COMP3121/9101 Assignment 2
z5014567 Senlin Deng

1. 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 $B_j's$ s are large numbers. Multiply these two polynomials using only 6 large number multiplications.

   We denote that $y = x^3$, and substitute y into these two polynomials. Now we have that $P_A(y) = A_0 + A_3y + A_6y^2$ and $P_B(y) = B_0 + B_3y + B_6y^2 + B_9y^3$. Now if we multiply these two polynomials, the degree of the resulted polynomial will be **5**.

   Since the degree of the product is 5, it can be determined uniquely by its values at 6 points. (e.g. we can choose the 6th roots of unity). Now we can evaluate the DFTs of the coefficient sequences of P_A and P_B at those 6 roots of unity. Those operations are cheap, since they are only roots of unity and scalars.

   And then we multiply those results pointwise, which are precisely 6 large number multiplications. We can also further solve this by an inverted constant matrix (e.g. use IDFT to get the coefficient of $P_A(x) * P_B(x)$).

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

   Multiply those two numbers directly: $(a + bi)(c + di) = ac - bd + (ad + bc)i$. Then we can use Karatsuba's trick to the product of $(a + b)(c + d) = ac + bd + (ad + bc)$. Then we have $ad + bc = (a + b)(c + d) - ac - bd$. So that we can substitute $ad + bc$ from the original multiplication.

   We then have $(a + bi)(c + di) = ac - bd + ((a + b)(c + d) - ac - bd)i$. So that we can only use 3 real number multiplications to get the result. (e.g. $\boldsymbol{ac}$, $\boldsymbol{bd}$, $(\boldsymbol{a} + \boldsymbol{b})(\boldsymbol{c} + \boldsymbol{d})$).

   b). Find $(a + bi)^2$ using only two multiplications of real numbers.

   $(a + bi)^2 = a^2 + 2abi - b^2 = a^2 - b^2 + 2abi = (a + b)(a - b) + (ab + ab)i$

   So that a square of a complex number can be computed by only two multiplications of real numbers. (e.g. $(\boldsymbol{a} + \boldsymbol{b})(\boldsymbol{a} - \boldsymbol{b})$ *and* $\boldsymbol{ab}$)

   c). Find the product $(a + bi)^2(c + di)^2$ using only five multiplications of real numbers.

   The equation can be re-written as $((a + bi)(c + di))^2$.

Firstly, we compute the product of two complex numbers $(a + bi)(c + di)$. And the result will still be a complex number. From question a), it can be concluded that the multiplication of two complex numbers can be completed with **3 real number** multiplications.

Then we take the square of the result complex number from last process. We denote $e + fi$ as our result. And we take the square, $(e + fi)^2$. From question b)., this process can be completed by only **2 multiplications** of real numbers.

As a result, this product can be found using only **5 multiplications** of real numbers in total.

3. *a).Revision:* Describe how to multiply two *n*-degree polynomials together in *O(nlogn),* using the FFT. You do not need to explain how FFT works – you may treat it as a black box.

We denote $P_A(x)$ *and* $P_B(x)$ as our two polynomials of degree of *n*. $P_A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{n-1}x^{n-1}$, $P_b(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + \cdots + b_{n-1}x^{n-1}$. Since the product of $P_A(x)$ *and* $P_B(x)$ is a polynomial of degree of 2n – 2 and there are (*2n – 1*) numbers of coefficients. So that we need (*2n – 1*)th roots of unity to evaluate the result polynomial. Thus, we **pad** $P_A(x)$ *and* $P_B(x)$ with **n – 1 zero** at the end.

Respectively, we can compute the **DFTs** of the coefficient sequence of $P_A(x)$ and $P_B(x)$ at (*2n – 1*)th roots of unity using **FFT** methodology. This process takes ***O(nlogn).*** Then we can do pointwise multiplications of these two **DFTs.** This Step takes ***O(n).*** Then we can use the **IDFT** to compute the coefficients of the result polynomial. By using IFFT, this step takes ***O(nlogn).***

Thus, this process can be completed with time ***O(nlogn)*** complexity.

b). In this part we will use the FFT algorithm described in class to multiply multiple polynomials together (not just two).

Suppose you have *K* polynomials $P_1, P_2, \ldots,\ P_k$ so that

$$degree(P_1) + \cdots + degree(P_k) = S$$

i). Show that you can find the product of these K polynomials in *O(K*S*logS)* time.

Since the sum of the degree of all polynomial is S, we have that the product of all polynomials is a polynomial, which has **a degree of S and (S + 1) of coefficients**.

Firstly, we **pad** each polynomial with some zero terms at the end, so that the number of terms of the resulted polynomial is S + 1. (e.g. $P_k(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{k-1}x^{degree(P_k)-1} + \cdots + 0 * x^{S-1} + 0 * x^S$, where $degree(P_k)$ denotes the original degree of P_k.

We then use the same methodology from part a) to compute DFT of each polynomial. Since the degree of padded polynomial is S, the time complexity to compute the DFT of each polynomial is ***O(S*logS)*** (by using the **FFT algorithm** to compute DFT).

And there are K polynomials, so that we need to compute K times of DFT in total. As a result, the time complexity to get all DFTs is *K** ***O(S*logS)***, which is ***O(K*S*logS)***.

We then need to do the multiplication for all polynomial. We multiply two of the polynomials and use the product to multiply another polynomial until there is only one polynomial left. It will take ***O(S)*** for each multiplication, and there are *(K – 1)* multiplications in total. So, this step takes ***O(K*S)***.

Finally, we need to use the resulted DFT to compute the coefficient sequence reversely. From the part a), we can use IDFT by using IFFF algorithm to finish this task. This can be computed in time ***O(S*logS)***.

As a result, in total this process takes ***O(K*S*logS) + O(K*S) + O(S*logS)***, which is ***O(K*S*logS).***

ii). Show that you can find the product of these K polynomials in *O(S*logS*logK)* time.

This problem can be solved by divide-and-conquer and binary tree.

Firstly, we can denote P_i $(where\ i = [1 \ldots k])$ as the bottom layer children of our binary tree. We then compute the product of the polynomials **by pairs** and take the resulted polynomials as our parents of each pair. For example, $P_{12} = P_1 * P_2,\ P_{34} = P_3 * P_4, \ldots,\ P_{(k-1)k} = P_{k-1} * P_k$. So that the time complexity to compute such pair is $(degree(P_{i-1}) + degree(P_i)) * \log\big(degree(P_{i-1}) + degree(P_i)\big)$ (where i denotes the general term). Thus, the total time complexity to compute all of the products will be $\boldsymbol{\sum_{i=1}^{k-1}(degree(P_{i-1}) + degree(P_i)) * \log\big(degree(P_{i-1}) + degree(P_i)\big) \le \sum_{i=1}^{k}(degree(P_i) * \log S) = S * \log S.}$ So that this process can be completed within ***O(S*logS)***.

Then we can do this process recursively by computing the products of each pair, based on reesulted polynomials. We can use the same methodology from above to know that each process can be completed within ***O(S*logS)***. We repeat doing this process until there is only one polynomial left and that one is our result polynomial. And the task is completed.

From the above process, each time we take the products of pairs and we half the amount of the polynomials. So that the total number of the processes will be $\boldsymbol{\lceil \log k \rceil}$, which is the depth of our binary tree.

Overall, the time complexity will be $\boldsymbol{\lceil \log k \rceil * O(S * logS) = O(S * logS * \lceil \log k \rceil)}$, which is $\boldsymbol{O(S * logS * log\ K).}$

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). 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 $n \geq 1$.

   This question can be proved by induction.

   Firstly, we **prove the condition is true when *n* = 1**. When *n* = 1, the LHS matrix is $\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, while the RHS matrix is $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. So that LHS = RHS. Thus, the condition is true when *n* = 1.

   Now we **assume the equation is true when *n* = *k* and prove it is also true for *n* = *k* + *1***, which means we need to prove $\begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1}$

   Since the equation is true for *n* = *k*. We have that $\begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k$. So that, we have $\begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^k * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1}$. Now that we need to prove $\begin{pmatrix} \boldsymbol{F_{k+1}} & \boldsymbol{F_k} \\ \boldsymbol{F_k} & \boldsymbol{F_{k-1}} \end{pmatrix} * \begin{pmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} \end{pmatrix}$ *is equal to* $\begin{pmatrix} \boldsymbol{F_{k+2}} & \boldsymbol{F_{k+1}} \\ \boldsymbol{F_{k+1}} & \boldsymbol{F_k} \end{pmatrix}$.

   By matrix multiplication, we have $\begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{k+1} * 1 + F_k * 1 & F_{k+1} * 1 + F_k * 0 \\ F_k * 1 + F_{k-1} * 1 & F_k * 1 + F_{k-1} * 0 \end{pmatrix} = \begin{pmatrix} F_{k+1} + F_k & F_{k+1} \\ F_{k+1} + F_{k-1} & F_k \end{pmatrix}$. From the definition of Fibonacci sequence, we can further substitute $F_{k+1} + F_k \; and \; F_{k+1} + F_{k-1}$, with $F_{k+2} \; and \; F_{k+1}$ respectively. So that $\begin{pmatrix} F_{k+1} + F_k & F_{k+1} \\ F_{k+1} + F_{k-1} & F_k \end{pmatrix} = \begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix}$, which means $\begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix}$.

   From the above, we have that $\begin{pmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1}$, so that $\begin{pmatrix} \boldsymbol{F_{k+2}} & \boldsymbol{F_{k+1}} \\ \boldsymbol{F_{k+1}} & \boldsymbol{F_k} \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} \end{pmatrix}^{\boldsymbol{k+1}}$.

   So, we have proved that the equation holds when *n* = *k* + *1*. Thus, **by induction, 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 integers *n* ≥ 1**.

b). Hence or otherwise, give an algorithm that finds F_n in *O(logn)* time.

From the part a), we have proved that $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ holds for all integers greater or equal to one. So that, F_n can be found by computing the matrix $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ and F_n is the element in the position (1,1) of the resulted matrix.

Thus, **to find F_n in *O(logn)* time is equivalent to compute $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ in *O(logn)* time**. And we further denote $M^n\ as\ \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$, and also, we can perform the multiplication of every two matrices in constant time *O(1)*.

If *n* is a perfect power of 2:
To compute M^n, we can calculate $M^{\frac{n}{2}}$ and multiply it with itself (e.g. $M^n = M^{\frac{n}{2}} * M^{\frac{n}{2}}$). And we can solve this recursively until we compute $M^2 = M^1 * M^1$. Each multiplication takes *O(1)* and we assumed that n is a perfect power of 2, so there are $\log n$ multiplications in total. Thus, the time complexity is $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$.

If n is not a perfect power of 2:
We can rewrite n as $n = 2^{\lfloor \log n \rfloor} + m < 2^{\lfloor \log n \rfloor + 1}$. So that $m < 2^{\lfloor \log n \rfloor + 1} - 2^{\lfloor \log n \rfloor} = 2^{\lfloor \log n \rfloor}$. And we can do this process recursively (e.g. $m = 2^{\lfloor \log n \rfloor - 1} + m_0$). And rewrite n as $n = 2^{\lfloor \log n \rfloor} + 2^{\lfloor \log m \rfloor} + 2^{\lfloor \log m_0 \rfloor} + \ ... + 1$. We can then use the same methodology to compute M^n, which takes $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n} + \mathbf{log}\,\boldsymbol{m} + \mathbf{log}\,\boldsymbol{m_0} + \cdots + \mathbf{1})$. And $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n} + \mathbf{log}\,\boldsymbol{m} + \mathbf{log}\,\boldsymbol{m_0} + \cdots + \mathbf{1})$ can be taken as $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$, since $\log n$ is the largest term.

As a result, we can compute a matrix of power n with $O(\log n)$ no matter if n is a perfect power of 2. And we can further determine the Fibonacci number F_n by enumerating the element on (1,1) position of the resulted matrix, which means **we can compute $\boldsymbol{F_n}$ with $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$ time complexity**.

5. Your army consists of a line of N giants, each with a certain height. You must designate precisely L ≤ N of them to be leaders. Leaders must be spaced out across the line; specifically, every pair of leaders must have at least K ≥ 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). 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.

Firstly, we denote P as our resulted list which stores the candidates of leaders satisfying the constrains. We **iterate the list H** and check if each element is greater than or equal to the given additional integer T. Furthermore, if we find such element $\boldsymbol{H[i] \geq T}$, we further check if $\boldsymbol{H[i+k+1] \geq T}$. If so, $H[i]$ can potentially be a leader of given T. Then we can put $H[i]$ in our list P.

If the size of P is greater than or equal to L, there exists some valid choice of potential leaders.

Since the process is performed by iterating the list H, this progress can be completed in N steps (where N is the length of the list). Thus, the time complexity is *O(N)*.

b). Hence, show that you can solve the optimisation version of this problem in O(N logN) time.

Firstly, we sort the list H in **non-increasing** order using **merge sort**. This step takes *O(N logN)*. Then we denote H' as our sorted list.

Then we can do **binary search** on H'. For each currently traced element $H'[i]$, we check if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than $H'[i]$, which return us to use the algorithm from part a). And we can determine whether $H'[i]$ is a candidate. As a result, this process can be finished with ***O(logN)*** for binary search and ***O(N)*** to determine whether $H'[i]$ satisfies the constrains or not. By determining whether $H'[i]$ is the maximum height of the shortest leader, we check if $H'[i+1]$ is a candidate. If so, $H'[i]$ is not our finial answer. Otherwise $H'[i]$ is the result.

From the above algorithm, this process can be computed in ***O(N logN)*** in total.