1.

(a)

Multiplication of two n-degree polynomials takes O(n2) in coefficient representation.

There is the other representation for polynomials, which is point-value representation. A point-value representation of a polynomial A(x) of degree n is a set of n point-value pairs

{{(x0, y0), (x1, y1), …, (xn-1,yn-1)}}, all xk are distinct and yk = A(xk) for k = 0, 1, 2, 3, …, n – 1.

Multiplication of two n-degree polynomials in point-value representation takes O(n).

Summary:

The process of transforming from coefficient representation to point-value representation is called evaluation, which takes O(n2), the inverse process, which is called interpolation, takes O(n2) as well. The multiplication in point-value representation takes O(n).

By using Fast Fourier Transform and its inverse process, we can complete the evaluation and interpolation process described above both in O(nlogn).

Therefore we can multiply two n-degree polynomials together in O(nlogn).

(b)

(i)

the result of these K polynomials is in degree of S, therefore we need to S + 1 values to uniquely determine it. So for every polynomials P1… Pk, we need to evaluate Pi at the roots of unity of order S + 1 by using fast fourier transform. This takes O(S logS), there are K polynomials, so the total time complexity is KS logS.

(ii)

We use divide and conquer to divide the K polynomials into two sub-group

2.

square the polynomial xv1 +···+xvN

we can get that the possible sums are the exponent whose coefficient is two.

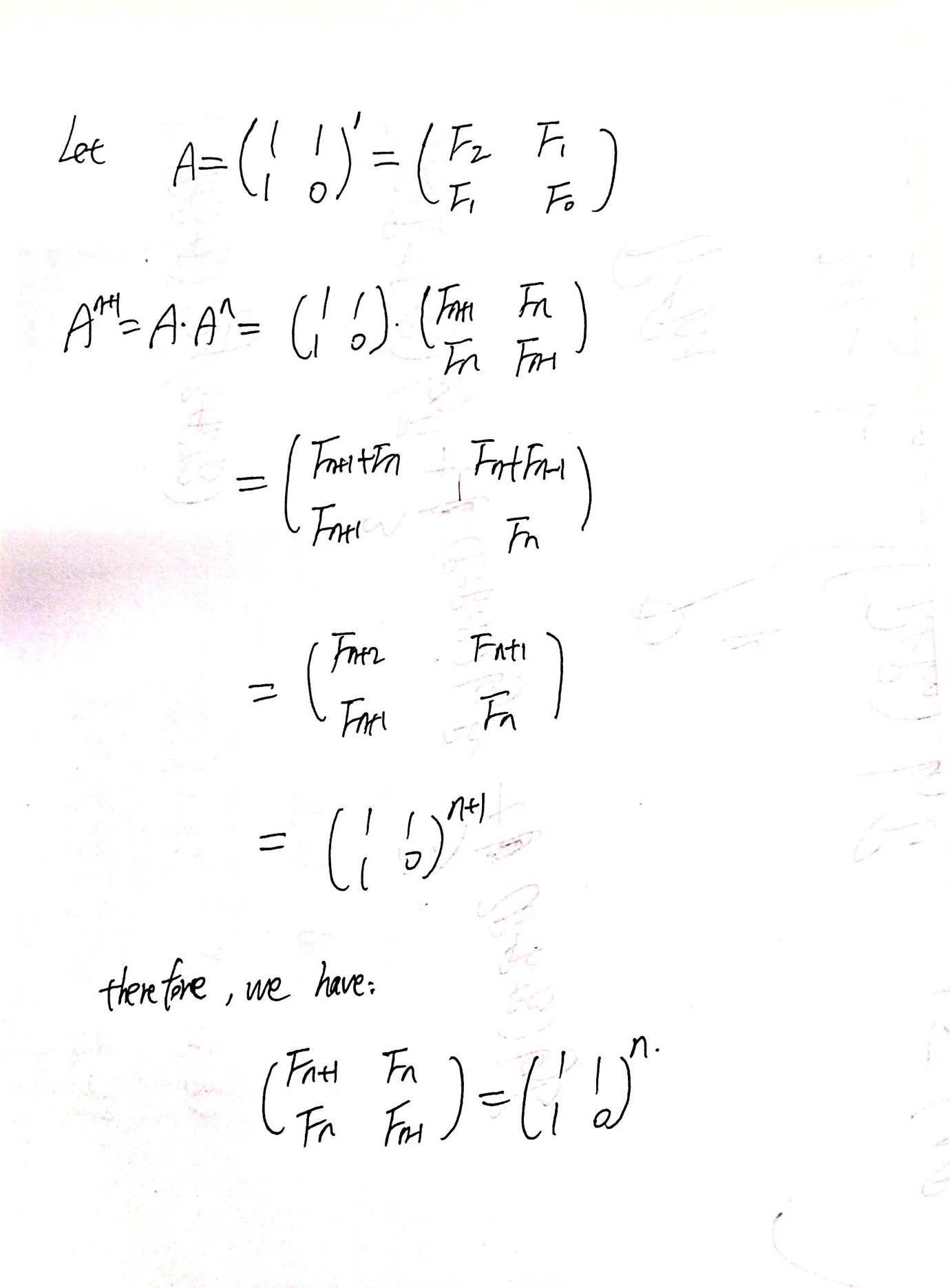
For example

(x1 + x4 + x5)\*(x1 + x4 + x5) = x2 + 2x5 + 2x6 + x8 + 2x9 + x10 . So the possible sums are 5, 6, 9, the coefficients for red part is 2, and the exponent is what we want, they are 5, 6, 9.

By using this idea, we can use FFT to calculate convolution of one sequence and itself. The degree is M, so the time complexity is O(M logM).

3.

(a)



(b).

From 3.a, we can get Fn by multiplying the matrix {{1, 1}, {1, 0}} n times, it’s like we are calculate the power(x, n), where x is the matrix {{1, 1}, {1, 0}}. We can use recursion and divide and conquer to complete it in time O(logn).

Power(x, n)

{

if (n == 0)

return x;

else if(n % 2 == 0)

return Power(x, n / 2) \* Power(x, n / 2);

else

return x \* Power(x, n / 2) \* Power(x, n / 2);

}

when x is the matrix, the algorithm is as follow:

matrix[2][2] = {{1, 1,}, {1, 0}}

power(matrix[2][2], n )

{

if(n == 0 || n == 1)

return;

int matrixHelper[2][2] = = {{1, 1,}, {1, 0}};

power(matrix, n / 2);

multiply(matrix, matrix);

if(n % 2 != 0)

multiply(matrix, matrixHelper);

}

multiply function is the function to calculate two matrix whose size is both 2 x 2;

4.

Suppose N = A + B, which means I have to sell all items to either Alice or Bob, I will sell from the first item, and compare a[1] and b[1], if a[1] > b[1], I will sell it to Alice, otherwise, I will sell it to Bob,

for i-th item, compare a[i] and b[i]:

a[i] > b[i] ? sell it to Alice : sell it to Bob.

This takes O(n) time.

If N <= A + B, I will first store array a[] and b[] like this: the data in a[] and b[] have satellite data, the data is like a tuple: (item i , value for i). Then use merge sort to sort a[] and b[] according to the value for i in each tuple. This takes O(nlogn) time.

Then starts from the last elements of both arrays, the last element is the tuple (i, value for i) where value for i is largest.

If Bod and Alice both want to pay most to buy the same item

If N <= A + B, I will use merge sort to sort the items according to the maximum price that Alice and Bob would like to pay, which means sort the i-th item according to max(a[i], b[i]).

Then starts from the most valuable item, compare a[i] and b[i], if a[i] > b[i], sell it to Alice, and A = A – 1, if b[i] > a[i], sell it to Bob, and B = B – 1. If a[i] == b[i], sell it either to Alice or Bob, and set the A = A – 1 or B = B – 1, then move to the second most valuable item, repeat the process. If A == 0 and B != 0, then I cannot sell any item Alice, from now on, I have to sell items to Bob, so no need to compare, just sell the items according to the values. If B == 0 and A != 0, same with the former situation(A == 0 and B != 0).

Repeat above process until A == 0 and B == 0.

This takes O(nlogn).

5.

(a).

Firstly, go through the heights array H, save the element’s index whose height is no less than T, for example:

The output of this operation is: indexArray = [i for i in range(len(H)) if H[i] >= T],

For example T = 7, we get: [1, 5, 6, 7, 8], which means that H[1], H[5] , H[6], H[7], H[8], their heights are no less than T = 7.

Then check in the indexArray, whether these exists L elements that their difference of value is larger than K.

Pseudo code:

Input array: indexArray

Output: true or false, whether these exists L elements that their difference of value is larger than K.

Set i = 0;

Set j = 1;

Set count = 1;

while(j < indexArray.size())

{

if(indexArray[j] - indexArray[i] > K)

{

count++;

i = j;

}

j++;

}

if count >= L return true, else return false;

Time Complexity: O(n) + O(n) = 2 \* O(n) = O(n).

(b)

Use merge sort to sort the array H, get a new array H1.

Then use the decision version of this problem, illustrated above.