COMP3121/3821/9101/9801 18s1
Assignment 2

Jia Jun Luo — z5115679

1. (a) [**5 marks**] *Revision:* Describe how to multiply two n-degree polynomials together in $O(nlogn)$ time, using the Fast Fourier Transform (FFT). You do not need to explain how FFT works - you may treat it as a black box.
Hint: Remember that FFT does not multiply polynomials, what does it do? Refer to the lecture slides.

   (b) In this part we will attend the Fast Fourier Transform (FFT) algorithm described in class to multiply multiple polynomials together (not just two). Suppose you have K polynomials $P_1, ..., P_K$ so that:

$$\text{degree}(P_1)+...+\text{degree}(P_K) = S$$

   i. [**5 marks**] Show that you can find the product of these K polynomials in $O(KSlogS)$ time.
Hint: How many points do you need to uniquely determine an S -degree polynomial?

   ii. [**10 marks**] Show that you can find the product of these K polynomials in $O(SlogSlogK)$ time. Explain why your algorithm has the requried time complexity.
Hint: consider using divide-and-conquer!

---

a) The convolution of two signals in the discrete time domain is equivalent to the multiplication of two signals in the frequency domain (which can be represented as two n degree polynomials). This can be done using FFT.

To multiply two n degree polynomials, we first perform the Discrete Fourier Transform (FFT) on each of the polynomials. This allows us to obtain the coefficients of the complex roots of unity from these polynomials in $O(nlogn)$ time. Then, we perform $2n + 1$ multiplications in $O(n)$ time to obtain the coefficients of the product. However since this still within the discrete time spectrum, we convert back into the discrete time domain using the Inverse Discrete Fourier Transform in $O(nlogn)$ time, thus giving the final solution in $O(nlogn)$ time.

b)

i) First, we perform zero padding on all K polynomials such that they all have S terms. Then we perform the DFT on each of the polynomials, and hence, this process takes $O(KSlogS)$. After this, the multiplication of each pair of the polynomials continues, which is of $O(K)$ to end up with $S+1$ distinct points. Finally, the IDFT is performed to find the final product, which is of order $O((S+1)log(S+1))$. The final time complexity is therefore: $KSlogS + K + (S+1)log(S+1) = KSlogS + SlogS = (K+1)SlogS = O(KSlogS)$, as required.

ii) To find the polynomials in $O(SlogSlogK)$, we simply apply the process above to each pair of polynomials in K (and thus divide and conquer), to reduce K to $logK$. Instead of zero padding all K polynomials such that they all have S terms, and then performing DFT on each and every polynomial, we do the following:

Observe polynomial P_1 and P_2. Also observe that degree(P_1) + degree(P_2) = $S_{1,2}$. Therefore, we zero pad both of these polynomials to have $S_{1,2}$ terms, and multiply them via DFT. Then we continue with P_3 and P_4, and apply the same process. At the next level, we do the same with $P_{1,2}$ and $P_{3,4}$ to obtain $P_{1,2,3,4}$. We continue this until we get $P_{1,2,3...k}$ which will have $S+1$ distinct points, and then finally we perform IDFT to obtain the final product in $O((S+1)log(S+1))$ time. At each level, note that we are effectively halving K, so the DFT process takes $O(logKSlogS)$ instead of $O(KSlogS)$. The final time complexity is therefore: $logKSlogS + logK + (S+1)log(S+1) = logKSlogS + SlogS = (logK+1)SlogS = O(SlogSlogK)$, as required.

2. **[10 marks]** You have a set of N coins in a bag, each having a value between 1 and M, where $M \geq N$. Some coins may have the same value. You pick two coins (**without replacement**) and record the sum of their values. Determine what possible sums can be achieved, in $O(MlogM)$ time.
For example, if there are $N = 3$ coins in the bag with values 1,4 and 5 (so we could have $M = 5$), then the possible sums are 5,6 and 9.
Hint: if the coins have values $v_1, ..., v_N$, how might you use the polynomial $x^{v_1} + ... + x^{v_N}$?

Consider a sequence of 5 numbers $< v_1, v_2, v_3, v_4, v_5 >$ as set A. We choose $< v_2, v_4 >$, remove it from set A and name it as a different set B. Then we perform a convolution of A and B, using FFT in $O(nlogn)$. This produces the sequence $< v_1 \cdot v_4, v_1 \cdot v_2 + v_3 \cdot v_4, v_3 \cdot v_2 + v_5 \cdot v_4, v_5 \cdot v_2 >$. A is now $< v_1, v_3, v_5 >$ and now we choose a new set $< v_1, v_3 >$. We again perform FFT to give us a sequence of $< v_5 \cdot v_3, v_5 \cdot v_1 >$ and then we also compute at the end $v_1 + v_3$.

Now, we have some multiplications and some additions. To deal with this, we treat all multiplications as additions and all additions as multiplications, when we are computing the convolutions. Hence the sum: $v_1 \cdot v_2 + v_3 \cdot v_4$ becomes $(v_1 + v_2)(v_3 + v_4)$ (similar to De Morgan's Law). This is applied to all of the final sequences after convolution, giving us all the possible sums. The full algorithm is as follows:

(a) While input sequence is not empty:
   i. Choose 2 numbers without replacement from sequence and perform FFT with remaining sequence and selected sequence of 2 numbers.
   ii. Keeping track of additions and multiplications, we exchange additions for multiplications and multiplications for additions in the resulting FFT sequence. Then we record for each the sum of each factorisation $v_i + v_j$.
   iii. If the remaining sequence only has 2 numbers left, we add the two numbers to finish.
   iv. If the remaining sequence only has 1 number left, we must make sure to add the 2 numbers we have chosen without replacement.

Since we only use FFT, and we only iterate through the final sequence once after each FFT, the final time complexity is $n + MlogM = O(MlogM)$, as required.

3. 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) [**5 marks**] 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 $n \geq 1$.

   (b) [**10 marks**] Hence or otherwise, give an algorithm that finds F_n in $O(logn)$ time. *Hint: You may wish to refer to Example 1.5 on page 28 of the "Review Material" found as lecture notes for Topic 0 on the Course Resources page of the course website.*

---

a) Proof by induction:

Let $n = 1$

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

LHS =

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

Therefore the equation holds true for n = 1.
Let $n = k$

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

Let $n = k + 1$

$$\begin{pmatrix} F_{(k+1)+1} & F_{(k+1)} \\ F_{(k+1)} & F_k \end{pmatrix} = \begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{k+1}$$

RHS =

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

But as

$$\boxed{F_n = F_{n-1} + F_{n-2}\ ,\ n \geq 2} \tag{1}$$

Therefore:

$$F_{k+1} = F_k + F_{k-1}$$

and

$$F_{k+2} = F_{k+1} + F_k$$

Therefore: RHS =

$$\begin{pmatrix} F_{k+2} & F_{k+1} \\ F_{k+1} & F_k \end{pmatrix}$$

as required, hence proven by induction that this equation holds for all integers iff $n \geq 1$.

b) We can do this problem in $O(logn)$ by dividing the problem (the power n) in half. As shown on page 28 of the "Review Material", $y = x^n$ can be computed as such: $x^0 = 1, x^{2n} = x^{n^2} = y^2, x^{2n+1} = x \cdot x^{n^2} = x \cdot y^2$. We note that $2n + 1$ is odd and $2n$ is even. We can apply this to the Fibonacci matrix representation.

The algorithm is as follows:

(a) We first begin by finding $F_0 = 0$ and $F_1 = 1$. Then we can start finding the other parts of the final matrix. Given an identity matrix (to represent the resulting matrix X in it's current form throughout the recursion) and the Fibonacci matrix Y as shown on the RHS of the equation,

(b) If $n \leq 1$, return n.

(c) While $n > 0$,

   i. If n is odd as noted before, we must multiply the resulting matrix by the Fibonacci matrix ($X \ * \ Y$).

   ii. Then we divide n by 2, which is a situation that applies to both when n is odd and even as shown previously, and multiply the Fibonacci matrix by the Fibonacci matrix ($Y \ * \ Y$).

(d) Return the resulting matrix X.

4. **[10 marks]** You have N items for sale, numbered 1 to N. Alice is willing to pay $a[i] > 0$ dollars for item i, and Bob is willing to pay $b[i] > 0$ dollars for item i. Alice is willing to buy no more than A of your items, and Bob is willing to buy no more than B of your items. Additionally, you must sell each item to either Alice or Bob, but not both, so you can determine the **maximum** total amount of money you can earn in $O(NlogN)$ time.
Hint: Suppose $N = A + B$ and pretend you wish to sell all items to Alice, but must choose B of them to give to Bob instead. Which ones do you want to give to Bob first? Then extend your approach to handle $N < A + B$ as well.

Let $a[i]$ and $b[i]$ be doubly linked lists consisting of the price of the item and the index of the item.

If we consider the case $N = A + B$ and that we pretend to want to sell all the items to Alice, but must choose B of them to give to Bob instead: We can start by MergeSorting / Quicksorting in $O(nlogn)$ $a[i]$ and $b[i]$ in terms of decreasing price, we can then sell the first B items in b to Bob. The rest of the items can be sold to A. This will ensure the maximum total profit. This iteration of the lists and the items will be of $O(n)$.

If we consider the case $N < A + B$, while still wishing to sell all the items to Alice, we can do the same strategy as stated previously; we sell the first B items in b to Bob, followed by selling the rest to Alice (this is still assuming we must choose B of them to give to Bob instead). However, if we now want to maximise profit, the full algorithm is as such:

MergeSort / Quicksort a and b in descending order of price. Compare $a[1]$ to $b[1]$. If $a[1] >= b[1]$, we start with $a[1]$ and alternate from $a[i]$ to $b[j]$ iff $a[i]$ becomes less than $b[j]$, selling each item to the highest bidder until all N items are exhausted. If $b[1] < a[1]$, we simply start from b, applying the same process. The N items can be represented as a linked list with the item, item index and if it's sold or not. During the above process, we simply access $N[i]$ and mark it as sold in $O(1)$ time. Once all N items are sold, the algorithm finishes in $O(NlogN)$ time, as required.

5. Your army consists of a line of N giants, each with a certain height. You must designate precisely $L \leq N$ of them to be leaders. Leaders must be spaced out across the line; specifically, every pair of leaders must have at least $K \geq 0$ 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 $N = 10, L = 3, K = 2, H = [1, 10, 4, 2, 3, 7, 12, 8, 7, 2]$. 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) [**10 marks**] 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.

   (b) [**10 marks**] Hence, show that you can solve the optimisation version of this problem in $O(NlogN)$ time.

---

a) To decide if there exists a leader in H that satisfies the conditions and has a height no less than T, consider the following steps with the above example:

First, we iterate through the entire array, selecting at least $P \geq L$ leaders, until we reach the end of the array. To do this, we always choose the first in the array to be our first leader, skip the minimum K gap by observing the index $i + K + 1$ to be our next leader, and continue until the end of the array. This process looks like this (leaders are in bold and bracketed):

Giants	**(1)**	10	4	**(2)**	3	7	**(12)**	8	7	**(2)**
Index	0	1	2	3	4	5	6	7	8	9

Then, we remove our selected leaders, and continue until we find a suitable leader that has a height no less than T. When we have found such a leader, the algorithm stops in $O(N)$ time.

Giants	**(10)**	4	**(3)**	7	**(8)**	7
Index	1	2	4	5	7	8

Continuing...

Giants	**(4)**	**(7)**	**(7)**
Index	2	5	8

Note that in this specific case, all of the giants could have been leaders. However the final choice depends on the optimisation stage.

b)

We start by making a small modification to the algorithm in the previous question. Instead of looking for "if there exists some valid choice of leaders satisfying the constraints whose shortest leader has height no less than T", we keep track of all possible leaders whose height is greater than T. If we want the shortest leader to have the maximum height among all valid choices of leaders, that means we want the shortest leader in our list of viable leaders after we perform the modified algorithm in part a). We then perform MergeSort in $O(NlogN)$ on our list of viable leaders so the height of the possible leaders are in increasing order, and pick our first leader in the list to be our shortest leader.

For example, if $T = 6$, that means our possible leaders are (referring from the example in the question):

Possible Leaders	7	7	8	10	12
Index	5	8	7	1	6

Finally, we iterate through our leaders, choosing our leaders until we have L leaders (the algorithm exits here). We must also keep in mind that the minimum gap between leaders is K:

Possible Leaders	**(7)**	**(7)**	8	**(10)**	12
Index	5	8	7	1	6

Note that even if the first two 7's were swapped, there is still enough space between the first two 7's for the two 7's to be selected as the final leaders.

The optimisation stage consists of $T(N) = NlogN + N = O(NlogN)$, as required.