COMP3121 Assignment 2

Ziyi Cui z5097491

1(a).

1. Padding the 2 polynomials with 0 to the nearest power of 2. Assume degree 2n is the power of 2 after padding for the 2 polynomials.
2. Reduce the degree of the 2 polynomials by splitting the terms with even and odd powers to 2 groups replace x^2 with y. And do this step recursively until the polynomial cannot be split by this rule.
3. Substitute (2n + 1)/2 values for each polynomials to get 2n+1 P(w_n^k) unique values. We can use (2n + 1)/2 values to get 2n + 1 values because we can use cancelation lemma to simplify the evaluation part. Therefore, we can finish the evaluation in O(nlogn).
4. After we get 2n + 1 unique values for each polynomial, we finished the FFT part.
5. The we can use the 2n + 1 values for each polynomial to do the matrix inverse calculation to get the 2n+1 coefficients for the new polynomial.
6. We finished the polynomial multiplication.

1(b)(i).

To find the product of these K polynomials we need to find S + 1 unique values for each polynomials.
We can just follow the steps what we did in the part 1(a) with K polynomials not 2 and the number of unique values is S + 1 not 2n+1,
Then for each polynomial the time cost is O(SlognS).
Therefore, the time complexity is O(KSlogS) because we have K polynomials.

1(b)(ii).

We pairs the polynomials with the smallest and second smallest degrees, and do a polynomials multiplication with FFT to get a new polynomial with the sum of the degrees of the 2 old polynomials, right now we have K-1 polynomials.
And Then, we repeat the operation above until we get the final polynomial with degree S. It cost our K-1 many times operation.

Assume S is power of 2, and each polynomial has the same degrees.
We get the final polynomial by using the method from bottom-to-top,
If we look at it from the top-to-bottom, then we can represent the operations in a tree structure.

Degree=S

Degree=S/2 Degree=S/2

Deg=S/4 Deg=S/4 Deg=S/4 Deg=S/4

……………………………………………………………………

……………………………………………………………………

From the top to bottom, we reduce the size of the polynomial by half each time, and the depth of the tree is logK, and time complexity for each level is O(SlogS).
We can get the function: $T(S) = 2T\left(\frac{S}{2}\right) + S\log(S)$.

Therefore, we can get the time complexity of this solution is O(Slog(S)log(K)).
Even through the degree of each polynomial is not the same, we still can do this in the time of O(Slog(S)log(K)) in the worst case.

2.

The values between [1..M], so that the sum of 2 values is between [2..2M].
In this case we can represents each value to $v_1, \dots, v_n$.
And then we represents those values by polynomial:
P(x) = $x^{v_1} + \cdots + x^{v_n}$
We know the largest v_i is M, therefore, If we multiply P(x) and P(x) to get a new polynomial Q(x) with degree 2M. The powers range of Q(x) is [2..2M].
Right now we can use FFT to calculate Q(x) in the time O(MlogM).
However, we will get extra powers in Q(x) because all the terms in P(x) will multiply itself exactly once, and we do not want to count those extra powers into our all possible sums because we cannot add a coin with itself to get a sum.
Hence, we remove all the terms with coefficient 1 in Q(x).
Then powers of remaining terms are all the possible sums.

3(a).

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

$\begin{bmatrix} F_3 & F_2 \\ F_2 & F_1 \end{bmatrix}$ = $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2$ = $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ = $\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$

$\begin{bmatrix} F_4 & F_3 \\ F_3 & F_2 \end{bmatrix}$ = $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^3$ = $\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ = $\begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}$

……………………………………………

……………………………………………

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

Because $F_n + F_{n-1}$ = F_{n+1}

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

By Induction we can conclude that:

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

3(b).

Define M = $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$.

If we simply use the method in 3(a) to find the F_n by multiply M with itself n-1 times, which runs in time O(n).

What if we use divide-and-conquer method to do this question?

We can get :

$$M^n = M^{\frac{n}{2}} * M^{\frac{n}{2}}$$
$$M^{\frac{n}{2}} = M^{\frac{n}{4}} * M^{\frac{n}{4}}$$
$$\dots\dots\dots\dots\dots$$
$$M^4 = M^2 * M^2$$
$$M^2 = M * M$$

Right now we can get F_n in time O(logn).

If n is not power of 2, we can find the nearest power of two m that satisfy m < n firstly, and then use the results we calculated before we get M^m to fill the gap between M^m and M^n.

For Example:

n = 14

then m = 8

$M^2 = M * M$

$M^4 = M^2 * M^2$

$M^8 = M^4 * M^4$

$M^{14} = M^8 * M^4 * M^2$

So what if n = 15 ?

If n = 15 we can simply get it from M^{14} because we know M^n contains $F_{n-1}, F_n, F_{(n+1)}$, which means we can get $F_{15}\ from\ M^{14}$, also we can get F_{15} from M^{16}. Similarly, we can get F_{13} by calculate $M^{12} = M^8 * M^4$, which can use one less multiplication than M^{14}

Even the worst case, this algorithm can get F_n in the time O(logn).

4.

First of all make tuples of all items for A and B (item index, amount willing to paid by A/B), and then make 2 array to store the tuples for A(arr_A) and B(arr_B). This can be done in time O(n).

MergeSort the 2 arrays with the amount willing to paid in non-increasing order. (nlogn)

```
i = 0 , j = 0
Item_sold[N] = 0
boughtByA = 0
boughtByB = 0
# If one of the A and B bought enough items them we can sell items to another
# person
While(A> boughtByA and B > boughtByB and N > (boughtByA + boughtByB)):
        # sell the item at maximum price  to A or B
        If (arr_A[i][1] >= arr_B[j][1] ):
                # If the item already sold we check the next one
                # otherwise we sell it to A
                If(!item_sold[i]):
                        Sell the item[i] to A
                        item_sold[i] = 1
                        boughtByA += 1
                i += 1
        Else:
                If(!item_sold[j]):
                        Sell the item[j] to B
                        boughtByB += 1
                        item_sold[j] = 1
                j += 1
if (N == (bougthByA + boughtByB)):
        #Sold_out
        return;
# sell the items from highest to lowest price to another person until s/he cannot buy
# items anymore
if(A == boughtByA):
        while( B > boughtByB and N > (bougthByA + boughtByB)):
                If(!item_sold[j]):
                        Sell the item[j] to B
                        boughtByB += 1
                        item_sold[j] = 1
                j += 1
else:
        while( A > boughtByA and  N > (bougthByA + boughtByB)):
                If(!item_sold[i]):
                        Sell the item[i] to A
                        boughtByA += 1
                        item_sold[i] = 1
                i += 1
```

In this solution, we can always get the optimal solution because we always sell the item at highest price to A or B, If sell item i to A it must means the price A willing to pay to buy the item i is higher or equal to than B. Because the price A paid is higher than or equal to the highest price B can pay.

In terms of time complexity, T(n) = O(n) + O(nlogn) + O(n) = O(nlogn).

5(a).

Go through the H array from 1 to N,
Find the first people that H[i] >= T, and the find the next leader from index i + K + 1.
If we cannot find the enough of Leaders after we go over the array, it means no valid choice.
Otherwise, we can find valid choice in the array.

```
i = 0
num_leaders = 0
While(i < N):
        If (Num_leaders == L):
                Return True
        If (H[i] >= T):
                Num_leaders += 1
                i += k
        i += 1
return False
```

This solution runs in linear time.

5(b).

First we check whether or not exists some valid choice by using the solution in part(a) and simply change the T to 0.
If there are no valid solution then we done.

Otherwise, We append all the value in H to a new array A, and MergeSort Array A in a non-decreasing order.

Then, we pick the value at mid of Array A, which is A[N/2].

```
L = 0
R = N
While(L <= R)
=================    Loop Part      =================
mid = (L+R)/2
Assign A[mid] to T and run the function in 5(a) to check whether or not exists some
valid choice for T = A[mid]
If True:
        Possible_solution = T
        We only need to check right half of the array A, we need check whether there
        is a better solution than current one or not.
        L = mid + 1
Else:
        We only need to check left half of the array A, we need find one valid
        solution firstly.
        R = mid - 1
=================         End        =================
```

At the end of loop possible_solution is the final solution for the maximum height of the shortest leader among all valid choices of L leaders .

The time complexity is O(NlogN).

We always find the correct solution in this method because the algorithm always trying to find the highest value for T if the choice is valid.