COMP9101 Assignment 2

Name: Xuhua Le
Student Number: z5047516

Question 1.

Solution:

$P_A(x) = A_0 + A_3x^3 + A_6x^6, P_B(x) = B_0 + B_3x^3 + B_6x^6 + B_9x^9$

Firstly, we could observe that it's enough to be able to multiply a degree 2 polynomial by a degree 3 polynomial using 6 such multiplications: both P_A and P_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 6 points only.

For this reason, we could choose $\{-2, -1, 0, 1, 2, 3\}$ or alternatively, the root of unity of order 6. We can evaluate P_A at these 6 points and P_B at these 6 points respectively: these operations are just in linear time as these are only multiplications of a large number with a scalar.

Then we multiply the results point by point: this process requires 6 large number multiplications, we could determine the coefficients from these values by setting up a system of linear equations. Solving these linear equations is done by inverting a constant matrix(same in the lecture), so this inversion process could be done by hand and no computation. We then multiply the matrix by the vector formed by the pointwise multiplications, which also only multiplies these results by scalars, to give the final polynomial. As a result, we could multiply these two polynomials using only 6 large number multiplications.

Question 2.

Solution:

(a).
$(a + ib) * (c + id) = ac + (ad + bc)i - bd = ac + [(a + b) * (c + d) - ac - bd]i - bd$

Hence, we could do only 3 real number multiplications: $ac, bd, (a + b) * (c + d)$.

(b).
$(a + ib)^2 = a^2 + 2abi - b^2$

As $a^2 - b^2 = (a + b) * (a - b)$ is just only 1 real number multiplication, and $2ab$ is just "$ab + ab$" which only needs a real number multiplication ab, then we could do only 2 real number multiplications: $ab, (a + b) * (a - b)$.

(c).
$(a+ib)*(c+id) = ac+(ad+bc)i-bd = ac+[(a+b)*(c+d)-ac-bd]i-bd$

Hence, $(a+ib)*(c+id)$ only needs 3 real number multiplications: $ac, bd, (a+b)*(c+d)$.
After we calculate $(a+ib)*(c+id)$, we could rewrite it as the form: $A+iB$, and $A = ac-bd, B = (a+b)*(c+d)-ac-bd$, and A, B both are already known real numbers.
$\Rightarrow (a+ib)^2*(c+id)^2 = [(a+ib)*(c+id)]^2 = (A+iB)^2 = A^2+2ABi-B^2 = (A+B)*(A-B)+2ABi$
This product needs 2 real number multiplications: $(A+B)*(A-B), AB$.
As a result, the product of $(a+ib)^2*(c+id)^2$ needs 5 real number multiplications.

Question 3.

Solution:

(a).
As the Fast Fourier Transform(FFT) runs in $O(n*\log n)$ time, we could use FFT algorithm to transform this two n-degree polynomials into two corresponding DFT sequences, which is in $O(n*\log n)$ time. Then, we could multiply them point by point, which is in $O(n)$ time. After that, we could use IDFT algorithm to get the corresponding coefficients of the final polynomial, which is in $O(n*\log n)$ time. As a result, we could multiply two n-degree polynomials together in $O(n*\log n)$ time.

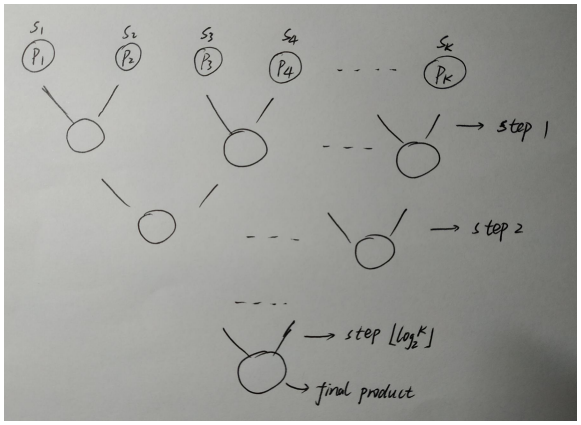
(b).
(i).
We have K polynomials $P_1, ..., P_K$, and $degree(P_1)+...+degree(P_K) = S$, as $degree(P_i) \geq 0, (i = 1, ...K)$, then $0 \leq degree(P_i) \leq S$. Then any polynomial P_i could be transformed into DFT sequence by FFT algorithm, and the running time would not exceed $O(S*\log S)$. Besides, the product of any two different polynomials P_i and P_j could be transformed into DFT sequence, and the running time would not exceed $O(S*\log S)$ as well.

Then, here is the algorithm:
For $i = 1$ to $K-1$, do: (PS: $O(K)$)
Transform P_i and P_{i+1} to their DFT forms by FFT algorithm. (PS: $O(S*\log S)$)
Multiply DFT(P_i) with DFT(P_{i+1}) point by point. (PS: $O(S)$)
Use IDFT to get the coefficients from the multiplication result sequence, and form a result polynomial P'. (PS: $O(S*\log S)$)
$P_{i+1} \leftarrow$ the result polynomial P'. (PS: $O(1)$)
Return P_K. (PS: $O(1)$)

Hence, this algorithm runs in $O(K * S * \log S)$ time.

(ii).
Let's denote the degree of $P_i\,(i = 1, ..., K)$ as S_i, for example, $degree(P_1) = S_1$, hence, $S_1+S_2+...+S_K = S$. As our target is to calculate the product of these K polynomials, then for simplification, we could assume that $S_1 \leq S_2 \leq ... \leq S_K$, and note that in this situation the final product is the same with other situations.

Then, we could set two polynomials P_i and P_{i+1} as a pair (P_i, P_{i+1}) for $i = 1$ to $(K - 1)$ if K is even. If K is odd, we could make $\frac{K-1}{2}$ pairs and a single polynomial P_K. Hence, for every step, we could calculate the multiplication of each pair separately and transmit the last single polynomial to next step if it exists, which is the Divide-and-Conquer method. And this process is like a inverted binary tree as follows:



For step 1: As $degree(P_1) = S_1, degree(P_2) = S_2$, then the multiplication of P_1 and P_2 runs in $O(S_1 * \log S_1) + O(S_2 * \log S_2)$ time as we could use FFT and IDFT algorithm. As $0 \leq S_1 \leq S_2 \leq ... \leq S_K \leq S$, then we have: $S_1 * \log S_1 + S_2 * \log S_2 \leq S_1 * \log S + S_2 * \log S = (S_1 + S_2) * \log S$,
which means the multiplication of P_1 and P_2 runs in $O((S_1 + S_2) * \log S)$ time.
Hence, the step 1 process's running time is:
$O((S_1+S_2)*\log S)+O((S_3+S_4)*\log S)+... = O((S_1+S_2+...+S_K)*\log S) = O(S * \log S)$.

For step 2: It's similar to step 1, the running time is:
$O((S_1 + S_2 + S_3 + S_4) * \log S) + O((S_5 + S_6 + S_7 + S_8) * \log S) + ... = O((S_1 + S_2 + ... + S_K) * \log S) = O(S * \log S)$.

Hence, for each step, it takes $O(S * \log S)$ time.
And we have $\lfloor \log_2 K \rfloor$ steps, then the total running time is: $O(S * \log S * \lfloor \log_2 K \rfloor) = O(S * \log S * \log K)$.
Hence, this algorithm runs in $O(S * \log S * \log K)$ time, which satisfies the re-

quirements.

Question 4.

Solution:

(a).
We could use induction method to prove this equation.
When $n = 1$, we can get:

$$\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1$$

Hence, this equation holds when $n = 1$.

Assume this equation holds for $n = m, (m \geq 1)$, which means:

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

Then, when $n = m + 1$, we could get:

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

Which means:

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

Then, this equation holds for $n = m + 1$.
Hence, as a result, the equation:

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

holds for all integer $n \geq 1$.

(b).
As

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

If we would like to find F_n in $O(\log n)$ time, we could try to calculate $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ in $O(\log n)$ time, then we could get F_n.

As $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n/2} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n/2}$, which means this n-order matrix multiplication could be divided into two identical $n/2$-order matrix multiplication.

Hence, we could only calculate $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n/2}$ and do another matrix multiplication for 1 time, which is the "divide-and-conquer" method.

Assume the time complexity of $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n}$ is $T(n)$, then $T(n) = T(\frac{n}{2}) + O(1)$.
Hence, we get:
$T(n) = T(\frac{n}{2}) + O(1)$.
$T(\frac{n}{2}) = T(\frac{n}{2^2}) + O(1)$.
.....
$T(2) = T(1) + O(1)$.

We could add these equations together, and we get: $T(n) = T(1) + O(1) * (\lfloor \log_2 n \rfloor - 1)$.
$\Rightarrow T(n) = O(\lfloor \log_2 n \rfloor) = O(\log n)$.
Besides, this method is used when n is even for every step. When n is odd for a step, we could divide the n-order matrix multiplication into two identical $\frac{n-1}{2}$-order matrix and a $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}$ matrix as follows:

$$\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n} = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n-1}{2}} * \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n-1}{2}} * \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix},$$

and this process's time complexity is roughly same with the equation above, then the total time complexity would not change, which is $O(\log n)$.

Hence, here is the divide-and-conquer algorithm for this problem:
(1). When n is even, we could divide the n-order matrix multiplication into two identical $\frac{n}{2}$-matrix multiplication, and do another matrix multiplication.
$$\Rightarrow \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n} = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n}{2}} * \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n}{2}}$$
(2). When n is odd, we could divide the n-order matrix multiplication into two identical $\frac{n-1}{2}$-order matrix multiplication and do another matrix multiplication, after that, do another matrix multiplication with $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}$.
$$\Rightarrow \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n} = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n-1}{2}} * \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n-1}{2}} * \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}$$
Repeat (1) and (2).
After that, we could calculate $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n}$, then we could find F_n in $O(\log n)$.

Question 5.

Solution:

(a).
As the every pair of leaders must have at least K giants standing in between them, then we need: $N \geq (K+1)*(L-1)+1$. Hence, if $N < (K+1)*(L-1)+1$, then we could not find such L leaders to satisfy the requirements, then we could end the searching process immediately.

If $N \geq (K+1)*(L-1)+1$, we could set a empty array named "Z".
Then, we could traverse the array H, and as long as we find a giant whose height $\geq T$, then append its index into array Z, and this process takes $O(N)$ time. If $length(Z) < L$, it's obvious that we could not find such L leaders to satisfy the requirements, then we could end the searching process immediately.

If $length(Z) \geq L$, then we have got rid of the influence of T, and array Z is an array whose length is not determined, but we could know that: $L \leq length(Z) \leq N$. As all the indexes of such giants whose height $\geq T$ are stored in array Z, then our aim is to try to find whether array Z has at least L numbers with a spacing of at least K.

It could be proved that if there exists some solutions, then the first element $Z[1]$ in array Z must exists in one of the feasible solutions: if $Z[i]\,(i \geq 2)$ could lead a sequence to satisfy the requirements, then we could simply replace $Z[i]$ with $Z[1]$, then we should get another feasible solution. Besides, if $Z[1]$ could not lead a sequence to meet the requirements, then there could not exist a solution.

Hence, we could treat $Z[1]$ as a starter node and traverse array Z, and count the elements(including $Z[1]$) that have K spaces with the previous node. If the total count $\geq L$, then there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than T, otherwise, there is no solution. As $L \leq length(Z) \leq N$, then this traversal process runs in $O(N)$ time.

As a result, this algorithm runs in $O(N)$ time and satisfies the requirements.

(b).
We could make a copy array H' of array H, and use Merge-Sort algorithm to sort array H', and this process takes $O(N * \log N)$ time.

After that, we could count L numbers from right to left (including the last one) in array H', and the target index is $(N-L+1)$. Then the height of the shortest leader among the L leaders is could not exceed $H'[N-L+1]$, and the interval is: $[H'[1],\, H'[N-L+1]]$.

Then, we could use Binary Search to process the interval $[H'[1],\, H'[N-L+1]]$. Every time you find a value in the interval, we treat it as "T" in the question (a), and use the algorithm in question (a) to check whether it meets the requirements, until we find the largest height which satisfy the requirements. Then we have found the maximum height of the shortest leader among all valid choices

of L leaders.

As the Binary Search process runs in $O(\log N)$ time, and the algorithm in question (a) runs in $O(N)$ time, then this algorithm runs in $O(N * \log N)$ time and meets the requirements.