QUESTION 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 are large numbers. Multiply these two polynomials using only 6 large number multiplications.

- For $P_A(x)$ -> the degree of the polynomial is 6
- For $P_B(x)$ -> the degree of the polynomial is 9
- If we multiply these two polynomials together, $P_A(x)P_B(x)$, we need (15+1) values to evaluate it which is greater/larger than the '6 large number multiplications' in the requirements.
- However, we can multiply any degree 2 polynomial by a degree 3 polynomial by using 6 multiplications

$$P_A(y) = A_0+A_3y + A_6y^2$$
$$P_B(y) = B_0+B_3y + B_6y^2+B_9y^3$$

- Now, when we multiply these two polynomials together, $P_A(y)P_B(y)$, the degree of the result should be (2+3=5), thus we can uniquely determine it by determining its value at 6 points. For this we can choose integers {-3, -2, -1, 0, 1, 2}, or we can choose the root of unity of order 6. We then evaluate P_A, P_B at these 6 points.

$$P_A(-3) = A_0+A_3(-3) + A_69$$
$$P_A(-2) = A_0+A_3(-2) + A_64$$
$$P_A(-1) = A_0+A_3(-1) + A_6$$
$$P_A(0) = A_0$$
$$P_A(1) = A_0+A_3 + A_6$$
$$P_A(2) = A_0+A_32 + A_64$$

$$P_B(-3) = B_0+B_3(-3) + B_69+B_9(-27)$$
$$P_B(-2) = B_0+B_3(-2) + B_64+B_9(-8)$$
$$P_B(-1) = B_0+B_3(-1) + B_6+B_9(-1)$$
$$P_B(0) = B_0$$
$$P_B(1) = B_0+B_3 + B_6+B_9$$
$$P_B(2) = B_0+B_32 + B_64+B_98$$

- Then we multiply these results that we got from previous step, pointwise, and this step requires 6 large multiplications.

$$P_c(-3) = (A_0+A_3(-3) + A_69) * (B_0+B_3(-3) + B_69+B_9(-27))$$
$$P_c(-2) = (A_0+A_3(-2) + A_64) * (B_0+B_3(-2) + B_64+B_9(-8))$$
$$P_c(-1) =(A_0+A_3(-1) + A_6)*(B_0+B_3(-1) + B_6+B_9(-1))$$
$$P_c(0) = A_0*B_0$$
$$P_c(1) = (A_0+A_3 + A_6) * (B_0+B_3 + B_6+B_9)$$
$$P_c(2) = (A_0+A_32 + A_64) * (B_0+B_32 + B_64+B_98)$$

- We can then determine the coefficient from the values by inverting the system of linear equations to constant matrix, then we can multiply the matrix by the vector that formed by the pointwise multiplications, which only multiply these results by scalars to give the final polynomial.

(Q2 -> next page)

QUESTION 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.

$$(\mathrm{a}+\mathrm{ib})*(c+id)$$
$$=ac+adi+cbi+bd(i)^2$$
$$=ac+adi+cbi-bd$$
$$=ac+(ad+cb)*i-bd$$
$$=ac+[(a+b)*(c+d)-ac-bd]*i-bd$$

Thus we can get that there are total 3 real number multiplications

(1) $a*c$
(2) $b*d$
(3) $(a+b)*(c+d)$

(b) Find $(\mathrm{a}+\mathrm{ib})^2$ using only two multiplications of real numbers.

$$(\mathrm{a}+\mathrm{ib})*(\mathrm{a}+\mathrm{ib})$$
$$=\mathrm{a}^2+ab*i+ab*i+b^2*(i)^2$$
$$=\mathrm{a}^2+2ab*i+b^2*\left(\sqrt{-1}\right)^2$$
$$=\mathrm{a}^2+2ab*i-b^2$$
$$=(\mathrm{a}+\mathrm{b})(\mathrm{a}-\mathrm{b})+(ab+ab)*i$$

Then we can get that there are total two multiplications of real numbers

(1) $(\mathrm{a}+\mathrm{b})(\mathrm{a}-\mathrm{b})$
(2) $a*b$

(c) Find the product $(\mathrm{a}+\mathrm{ib})^2(c+id)^2$ using only five real number multiplications.

From the previous two questions, part(a) and part(b) we can get that to calculate $(\mathrm{a}+\mathrm{ib})*(c+id)$ we need three number multiplications and calculate the square, $(\mathrm{a}+\mathrm{ib})^2$ we need two multiplications of real numbers, thus we can say that $(\mathrm{a}+\mathrm{ib})^2(c+id)^2=[(a+ib)*(c+id)]^2$, then we cn say to calculate this we only need (2+3=5) real number multiplications.

(Q3 -> next page)

QUESTION 3

(a) 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.

- Let's say that we have two polynomials , P_A, P_B
- The degree of polynomial, P_A and P_B are both n, or we can say they're both n-degree polynomials
- When we multiply these two polynomial together, $P_A * P_B$, the degree of it would be (n + n) = 2n, so the number of values that we need to evaluate/determine it is, (2n + 1) values.
- We then use FFT to evaluate P_A, P_B at the values, the roots of unity of order which is the smallest power of 2 no less than (2n + 1), then we get the value form, which would take time O(n log n). (as its FFT, we don't do the matrix-vector multiplication anymore.)
- After that we multiply P_A, P_B with the value of points to get the value form $P_A * P_B$, then use the IFFT to get the coefficient form for it which takes O(n log n).

(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 P1, . . . , PK so that

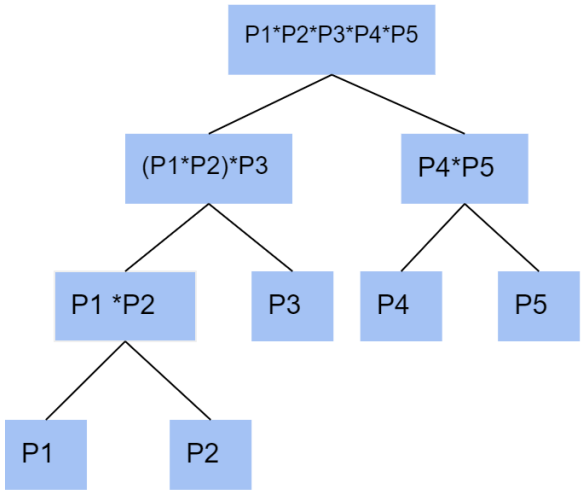
$$\text{degree}(P_1) + \cdots + \text{degree}(P_k) = S$$

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

- As we need to calculate ($P_1 * P_2 * P_3 * \ldots * P_k$), thus let G(i) equals to the final result of the multiplication of the K polynomials. Then we can tell that:
  $G(1) = P_1(x)$
  $G(2) = P_1(x) * P_2(x)$
  $G(3) = G(2) * P_3(x)$
  …
- As there are total (k-1) times multiplication of polynomials, and each time the degree would be smaller than S till the last one (equals to S), and for each polynomial, we need to evaluate it in (S+1) many points. Because of the degree of every step would be smaller or equals to S, and we are using FFT, thus we can get $O(S * \log(S))$ for each polynomial.
- As there are (k-1) such multiplications we get (k-1) * (S*log(S)) times to get the value form by using FFT and one more (S*log(S)) to get the coefficient form by using IFFT.
- Therefore, we can say that the whole process would cost O(K*S*log(s)).

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

- Idea: we can use divide and conquer algorithm, first multiply the polynomials in pairs of two, then we keep doing it till it becomes one polynomial.
- Thus we can possibly use perfect binary tree, assign all the polynomials that need to be multiple together, as the leaf of the perfect binary tree, if K $\neq 2^n$, (n > 0), then we can move the left most two leaves as the last level to form a $log_2(k) + 1$ level binary tree, like the graph below:



- For multiply each pair we can say that they have degree of x and y respectively, so the time complexity of multiply them together using FFT would be $(x + y) * \log(x + y)$, then we can show that $O((x + y)\log(x + y))$ as $O((x + y)\log(S))$, because of $(x + y)$ or $\log(x + y)$ is always $\leq S$, then for all product on level k (multiplication of polynomials) we need O(S*log(S)). Because of there are total $\lfloor log_2(k) \rfloor$ levels, thus we need this many multiplications (which is the depth of the perfect binary tree), or when k is odd its possibly $\lfloor log_2(k) \rfloor + 1$. Therefore we can say that it is $O(\log(k) * S * \log(S))$.

(Q4 -> next page)

QUESTION 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 … for all integers $n \geq 1$

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

1. By induction, first of all we show that when $n = 1$:

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

By the definition of Fibonacci numbers we know that $F_0 = 0, F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$

Then we can get $F_2 = F_1 + F_0 = 0 + 1 = 1$.

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

Therefore, it works for $n = 1$.

2. We let $n = k$:

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

3. We then let $n = k + 1$:

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

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

By the definition of Fibonacci numbers we know that $F_0 = 0, F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$

Then we know that $F_{k+1} + F_k = F_{k+2}$ and $F_k + F_{k-1} = F_{k+1}$

Thus we can get the result $\begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix}$ which equals to $\begin{pmatrix} F_{k+1+1} & F_{k+1} \\ F_{k+1} & F_{k-1+1} \end{pmatrix}$

Therefore it works for $n = k$ and $n = k + 1$.

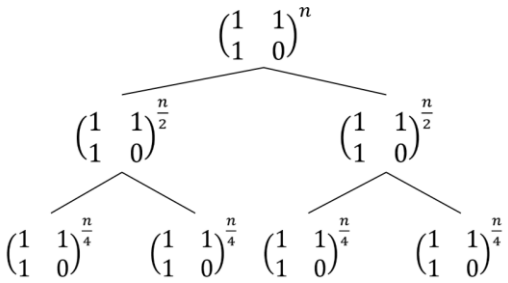
(b) Hence or otherwise, give an algorithm that find F_n in $O(\log(n))$

- For this question we can understand it as finding $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ in $O(\log(n))$

We can say that $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{2}} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{2}} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{4}} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{4}} * \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{4}} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{\frac{n}{4}} = \cdots$

If we view it as a balanced binary tree, we can get a binary tree with depth $\lfloor log_2(n) \rfloor$

We then can form a balanced binary tree like the graph below (next page) :



- we can think about this, $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n} = \left(\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n}{2}}\right)^{2} = \left(\left(\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n}{4}}\right)^{2}\right)^{2}$ …, once when the most inside exponent become like this $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{1}$, we start to go back and calculating (square) it, like the following equation $\left(\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{1}\right)^{2} = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{2}$ and $\left(\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{2}\right)^{2} = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{4}$ and … $\left(\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n}{2}}\right)^{2} = \begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{n}$. However, what if n is an odd number. We do $\begin{pmatrix}1 & 1\\ 1 & 0\end{pmatrix}^{\frac{n-1}{2}}$ when the exponent/power/n becomes odd and keep calculate it.
- If we related to the binary tree that we mentioned previously, each level we only need to do one ‘square’ calculation, and there are total $log_2(n)$ levels, that’s we need to do this many calculation do get the final result.
- Therefore, we can get F_n in $O(\log(n))$ time.

(Q5 -> next page)

QUESTION 5

(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.

- N -> number of giants
  L -> number of leaders
  K -> the min number of giants between every pair/two leaders
  T -> min leader height (from question (a))
  H[1 … N] -> heights of giants in the order that they stand in the line
- To solve the 'decision' version of this problem:
- Search/checking the giants height, (H[1 … N]), from left to right
  We need a local variable, count = 0, to count leader candidates
  If (the height of giant is greater or equals to T) then : we can adding up the count, and skip K giants.
  And keep searching/checking/repeat the previous step till reach the last value/height in the array, H.
- Then we can get the number of leader candidates, count, if the number is greater or equals to L (return True, otherwise false), then there exists some valid choice of leaders satisfying the constrains whose shortest leader has height no less than T.
- To search/checking every element in the heights array, we takes O(n) times.

(b) Hence, show that you can solve the optimization version of this problem in $O(N \log N)$ time.

- From the previous question, part(a), the algorithm checked if there one T as input, if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than T.
- The optimization version of this problem is to find the maximum height of the shortest leader among all valid choices of L leaders.
- Then we want to find the maximum possible T in this array.
- Get one copy of the array, one use to checking if there exists L valid leaders satisfying the constraints whose shortest leader has height of no less than T.
- We can do a merge sort the height, H, which takes $O(n\log(n))$ in increasing order, and view every element as candidate of T. Pick the middle one as T, put it into the algorithm that we got in previous question, if it returns true, means there is L leaders in the array with K gap, and they're all greater than T, thus every number/element in the left of middle T (the elements in the left are all smaller than T), all of them can return true in the algorithm. So we keep the middle term and go right, till some element returns false, then we goes back to left, keep put the current element, which is the current possible T, back to the 5(a) algorithm to check, till we find the largest T.
- To make this happen, we takes $O(n\log(n))$ to sort, and $O(\log(n)) * O(n)$ for the binary search and put the element back to the 5(a) algorithm to check if it returns true or false.
- Therefore, its $O(n\log(n))$ time.