COMP9801 Assignment 2

z5100764
Chunnan Sheng

Question 1
(a)

Calculate multiplication of the following two n-degree polynomials:

$$P_A(x) = A_0 + A_1 x + A_2 x^2 + \ldots + A_n x^n$$
$$P_B(x) = B_0 + B_1 x + B_2 x^2 + \ldots + B_n x^n$$

Step 1:

We select $2n+1$ points for each n-degree polynomial.
x coordinates of these points are complex roots of unity of order $2n+1$,
Then, these points on $P_A(x)$ are:

$$\left(\omega_{2n+1}^{0}, P_A(\omega_{2n+1}^{0})\right), \left(\omega_{2n+1}^{1}, P_A(\omega_{2n+1}^{1})\right), \ldots, \left(\omega_{2n+1}^{2n}, P_A(\omega_{2n+1}^{2n})\right)$$

While these points on $P_B(x)$ are:

$$\left(\omega_{2n+1}^{0}, P_B(\omega_{2n+1}^{0})\right), \left(\omega_{2n+1}^{1}, P_B(\omega_{2n+1}^{1})\right), \ldots, \left(\omega_{2n+1}^{2n}, P_B(\omega_{2n+1}^{2n})\right)$$

We can represent these points this way:

Polynomial A:

$$\begin{pmatrix} P_A(\omega_{2n+1}^{0}) \\ P_A(\omega_{2n+1}^{1}) \\ P_A(\omega_{2n+1}^{2}) \\ \ldots \\ P_A(\omega_{2n+1}^{2n}) \end{pmatrix} = \begin{pmatrix} \omega_{2n+1}^{0} & \omega_{2n+1}^{0} & \omega_{2n+1}^{0} & \ldots & \omega_{2n+1}^{0} & \omega_{2n+1}^{0} & \ldots & \omega_{2n+1}^{0} \\ \omega_{2n+1}^{0} & \omega_{2n+1}^{1} & \omega_{2n+1}^{2} & \ldots & \omega_{2n+1}^{n} & \omega_{2n+1}^{n+1} & \ldots & \omega_{2n+1}^{2n} \\ \omega_{2n+1}^{0} & \omega_{2n+1}^{2} & \omega_{2n+1}^{4} & \ldots & \omega_{2n+1}^{2n} & \omega_{2n+1}^{2(n+1)} & \ldots & \omega_{2n+1}^{4n} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \omega_{2n+1}^{0} & \omega_{2n+1}^{2n} & \omega_{2n+1}^{4n} & \ldots & \omega_{2n+1}^{2n^2} & \omega_{2n+1}^{2n(n+1)} & \ldots & \omega_{2n+1}^{4n^2} \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \ldots \\ A_n \\ 0 \\ \ldots \\ 0 \end{pmatrix}$$

Polynomial B:

$$\begin{pmatrix} P_B(\omega_{2n+1}^{0}) \\ P_B(\omega_{2n+1}^{1}) \\ P_B(\omega_{2n+1}^{2}) \\ \ldots \\ P_B(\omega_{2n+1}^{2n}) \end{pmatrix} = \begin{pmatrix} \omega_{2n+1}^{0} & \omega_{2n+1}^{0} & \omega_{2n+1}^{0} & \ldots & \omega_{2n+1}^{0} & \omega_{2n+1}^{0} & \ldots & \omega_{2n+1}^{0} \\ \omega_{2n+1}^{0} & \omega_{2n+1}^{1} & \omega_{2n+1}^{2} & \ldots & \omega_{2n+1}^{n} & \omega_{2n+1}^{n+1} & \ldots & \omega_{2n+1}^{2n} \\ \omega_{2n+1}^{0} & \omega_{2n+1}^{2} & \omega_{2n+1}^{4} & \ldots & \omega_{2n+1}^{2n} & \omega_{2n+1}^{2(n+1)} & \ldots & \omega_{2n+1}^{4n} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \omega_{2n+1}^{0} & \omega_{2n+1}^{2n} & \omega_{2n+1}^{4n} & \ldots & \omega_{2n+1}^{2n^2} & \omega_{2n+1}^{2n(n+1)} & \ldots & \omega_{2n+1}^{4n^2} \end{pmatrix} \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ \ldots \\ B_n \\ 0 \\ \ldots \\ 0 \end{pmatrix}$$

The time complexity to calculate

$$\begin{pmatrix} P_A(\omega_{2n+1}^0) \\ P_A(\omega_{2n+1}^1) \\ P_A(\omega_{2n+1}^2) \\ \dots \\ P_A(\omega_{2n+1}^{2n}) \end{pmatrix} \text{ and } \begin{pmatrix} P_B(\omega_{2n+1}^0) \\ P_B(\omega_{2n+1}^1) \\ P_B(\omega_{2n+1}^2) \\ \dots \\ P_B(\omega_{2n+1}^{2n}) \end{pmatrix}$$

is $O(2n\log(2n)) = O(n\log(n))$ using FFT algorithm.

Step 2:

Multiply $P_A(x)$ and $P_B(x)$ of each point, then we get $2n+1$ points of the new polynomial that is multiplication of P_A and P_B .

$$\begin{gathered} \left(\omega_{2n+1}^0, P_A(\omega_{2n+1}^0) P_B(\omega_{2n+1}^0)\right), \\ \left(\omega_{2n+1}^1, P_A(\omega_{2n+1}^1) P_B(\omega_{2n+1}^1)\right), \\ \left(\omega_{2n+1}^2, P_A(\omega_{2n+1}^2) P_B(\omega_{2n+1}^2)\right), \\ \dots, \\ \left(\omega_{2n+1}^{2n}, P_A(\omega_{2n+1}^{2n}) P_B(\omega_{2n+1}^{2n})\right) \end{gathered}$$

Time complexity of this step is $O(2n)=O(n)$

Step 3:

We should figure out all coefficients of the new polynomial according to the points got at step 2.

Assume $C_0, C_1, C_2, \dots, C_{2n}$ are coefficients of the new polynomial, then,

$$\begin{pmatrix} \omega_{2n+1}^0 & \omega_{2n+1}^0 & \omega_{2n+1}^0 & \dots & \omega_{2n+1}^0 \\ \omega_{2n+1}^0 & \omega_{2n+1}^1 & \omega_{2n+1}^2 & \dots & \omega_{2n+1}^{2n} \\ \omega_{2n+1}^0 & \omega_{2n+1}^2 & \omega_{2n+1}^4 & \dots & \omega_{2n+1}^{4n} \\ \dots & \dots & \dots & \dots & \dots \\ \omega_{2n+1}^0 & \omega_{2n+1}^{2n} & \omega_{2n+1}^{4n} & \dots & \omega_{2n+1}^{4n^2} \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \dots \\ C_{2n} \end{pmatrix} = \begin{pmatrix} P_A(\omega_{2n+1}^0) P_B(\omega_{2n+1}^0) \\ P_A(\omega_{2n+1}^1) P_B(\omega_{2n+1}^1) \\ P_A(\omega_{2n+1}^2) P_B(\omega_{2n+1}^2) \\ \dots \\ P_A(\omega_{2n+1}^{2n}) P_B(\omega_{2n+1}^{2n}) \end{pmatrix}$$

then,

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \dots \\ C_{2n} \end{pmatrix} = \frac{1}{2n+1} \begin{pmatrix} \omega_{2n+1}^0 & \omega_{2n+1}^0 & \omega_{2n+1}^0 & \dots & \omega_{2n+1}^0 \\ \omega_{2n+1}^0 & \omega_{2n+1}^{-1} & \omega_{2n+1}^{-2} & \dots & \omega_{2n+1}^{-2n} \\ \omega_{2n+1}^0 & \omega_{2n+1}^{-2} & \omega_{2n+1}^{-4} & \dots & \omega_{2n+1}^{-4n} \\ \dots & \dots & \dots & \dots & \dots \\ \omega_{2n+1}^0 & \omega_{2n+1}^{-2n} & \omega_{2n+1}^{-4n} & \dots & \omega_{2n+1}^{-4n^2} \end{pmatrix} \begin{pmatrix} P_A(\omega_{2n+1}^0) P_B(\omega_{2n+1}^0) \\ P_A(\omega_{2n+1}^1) P_B(\omega_{2n+1}^1) \\ P_A(\omega_{2n+1}^2) P_B(\omega_{2n+1}^2) \\ \dots \\ P_A(\omega_{2n+1}^{2n}) P_B(\omega_{2n+1}^{2n}) \end{pmatrix}$$

We can use the same FFT algorithm where ω_{2n+1}^k is replaced by ω_{2n+1}^{-k} , $k \in \{0,1,2,\dots,2n\}$. Therefore time complexity of step 3 is also $O(2n\log(2n)) = O(n\log(n))$.

As a conclusion, multiplication of two n-degree polynomials costs $O(n\log(n)) + O(n) + O(n\log(n)) = O(n\log(n))$ time.

Question 1
(b) (i)

Step 1:

We select $S+1$ points for each polynomial.
We assume x coordinates of these points are complex roots of unity of order $S+1$,
Then, these points on polynomial $P_i(x), i\in\{1,2,\dots,K\}$ are:
$(\omega_{S+1}^0, P_i(\omega_{S+1}^0)), (\omega_{S+1}^1, P_i(\omega_{S+1}^1)), \dots, (\omega_{S+1}^S, P_i(\omega_{S+1}^S))$

Then, values of these points on polynomial $P_i(x), i\in\{1,2,\dots,K\}$ will be calculated this way:

$$\begin{pmatrix} P_i(\omega_{S+1}^0) \\ P_i(\omega_{S+1}^1) \\ P_i(\omega_{S+1}^2) \\ \dots \\ P_i(\omega_{S+1}^S) \end{pmatrix} = \begin{pmatrix} \omega_{S+1}^0 & \omega_{S+1}^0 & \omega_{S+1}^0 & \dots & \omega_{S+1}^0 & \omega_{S+1}^0 & \dots & \omega_{S+1}^0 \\ \omega_{S+1}^0 & \omega_{S+1}^1 & \omega_{S+1}^2 & \dots & \omega_{S+1}^{degree(P_i)} & \omega_{S+1}^{degree(P_i)+1} & \dots & \omega_{S+1}^S \\ \omega_{S+1}^0 & \omega_{S+1}^2 & \omega_{S+1}^4 & \dots & \omega_{S+1}^{2\cdot degree(P_i)} & \omega_{S+1}^{2(degree(P_i)+1)} & \dots & \omega_{S+1}^{2S} \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \omega_{S+1}^0 & \omega_{S+1}^S & \omega_{S+1}^{2S} & \dots & \omega_{S+1}^{S\cdot degree(P_i)} & \omega_{S+1}^{S(degree(P_i)+1)} & \dots & \omega_{S+1}^{S^2} \end{pmatrix} \begin{pmatrix} I_0 \\ I_1 \\ I_2 \\ \dots \\ I_{degree(P_i)} \\ 0 \\ \dots \\ 0 \end{pmatrix}$$

Time complexity of this calculation using FFT algorithm is $O(S\log(S))$.
Since there are K such polynomials, the entire time complexity of step 1 is $O(K\,S\log(S))$.

Step 2:

We should do multiplications and get $S+1$ points of the new polynomial.
These points can be seen as follows:

$$\left(\omega_{S+1}^0, \prod_{i=1}^{K} P_i(\omega_{S+1}^0)\right),$$
$$\left(\omega_{S+1}^1, \prod_{i=1}^{K} P_i(\omega_{S+1}^1)\right),$$
$$\left(\omega_{S+1}^2, \prod_{i=1}^{K} P_i(\omega_{S+1}^2)\right),$$
$$\dots,$$
$$\left(\omega_{S+1}^S, \prod_{i=1}^{K} P_i(\omega_{S+1}^S)\right)$$

Time complexity of this calculation is $O(K\cdot S)$

Step 3:

We should figure out all coefficients of the new polynomial according to those points gained at step 2.

Then, what we need to do is similar to step 3 of question (a).
Provided that $C_{0,} C_{1,} C_{2,} \dots, C_S$ are coefficients of the new polynomial, then,

$$\begin{pmatrix} \omega_{S+1}^0 & \omega_{S+1}^0 & \omega_{S+1}^0 & \cdots & \omega_{S+1}^0 \\ \omega_{S+1}^0 & \omega_{S+1}^1 & \omega_{S+1}^2 & \cdots & \omega_{S+1}^S \\ \omega_{S+1}^0 & \omega_{S+1}^2 & \omega_{S+1}^4 & \cdots & \omega_{S+1}^{2S} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \omega_{S+1}^0 & \omega_{S+1}^S & \omega_{S+1}^{2S} & \cdots & \omega_{S+1}^{S^2} \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \cdots \\ C_S \end{pmatrix} = \begin{pmatrix} \prod_{i=1}^{K} P_i(\omega_{S+1}^0) \\ \prod_{i=1}^{K} P_i(\omega_{S+1}^1) \\ \prod_{i=1}^{K} P_i(\omega_{S+1}^2) \\ \cdots \\ \prod_{i=1}^{K} P_i(\omega_{S+1}^S) \end{pmatrix}$$

Then,

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ \cdots \\ C_S \end{pmatrix} = \frac{1}{S+1} \begin{pmatrix} \omega_{S+1}^0 & \omega_{S+1}^0 & \omega_{S+1}^0 & \cdots & \omega_{S+1}^0 \\ \omega_{S+1}^0 & \omega_{S+1}^{-1} & \omega_{S+1}^{-2} & \cdots & \omega_{S+1}^{-S} \\ \omega_{S+1}^0 & \omega_{S+1}^{-2} & \omega_{S+1}^{-4} & \cdots & \omega_{S+1}^{-2S} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \omega_{S+1}^0 & \omega_{S+1}^{-S} & \omega_{S+1}^{-2S} & \cdots & \omega_{S+1}^{-S^2} \end{pmatrix} \begin{pmatrix} \prod_{i=1}^{K} P_i(\omega_{S+1}^0) \\ \prod_{i=1}^{K} P_i(\omega_{S+1}^1) \\ \prod_{i=1}^{K} P_i(\omega_{S+1}^2) \\ \cdots \\ \prod_{i=1}^{K} P_i(\omega_{S+1}^S) \end{pmatrix}$$

We can use the same FFT algorithm where ω_{S+1}^m is replaced by ω_{S+1}^{-m} , $m \in \{0,1,2,\ldots,S\}$.
Therefore, time complexity of step 3 is $O(S\log(S))$.

As a conclusion, multiplication of K polynomials costs
$O(KS\log(S)) + O(KS) + O(S\log(S)) = O(KS\log(S))$ time.

Question 1
(b) (ii)

If we multiply these polynomial in pairs, number of multiplications is $M_1 = \left\lfloor \frac{K}{2} \right\rfloor$. We can continue doing this step by step until all polynomials merge into a single polynomial. If we assume multiplications needed at step i is M_i, then multiplications of the next step is $M_{i+1} = \left\lceil \frac{M_i}{2} \right\rceil$. So total number of these steps will be $O(\log(K))$ because M will decrease exponentially to 1 within $\log(K)$ time.

Secondly, if we suppose degree of each polynomial is S_j^i at step i, then $S = \sum_{j=1}^{N} S_j^i$ where N is number of polynomials at this step. Time to multiply each pair of polynomial is $(S_j^i + S_{j+1}^i)\log(S_j^i + S_{j+1}^i)$, then, time spent on this step is: $\sum_{j=1}^{N/2} \left[(S_{2j-1}^i + S_{2j}^i)\log(S_{2j-1}^i + S_{2j}^i) \right]$.

Thirdly, $(a+b)^{a+b} = a^{a+b} + \ldots + \binom{a+b}{b} a^a b^b + \ldots + b^{a+b} > \binom{a+b}{b} a^a b^b > a^a b^b$ where $a \in \mathbb{N}, b \in \mathbb{N}$, which means $(a+b)\log(a+b) > a\log a + b\log b$.

Then, we can prove that $\left(\sum_{i=1}^{n} a_i\right)\log\left(\sum_{i=1}^{n} a_i\right) > \sum_{i=1}^{n} (a_i \log a_i)$ where $a_i \in \mathbb{N}, n \in \mathbb{N}, i \in \mathbb{N}$.

Thus, $S\log(S) = \left(\sum_{j=1}^{N} S_j^i\right)\log\left(\sum_{j=1}^{N} S_j^i\right) > \sum_{j=1}^{N/2} \left[(S_{2j-1}^i + S_{2j}^i)\log(S_{2j-1}^i + S_{2j}^i) \right]$. It means time spent at each step is less than $S\log(S)$. Therefore time complexity of the entire algorithm is $O(S\log(S)\log(K))$.

Question 2

Step 1:

We suppose coin values are $v_{1,} v_{2,} \ldots, v_N$, then we should count number of coins of the same value together before we deal with this problem via polynomials. For example, if there are only one coin of value 3, coefficient of x^3 will be 1; if there are 5 coins of value 4, coefficient of x^4 will be 5.

Thus, we can get a new list of unique coin values from 0 to M and their corresponding coefficients. We know that degree of this polynomial is M , and its coefficients are $A_{0,} A_{1,} A_{2,} \ldots, A_M$. There may be zero-value coefficients, which means coins of certain values may not exist.

Time complexity of step 1 is $O(N)$

Step 2:

We try to figure out polynomial values of complex roots of unity of order $2M+1$:

$$\begin{pmatrix} P_A(\omega_{2M+1}^0) \\ P_A(\omega_{2M+1}^1) \\ P_A(\omega_{2M+1}^2) \\ \ldots \\ P_A(\omega_{2M+1}^{2M}) \end{pmatrix} = \begin{pmatrix} \omega_{2M+1}^0 & \omega_{2M+1}^0 & \omega_{2M+1}^0 & \ldots & \omega_{2M+1}^0 & \omega_{2M+1}^0 & \ldots & \omega_{2M+1}^0 \\ \omega_{2M+1}^0 & \omega_{2M+1}^1 & \omega_{2M+1}^2 & \ldots & \omega_{2M+1}^M & \omega_{2M+1}^{M+1} & \ldots & \omega_{2M+1}^{2M} \\ \omega_{2M+1}^0 & \omega_{2M+1}^2 & \omega_{2M+1}^4 & \ldots & \omega_{2M+1}^{2M} & \omega_{2M+1}^{2(M+1)} & \ldots & \omega_{2M+1}^{4M} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \omega_{2M+1}^0 & \omega_{2M+1}^{2M} & \omega_{2M+1}^{4M} & \ldots & \omega_{2M+1}^{2M^2} & \omega_{2M+1}^{2M(M+1)} & \ldots & \omega_{2M+1}^{4M^2} \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \ldots \\ A_M \\ 0 \\ \ldots \\ 0 \end{pmatrix}$$

$O(2M \log(2M)) = O(M \log(M))$ time is needed to calculate $\begin{pmatrix} P_A(\omega_{2M+1}^0) \\ P_A(\omega_{2M+1}^1) \\ P_A(\omega_{2M+1}^2) \\ \ldots \\ P_A(\omega_{2M+1}^{2M}) \end{pmatrix}$ via FFT algorithm.

Step 3:

We multiply this polynomial with itself , and get a new polynomial $P_B(x)$ whose coefficients are $B_{0,} B_{1,} B_{2,} \ldots, B_{2M}$:

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ \ldots \\ B_{2M} \end{pmatrix} = \begin{pmatrix} \omega_{2M+1}^0 & \omega_{2M+1}^0 & \omega_{2M+1}^0 & \ldots & \omega_{2M+1}^0 \\ \omega_{2M+1}^0 & \omega_{2M+1}^1 & \omega_{2M+1}^2 & \ldots & \omega_{2M+1}^{2M} \\ \omega_{2M+1}^0 & \omega_{2M+1}^2 & \omega_{2M+1}^4 & \ldots & \omega_{2M+1}^{4M} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \omega_{2M+1}^0 & \omega_{2M+1}^{2M} & \omega_{2M+1}^{4M} & \ldots & \omega_{2M+1}^{4M^2} \end{pmatrix}^{-1} \begin{pmatrix} [P_A(\omega_{2M+1}^0)]^2 \\ [P_A(\omega_{2M+1}^1)]^2 \\ [P_A(\omega_{2M+1}^2)]^2 \\ \ldots \\ [P_A(\omega_{2M+1}^{2M})]^2 \end{pmatrix}$$

$$= \frac{1}{2M+1}\begin{pmatrix} \omega_{2M+1}^{0} & \omega_{2M+1}^{0} & \omega_{2M+1}^{0} & \cdots & \omega_{2M+1}^{0} \\ \omega_{2M+1}^{0} & \omega_{2M+1}^{-1} & \omega_{2M+1}^{-2} & \cdots & \omega_{2M+1}^{-2M} \\ \omega_{2M+1}^{0} & \omega_{2M+1}^{-2} & \omega_{2M+1}^{-4} & \cdots & \omega_{2M+1}^{-4M} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \omega_{2M+1}^{0} & \omega_{2M+1}^{-2M} & \omega_{2M+1}^{-4M} & \cdots & \omega_{2M+1}^{-4M^2} \end{pmatrix}\begin{pmatrix} [P_A(\omega_{2M+1}^{0})]^2 \\ [P_A(\omega_{2M+1}^{1})]^2 \\ [P_A(\omega_{2M+1}^{2})]^2 \\ \cdots \\ [P_A(\omega_{2M+1}^{2M})]^2 \end{pmatrix}$$

Time complexity of this step is $O(2M\log(2M))=O(M\log(M))$ via FFT algorithm.

Step 4:

Iterate the array of coefficients $B_{0,} B_{1,} B_{2,} \ldots, B_{2M}$ and delete all items whose values are less than 2. Then among the remaining coefficients, corresponding **exponents** (they are also indexes of this array) of variable x are all the possible sums of 2 coins.

For example, if $P_B(x)=x^2+2x^5+2x^6+x^8+2x^9+x^{10}$, possible sums of two coins are 5, 6 and 9. Only linear time is needed to complete this step.

As a conclusion, the entire algorithm needs $O(M\log(M))$ time.

Question 3 (a)

If we assume that $F_{n-1}=a$ and $F_n=b$ where $n\in\mathbb{N}$, then $F_{n+1}=F_n+F_{n-1}=a+b$, thus $\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} a+b & b \\ b & a \end{pmatrix}$.

Since $F_{n+2}=F_{n+1}+F_n=a+2b$, we can say $\begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} = \begin{pmatrix} a+2b & a+b \\ a+b & b \end{pmatrix}$.

Provided that

$$\begin{pmatrix} a+2b & a+b \\ a+b & b \end{pmatrix} = \begin{pmatrix} a+b & b \\ b & a \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} ,$$

we can figure out that

$$\begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} .$$

Given that $\begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, It can be proved that

$$\begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \quad \text{where } n\in\mathbb{N} .$$

Question 3 (b)


```
// Multiplication of n matrices.
// Each matrix == [[1, 1],[1, 0]]
// This is a recursive function and
// its time complexity is O(log(n))
Matrix Fibonacci_power(int n)
{
    // This is the start point of recursion
    if (n == 1)
    {
        return[[1, 1], [1, 0]];
    }
    // Recursively call Fibonacci_power
    Matrix child = Fibonacci_power(n / 2);

    if (n % 2 == 0)
    {
        return child * child;
    }
    else
    {
        return child * child * [[1, 1], [1, 0]];
    }
}

// Main entrance of this algorithm
int Fibonacci(int n)
{
    if (n >= 0)
    {
        // Multiply n + 1 matrices
        // Time complexity is O(log(n)) here.
        Matrix result = Fibonacci_power(n + 1);

        // Return the element at the last row and the last column
        return result[1][1];
    }
    else
    {
        throw ERROR;
    }
}
```

Question 4

```
// C++ source code
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>

// This struct stores:
// 1. index of the item;
// 2. difference (no less than zero) of the prices between A and B;
// 3. ID of the person who pays more than the other person;
struct Item
{
    int index;
    int price_diff;
    char who_pays_more;

    Item() : index{ 0 }, price_diff{ 0 }, who_pays_more{ 'A' } {}

    Item(int i, int diff, char who)
        : index{ i }, price_diff{ diff }, who_pays_more{ who }
    {}
};

// Compare two items via price differences
bool diff_more(const Item & item1, const Item & item2)
{
    if (item1.price_diff > item2.price_diff)
    {
        return true;
    }

    return false;
}

// Main entrance of this algorithm.
// Time complexity is O(N * log(N))
std::map<int, char> maximize(int N, int * a, int * b, int A_limit, int B_limit)
{
    // Stores all the items containing price differences and
    // information that which person pays more than the other person
    std::vector<Item> items;

    // Iterate all the items
    for (int i = 0; i < N; ++i)
    {
        if (a[i] >= b[i])
        {
            // Insert the item with price difference and label it that
            // A pays more than B for this item.
            items.push_back(Item{ i, a[i] - b[i], 'A' });
        }
        else
        {
            // Insert the item with price difference and label it that
            // B pays more than A for this item.
            items.push_back(Item{ i, b[i] - a[i], 'B' });
        }
    }

    // Sort all the items in non-increase order of price differences,
    // time complexity is O(N * log (N)).
    std::sort(items.begin(), items.end(), diff_more);

    int a_items = 0;
    int b_items = 0;

    std::map<int, char> result;

    // Iterate all the items and decide whether to sell this item to A or B.
    // Time complexity of this "for" loop is O(N).
    // If you want elements in the result to be sorted, it would be O(N * log(N))
```

```
    for (Item & item : items)
    {
        if (item.who_pays_more == 'A')
        {
            if (a_items < A_limit)
            {
                // Let A buy this item
                result.insert({ item.index, 'A' });
                ++a_items;
            }
            else
            {
                // A's shopping cart is full even though A bids higher.
                // So we have to sell this item to B.
                result.insert({ item.index, 'B' });
                ++b_items;
            }
        }
        else
        {
            if (b_items < B_limit)
            {
                // Let B buy this item
                result.insert({ item.index, 'B' });
                ++b_items;
            }
            else
            {
                // B's shopping cart is full even though B bids higher.
                // So we have to sell this item to A.
                result.insert({ item.index, 'A' });
                ++a_items;
            }
        }
    }

    return result;
}


int main()
{
    int N = 6; int A = 3; int B = 4;

    int a[6] = { 5, 7, 2, 4, 8, 3 };
    int b[6] = { 3, 2, 9, 2, 2, 4 };

    auto ret = maximize(N, a, b, A, B);

    for (auto & pair : ret)
    {
        std::cout << "( " << pair.first << "\t" << pair.second << " )\n";
    }

    return 0;
}
```

Question 5 (a)

```cpp
// C++ source code
// "The shortest leader's height should be no less than T" is exactly
// the same as "all leaders' height should be no less than T".
#include <iostream>
#include <vector>

// Find leaders from array of giants using greedy method.
// Time complexity of this function is O(N)
std::vector<size_t> find_leaders(const std::vector<int> & giants, int K, int T)
{
    // Container of leaders
    std::vector<size_t> leaders;

    size_t i = 0;

    while (i < giants.size())
    {
        // Iterate each giant in the array
        // Stop the iteration if there is a giant whose height is not less than T
        for (; i < giants.size() && giants[i] < T; ++i);

        // If the index exceeds range of the array, stop the while loop
        if (i >= giants.size())
        {
            break;
        }

        // Push the giant whose height is not less than T into the container of leaders
        leaders.push_back(i);
        // Jump the index so that there are at least K giants between 2 leaders
        i += K + 1;
    }

    return leaders;
}

// This function simply judge if we can find at least L leaders whose
// height are at least T
bool can_find_leaders(const std::vector<int> & giants, int L, int K, int T)
{
    std::vector<size_t> ret = find_leaders(giants, K, T);

    // If we can find at least L leaders from the array of giants
    if (ret.size() >= L)
    {
        return true;
    }

    // If we cannot find at least L leaders from the array of giants
    return false;
}
```

Question 5 (b)

```
// This function simply judge if we can find at least L leaders whose height are at least T
// Slightly modify this function in question (a).
// A new argument "ret" can help us get all the selected leaders
bool can_find_leaders(const std::vector<int> & giants, int L, int K, int T,
std::vector<size_t> & ret)
{
    ret = find_leaders(giants, K, T);

    // If we can find at least L leaders from the array of giants
    if (ret.size() >= L)
    {
        return true;
    }

    // If we cannot find at least L leaders from the array of giants
    return false;
}


// Use "Divide and Conquer" method to solve the optimisation version of this problem
// Time complexity: O(N * log(N))
std::vector<size_t> find_leaders_shortest_max(const std::vector<int> & giants, int L, int K)
{
    std::vector<size_t> ret;

    // Copy the array of giants and sort them in non-decrease order of their heights
    // Time complexity of sort is N * log(N)
    std::vector<int> giants_copy = giants;
    std::sort(giants_copy.begin(), giants_copy.end());

    // tail is initialized as index of the last giant
    size_t tail = giants_copy.size() - 1;

    // head is index of the first giant
    size_t head = 0;

    // Figure out the answer using divide and conquer
    // Time complexity of this loop is N * log(N)
    while (tail - head > 1)
    {
        size_t mid = (head + tail) / 2;

        // Time complexity of this step is O(N)
        if (!can_find_leaders(giants, L, K, giants_copy[mid], ret))
        {
            tail = mid;
        }
        else
        {
            head = mid;
        }
    }

    // Check if the tail is the answer.
    if (can_find_leaders(giants, L, K, giants_copy[tail], ret))
    {
        return ret;
    }
    else // If the tail is too large, it is certain that
        // the answer is the head
    {
        return find_leaders(giants, K, giants_copy[head]);
    }
}
```