a1+a2)log(a1+a2) into O((a1+a2)logS, because log(a1 + a2) ≤ S. (x log x + y log y <= x log S + y log S where S is the sum of all degrees.)

We can obtain the bound O(S logS) by adding up such bounds for all products on level k, based on the degrees of all polynomials at each of the levels add up precisely to S which we have proved above. Because we have [logK]+ 1 levels we obtain the product of these K polynomials in O(S logS logK) time.

4.

(a) When n = 1 , we have $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^1$

Base on above equation, we prove that this relationship is right for n=1.

Let m ≥ 1, m is integer, and assume n = m (applying **Inductive Hypothesis**). Then we have

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{m+1} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{m} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} F_{m+1} & F_m \\ F_m & F_{m-1} \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

By the Inductive Hypothesis. Hence

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{m+1} = \begin{bmatrix} F_{m+1} + F_m & F_{m+1} \\ F_m + F_{m-1} & F_m \end{bmatrix}$$
$$= \begin{bmatrix} F_{m+2} & F_{m+1} \\ F_{m+1} & F_m \end{bmatrix}$$

by the definition of the Fibonacci numbers.
Hence, this relationship is right for n = m + 1, so by induction it is true for all integers n ≥ 1.

(b) The algorithm that finds F(n) in O(logn) time.
Firstly, let G = $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ and assume we know G^x.
Then, we can compute G^2x by simply squaring Gx which we supposed to know. It will take O(1) time.
Since it is enough to compute G^n, we can do so by first computing (G^2)^t for all t ≤[log2 n]. It will take O(logn) steps by repeatedly squaring G.
Then, we can consider the base 2 representation of n: this tells us precisely which G2t matrices we should multiply together to obtain Gn.

Then, we can focus on obtain which (G^2)^t matrices should be multiplied together in order to get G^n by finding the t satisfied the nearest 2^t = n.
We can use **divide and conquer** to find the alternative (and equivalent) solution t satisfied the nearest 2^t = n.
If n is even, recursively compute G^n/2 and square it in O(1) to compute G^n.
If n is odd, recursively compute G^(n−1)/2, square it and then multiply by another G. In this case, the last step also occurs in O(1).

There are O(logn) steps of the recursion only, so this algorithm runs in O(logn).

5. (a)
   The goal of our algorithm is to decide if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than T, which means the information of the giant we need to know is if its height is no less or less than T.

   If the height of the particular giant is no less than T, we seem this giant as a valid choice of leaders.

   First, we scan array H from left to right and take the first valid choice of leaders we meet, then skip scan the next K giants in array H and repeat this process.

   If the total number of valid choice of leaders is no less than L, we return true and otherwise false. This algorithm can be considered as **easy greedy**, and the time complexity is O(N).

   (b)
   The optimisation version of the decision problem discussed is to find the largest value of T (i.e. most valid choice of leaders in part(a) that returns true).

   Suppose the decision algorithm will returns true for some T. We can predict that it will return true for all smaller values of T as well. The reason is every giant that is eligible for this T will also be eligible for smaller T.

   Based on above information, we can obtain that the decision problem in part(a) is monotonic in T.

   Thus, we can find the maximum value of T where our decision problem in part(a) returns true by using **binary search**.

   It is enough for us to check giants' height only as candidate answer because the answer will on change.

   Therefore, we can sort our heights in O(N logN) and using binary search to search over all those values to decide whether to go higher or lower based on decision problem.

   There are O(logN) iterations in the binary search, each taking O(N) to resolve, so the optimisation version of this problem in O(N logN) time.