CS 170 Homework 3

Due **2/10/2020, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 Modular Fourier Transform

Fourier transforms (FT) have to deal with computations involving irrational numbers which can be tricky to implement in practice. Motivated by this, in this problem you will demonstrate how to do a Fourier transform in modular arithmetic, using modulo 5 as an example.

(a) There exists $\omega \in \{0, 1, 2, 3, 4\}$ such that ω are 4^{th} roots of unity (modulo 5), i.e., solutions to $z^4 = 1$. When doing the FT in modulo 5, this ω will serve a similar role to the primitive root of unity in our standard FT. Show that $\{1, 2, 3, 4\}$ are the 4^{th} roots of unity (modulo 5). Also show that $1 + \omega + \omega^2 + \omega^3 = 0 \pmod 5$ for $\omega = 2$.

(b) Using the matrix form of the FT, produce the transform of the sequence $(0, 1, 0, 2)$ modulo 5; that is, multiply this vector by the matrix $M_4(\omega)$, for the value $\omega = 2$. Be sure to explicitly write out the FT matrix you will be using (with specific values, not just powers of ω). In the matrix multiplication, all calculations should be performed modulo 5.

(c) Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. (Again all arithmetic should be performed modulo 5.)

(d) Now show how to multiply the polynomials $2x^2 + 3$ and $-x + 3$ using the FT modulo 5.

Solution:

(a) We can check that $1^4 = 1 \pmod 5$,
$2^4 = 16 = 1 \pmod 5$,
$3^4 = 81 = 1 \pmod 5$,
$4^4 = 256 = 1 \pmod 5$.

Observe that taking $\boxed{\omega = 2}$ produces the following powers: $(\omega, \omega^2, \omega^3) = (2, 4, 3)$. Verify that

$$1 + \omega + \omega^2 + \omega^3 = 1 + 2 + 4 + 3 = 10 = 0 \pmod 5.$$

(b) For $\omega = 2$:

$$M_4(2) = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 3 \\ 1 & 4 & 1 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}.$$

Multiplying with the sequence $(0, 1, 0, 2)$ we get the vector $\boxed{(3, 3, 2, 2)}$.

(c) For $\omega = 2$, the inverse matrix of $M_4(2)$ is the matrix

$$4^{-1} \cdot \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 3 & 4 & 2 \\ 1 & 4 & 1 & 4 \\ 1 & 2 & 4 & 3 \end{bmatrix}$$

Verify that multiplying these two matrices mod 5 equals the identity. Also multiply this matrix with vector $(3, 3, 2, 2)$ to get the original sequence.

(d) For $\omega = 2$: We first express the polynomials as vectors of dimension 4 over the integers mod 5: $a = (3, 0, 2, 0)$, and $b = (3, -1, 0, 0) = (3, 4, 0, 0)$ respectively. We then apply the matrix $M_4(2)$ to both to get the transform of the two sequences. That produces $(0, 1, 0, 1)$ and $(2, 1, 4, 0)$ respectively. Then we just multiply the vectors coordinate-wise to get $(0, 1, 0, 0)$. This is the transform of the product of the two polynomials. Now, all we have to do is multiply by the inverse FT matrix $M_4(2)^{-1}$ to get the final polynomial in the coefficient representation. Recall that whe working with the FT outside of modspace, our inverse matrix of $M_4(\omega)$ would be given by $\frac{1}{4}M_4(\omega^{-1})$. In modspace, we can replace $\frac{1}{4}$ with the multiplicative inverse of 4, and ω^{-1} with the multiplicative inverse of 2. The properties of 2 that we found in the first part allow for this identity to hold. Thus the product is as follows: $\boxed{(4, 2, 1, 3)}$ or $\boxed{3x^3 + x^2 + 2x + 4}$.

3 Inverse FFT

Recall that in class we defined M_n, the matrix involved in the Fourier Transform, to be the following matrix:

$$M_n = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{bmatrix},$$

where ω is a primitive n-th root of unity.
For the rest of this problem we will refer to this matrix as $M_n(\omega)$ rather than M_n. In this problem we will examine the inverse of this matrix.

(a) Define

$$M_n(\omega^{-1}) = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \dots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \dots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Recall that $\omega^{-1} = 1/\omega = \bar{\omega} = \exp(-2\pi i/n)$.

Show that $\frac{1}{n}M_n(\omega^{-1})$ is the inverse of $M_n(\omega)$, i.e. show that

$$\frac{1}{n}M_n(\omega^{-1})M_n(\omega) = I$$

where I is the $n \times n$ identity matrix – the matrix with all ones on the diagonal and zeros everywhere else.

(b) Let A be a square matrix with complex entries. The *conjugate transpose* $A^\dagger$ of A is given by taking the complex conjugate of each entry of A^T. A matrix A is called *unitary* if its inverse is equal to its conjugate transpose, i.e. $A^{-1} = A^\dagger$. Show that $\frac{1}{\sqrt{n}}M_n(\omega)$ is unitary.

(c) Suppose we have a polynomial $C(x)$ of degree at most $n-1$ and we know the values of $C(1), C(\omega), \ldots, C(\omega^{n-1})$. Explain how we can use $M_n(\omega^{-1})$ to find the coefficients of $C(x)$.

Solution:

(a) We need to show that the entry at position (j,k) of $M_n(\omega^{-1})M_n(\omega)$ is n if $j = k$ and 0 otherwise. Recall that by definition of matrix multiplication, the entry at position (j,k) is (where we are indexing the rows and columns starting from 0):

$$\begin{aligned}\sum_{l=0}^{n-1} M_n(\omega^{-1})_{jl} M_n(\omega)_{lk} &= \sum_{l=0}^{n-1} \omega^{-lj}\omega^{kl} \\ &= \sum_{l=0}^{n-1} \omega^{-lj+kl} \\ &= \sum_{l=0}^{n-1} \omega^{l(k-j)}\end{aligned}$$

If $j = k$ then this just becomes

$$\begin{aligned}\sum_{l=0}^{n-1} \omega^{0\cdot l} &= \sum_{l=0}^{n-1} \omega^0 \\ &= \sum_{l=0}^{n-1} 1 \\ &= n\end{aligned}$$

On the other hand, if $j \neq k$ then $\omega^{k-j} \neq 1$ so we can use the formula for summing a geometric series, namely

$$\sum_{l=0}^{n-1} \omega^{l(k-j)} = \frac{1-\omega^{n(k-j)}}{1-\omega^{k-j}}$$

Now recall that since ω is an n^{th} root of unity, ω^{nm} for any integer m is equal to 1. Thus the expression above simplifies to

$$\frac{1-1}{1-\omega^{k-j}} = 0$$

Here's another nice way to see this fact. Observe that we can factor the polynomial $X^n - 1$ to get

$$X^n - 1 = (X-1)(X^{n-1} + X^{n-2} + \ldots + X + 1)$$

Now observe that ω^{k-j} is a root of $X^n - 1$ and thus it must be a root of either $X - 1$ or $X^{n-1} + X^{n-2} + \ldots + X + 1$. And if $k \neq j$ then $\omega^{k-j} \neq 1$ so it cannot be a root of $X - 1$. Thus it is a root of $X^{n-1} + X^{n-2} + \ldots + X + 1$, which is equivalent to the statement that $\omega^{(n-1)(k-j)} + \omega^{(n-2)(k-j)} + \ldots + \omega^{k-j} + 1 = 0$, which is exactly what we were trying to prove.

(b) Observe that $(M_n(\omega))^\dagger = M_n(\omega^{-1})$. So $\frac{1}{\sqrt{n}} M_n(\omega) \frac{1}{\sqrt{n}} (M_n(\omega))^\dagger = \frac{1}{n} M_n(\omega) M_n(\omega^{-1}) = I$.

(c) Let $c_0, \ldots, c_{n-1}$ be the coefficients of $C(x)$. Then as we saw in class,

$$M_n(\omega) \begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} C(1) \\ C(\omega) \\ \vdots \\ C(\omega^{n-1}) \end{bmatrix}$$

Thus

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_{n-1} \end{bmatrix} = M_n(\omega)^{-1} \begin{bmatrix} C(1) \\ C(\omega) \\ \vdots \\ C(\omega^{n-1}) \end{bmatrix}$$

And as we showed in part (a), $M_n(\omega)^{-1} = (1/n) M_n(\omega^{-1})$. Thus to find the coefficients of $C(x)$ we simply have to multiply $(1/n) M_n(\omega^{-1})$ by the vector $\begin{bmatrix} C(1) & C(\omega) & \ldots & C(\omega^{n-1}) \end{bmatrix}$.

4 Polynomial from roots

Given a polynomial with exactly n distinct roots at $r_1, \ldots, r_n$, compute the coefficient representation of this polynomial. Your runtime should be $\mathcal{O}(n \log^c n)$ for some constant $c > 0$ (you should specify what c is in your runtime analysis). There may be multiple possible answers, but your algorithm should return the polynomial where the coefficient of the highest degree term is 1.

You can give only the algorithm description and runtime analysis, a three-part solution is not required.

Hint: A root of a polynomial p is a number r such that $p(r) = 0$. One polynomial with roots $r_1, ..., r_k$ is

$$p(x) = \prod_{i=1}^{k}(x - r_i)$$

.

Solution:

Main idea

We are trying to find the coefficients of the polynomial $(x - r_1)(x - r_2) \cdots (x - r_n)$. Split the roots into two (approximately) equal halves: $r_1, r_2, \ldots, r_{\lfloor n/2 \rfloor}$ and $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Recursively find the polynomial whose roots are $r_1, \ldots, r_{\lfloor n/2 \rfloor}$, and the polynomial whose roots are $r_{\lfloor n/2 \rfloor + 1}, \ldots, r_n$. Multiply these two polynomials together using FFT, which takes $O(n \log n)$. When the base case is reached with only 1 root r, return $(x - r)$.

Runtime Analysis

The recurrence for this algorithm is $T(n) = 2T(n/2) + O(n \log n)$. Note that the master theorem doesn't apply here, so we'll have to solve this recurrence relation out directly. If we write out the recurrence tree, we can see that on the ith level, we do $2^i * (\frac{n}{2^i}) * log(\frac{n}{2^i})$ work for $\log n$ levels. We can upper bound the work on each level with $n \log n$, to get a total runtime of $\mathrm{O}(n \log^2 n)$.

5 Triple sum

We are given an array $A[0..n-1]$ with n elements, where each element of A is an integer in the range $0 \leq A[i] \leq n$ (the elements are not necessarily distinct). We would like to know if there exist indices i, j, k (not necessarily distinct) such that

$$A[i] + A[j] + A[k] = n$$

Design an $\mathcal{O}(n \log n)$ time algorithm for this problem. Note that you do not need to actually return the indices; just yes or no is enough.

Please give a 3-part solution to this problem.

Solution:

Main idea Exponentiation converts multiplication to addition. Observe $x^3 * x^2 = x^{2+3} = x^5$. So, define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form $x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether $p(x)^3$ contains x^n as a term.

Pseudocode

procedure ALGORITHM TRIPLESUM($(A[0 \ldots n-1],\ t)$)
  Set $p(x) := \sum_{i=0}^{n-1} x^{A[i]}$.
  Set $q(x) := p(x) \cdot p(x) \cdot p(x)$, computed using the FFT.
  Return whether the coefficient of x^n in q is nonzero.

Proof of Correctness Observe that

$$q(x) = p(x)^3 = \left(\sum_{0 \leq i < n} x^{A[i]} \right)^3 = \left(\sum_{0 \leq i < n} x^{A[i]} \right) * \left(\sum_{0 \leq j < n} x^{A[j]} \right) * \left(\sum_{0 \leq k < n} x^{A[k]} \right)$$

$$= \sum_{0 \leq i,j,k < n} x^{A[i]} x^{A[j]} x^{A[k]} = \sum_{0 \leq i,j,k < n} x^{A[i]+A[j]+A[k]}.$$

Therefore, the coefficient of x^n in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] = n$. So the algorithm is correct. (In fact, it does more: the coefficient of x^n tells us *how many* such triples (i, j, k) there are.)

Runtime Analysis Constructing $p(x)$ clearly takes $O(n)$ time. $p(x)$ is a polynomial of degree at most $n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^t takes constant time, so overall the algorithm takes $O(n \log n)$ time.

Comment: This problem promised you that each element of the array is in the range $0 \ldots n$. What if we didn't have any such promise? Then the FFT-based method above becomes inefficient (because the degree of the polynomial is as large as the largest element of A). It is easy to find a $O(n^2)$ time algorithm, but no faster algorithm is known. In particular, it is a famous open problem (called the 3SUM problem) whether this problem can be solved more efficiently than $O(n^2)$ time. This problem has been studied extensively, because it is closely connected to a number of problems in computational geometry.

6 Searching for Viruses

Sherlock Holmes is trying to write a computer antivirus program. He thinks of computer RAM as being a binary string s_2 of length m, and a virus as being a binary string s_1 of length $n < m$. His program needs to find all occurrences of s_1 in s_2 in order to get rid of the virus. Even worse, though, these viruses are still damaging if they differ slightly from s_1. So he wants to find all copies of s_1 in s_2 that differ in at most k locations for arbitrary $k \leq n$.

(a) Give a $O(nm)$ time algorithm for this problem. **Solution:**

For each of $i \in \{0, 1, \ldots, m-n\}$ starting points in s_2, check if the substring $s_2[i : i+n-1]$ differs from s_1 in at most k positions. The check takes $O(n)$ time at each of $O(m)$ starting points, so the time complexity is $O(mn)$.

(b) Give a $O(m \log m)$ time algorithm for any k.

Solution: If we replace the 0's in a bit string with -1's, checking whether two length n strings differ at no more than k bits is equivalent to checking whether the dot product of the two bit strings (i.e. bit-wise multiplication and add up the results) is at least $n - 2k$, as the positions they agree will contribute 1 to the dot product, while positions they disagree will contribute -1 to the dot product.

Now all we need are the $m - n + 1$ dot products of all the $m - n + 1$ length n substrings of s_2 with the bit string s_1. For people familiar with convolution, it should be straightforward to see this is exactly the same computation.

To present it as polynomial multiplication, consider $p_1(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ as a degree $n-1$ polynomial with $a_d = s_1'(n-d-1)$ for all $d \in \{0, 1, \ldots, n-1\}$, where s_1' is just s_1 with 0's replaced by -1's. $p_2(x) = b_0 + b_1 x + \cdots + b_{m-1}x^{m-1}$ is a degree $m-1$ polynomial with $b_d = s_2'(d)$ for all $d \in \{0, 1, \ldots, m-1\}$, where again s_2' is just s_2 with 0's replaced by -1's. Notice $p_1(x)$ is reversed in the sense that the coefficients are in opposite order of the bits in s_1'

Now consider $p_3(x) = p_1(x) \times p_2(x) = c_0 + c_1 x + \cdots$, the coefficient of x^{n-1+j} in $p_3(x)$ is $c_{n-1+j} = \sum_{i=0}^{n-1} a_{n-1-i} b_{j+i} = \sum_{i=0}^{n-1} s_1'(i) s_2'(j+i)$ for any $j \in \{0, 1, \ldots, m-n\}$, which is exactly the dot product of the substring in s_2' starting at index j and the string s_1'. Thus all we need is to compute $p_3(x)$, and output all the j's between 0 and $m-n$ such that $c_{n-1+j} \geq n - 2k$.

The computation takes a FFT, a point-wise product, an inverse FFT, and a linear scan of the coefficients. The running time of this algorithm, $O(m \log n)$, is dominated by the FFT and inverse FFT steps, which each take $O(m \log m)$ time. The point-wise product and search for $c(i) \geq n - 2k$ each take $O(n)$ time.

You do not need a 3-part solution for either part. Instead, describe the algorithms clearly and give an analysis of the running time.

7 Vertex Cut

Let $G = (V, E)$ be an undirected, unweighted graph with $n = |V|$ vertices. The *distance* between two vertices $u, v \in G$ is the length of the shortest path between them. A *vertex cut* of G is a subset $S \subseteq V$ such that removing the vertices in S (as well as incident edges) disconnects G.

Show that if there exist $u, v \in G$ of distance $d > 1$ from each other, that there exists a vertex cut of size at most $\frac{n-2}{d-1}$. Assume G is connected.

Solution: For each $k \geq 0$, let U_k be the set of vertices of distance k from u. Note that the U_k are disjoint (each vertex has a well-defined distance to u), $U_0 = \{u\}$, and $v \in U_d$.
Also notice that for any k, j where $k > j+1$, there cannot be an edge between a vertex in U_k and a vertex in U_j (if there was, then some vertex in U_k would be of distance only $j+1 < k$ from u). But for every j where U_j and U_{j+1} are nonempty, there must be an edge from a vertex in U_j to U_{j+1} (otherwise G would not be connected). So any nonempty U_k where $k \geq 1$ is a vertex cut (if we remove it, there cannot exist paths from U_j where $j < k$ to U_i where $i > k$).
Consider $U_1, \ldots, U_{d-1}$. These $d-1$ sets of vertices are disjoint, and all together they have at most $n-2$ vertices (excluding u and v). In addition, all of them must be nonempty for a path to exist from u to v.
We show that one of these sets must have size at most $\frac{n-2}{d-1}$. If this was not the case, they would all have size $> \frac{n-2}{d-1}$, meaning there would be $> \frac{n-2}{d-1}(d-1) = n-2$ vertices among them, which is impossible because there are at most $n-2$ vertices among these sets. (Not everyone is above average).
As one of $U_1, \ldots, U_{d-1}$ has size at most $\frac{n-2}{d-1}$, it is a vertex cut of size $\frac{n-2}{d-1}$.