CS 170 Homework 3

Due **9/18/2023, at 10:00 pm (grace period until 11:59pm)**

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".

Solution: None.

2 Inverse FFT

Recall from lecture that a polynomial of degree d can either be specified by its coefficients or by its values on $d+1$ points. If we choose the points to be roots of unity then we can use FFT to switch between the two representations efficiently.

(a) Consider two polynomials $A(x) = 1 + 5x + 3x^2 + 4x^3$ and $B(x) = 3 + 4x + 2x^3$. Pick an appropriate value of n and write down the values of $A(x)$, $B(x)$ and $C(x) = A(x)B(x)$ at each of the n roots of unity.

Your n should be large enough to allow recovery of the coefficients of C from its evaluation on the n points using inverse FFT. What value of n did you pick?
(Note: n should be a power of 2).

Hint: What will be the degree of C?

Solution: We see that the degree of C is going to be the sum of the degrees of A and B, hence C will be a degree 6 polynomial. Therefore, we should pick n to be 8, Since this is the smallest power of 2 that is larger than 6. I read on Ed discussion that we won't be evaluating, so I won't do that. If I was, I'd use WolframAlpha.

Now we will focus recovering the coefficients of C given its evaluation on n points. 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. We will examine some proprties of the inverse of this matrix.

(b) 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.

Solution: We see by inspection that M_n that its elements can be written as: $(M_n(\omega))_{ij} = \omega^{ij}$. Further, We see that $(M_n(\omega^{-1}))_{ij} = \omega^{-ij}$. This is useful because when we take the product of the two, the (i, j)-th entry is marked by taking the dot product of the i-th row with the j-th column. we see that we multiply one row with one column. Let i denote the row, and j denote the column, then for a particular multiplication, we can write this as:

$$M_{ij} = \sum_{k}^{n}(M_n(\omega^{-1}))_{ik}(M_n(\omega))_{kj} = \sum_{k=1}^{n}(\omega^{-ik})(\omega^{kj}) = \sum_{k=1}^{n}(\omega^{i-j})^k$$

When $i \neq j$, then $i - j \neq 0$, so we can use the relation given in lecture:

$$\sum_{k=0}^{n}\omega_j^k = \begin{cases} m & j = 0 \\ 0 & \text{otherwise} \end{cases}$$

This tells us that whenever $i - j \neq 0$, then we have:

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

Alternatively, if $i = j$, then $i - j = 0$, so:

$$\sum_{k=1}^{n}(\omega^{0})^k = n$$

So in other words, when we are on an off-diagonal element, the matrix element is zero, but when we're on a diagonal element, then the matrix element is n. Therefore, we have the following matrix:

$$M_n(\omega^{-1})M_n(\omega) = \begin{bmatrix} n & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & n \end{bmatrix}$$

Multiplying this by $\frac{1}{n}$ as indicated by the equation gives:

$$\frac{1}{n}M_n(\omega^{-1})M_n(\omega) = \frac{1}{n}\begin{bmatrix} n & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & n \end{bmatrix} = \begin{bmatrix} 1 & \dots & 0 \\ 0 & \ddots & 0 \\ 0 & \dots & 1 \end{bmatrix} = I$$

as desired.

(c) 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.

Solution: First, we use the insight from the previous problem that $M_{ij} = M_{ji}$, since $M_{ij} = \omega^{ij} = \omega^{ji} = M_{ji}$, so in other words, M is symmetric. Further, we'll use the fact that since ω is a root of unity, that $(\omega^k)^* = \omega^{-k}$. If we combine these two facts, we see that:

$$(M_n(\omega))^\dagger = (M_n(\omega)^\top)^*$$

Now if we look at this element wise:

$$(M_n(\omega))^\dagger = M_{ji}^* = (\omega^{ji})^* = (\omega^{-ji}) = (M_n(\omega^{-1}))_{ji}$$

In other words, this means that $(M_n(\omega))^\dagger = M_n(\omega^{-1})$. Mathematically this is shown above, but intuitively this is the case because M is a symmetric matrix so the transpose does nothing, then taking the conjugate flips the sign of the exponent, giving us $M_n(\omega^{-1})$.

This is useful because now we know that $(M_n(\omega))^\dagger = M_n(\omega^{-1})$, meaning that we can finally show:

$$\frac{1}{\sqrt{n}} M_n(\omega) \cdot \left(\frac{1}{\sqrt{n}} M_n(\omega) \right)^\dagger = \frac{1}{n} M_n(\omega) M_n(\omega^{-1}) = I$$

as desired.

(d) 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: Here we could recover the coefficients by applying the inverse Fourier transform, which is given in matrix form from $M_n(\omega^{-1})$. We can write this out in summation notation using the equation from lecture:

$$c_l = \frac{1}{m} \sum_{j=0}^{m-1} c(\omega_j)(\omega_{m-l})^j$$

or we could also write this out in matrix notation:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = M_n(\omega^{-1}) \begin{bmatrix} C(1) \\ C(2) \\ \vdots \\ C(\omega^{n-1}) \end{bmatrix}$$

(e) Show that $M_n(\omega^{-1})$ can be broken up into four $n/2 \times n/2$ matrices in almost the same way as $M_n(\omega)$. Specifically, suppose we rearrange columns of M_n so that the columns with an even index are on the left side of the matrix and the columns with an odd index are on the right side of the matrix (where the indexing starts from 0). Show that after this rearrangement, $M_n(\omega^{-1})$ has the form:

$$\begin{bmatrix} M_{n/2}(\omega^{-2}) & \omega^{-j} M_{n/2}(\omega^{-2}) \\ M_{n/2}(\omega^{-2}) & -\omega^{-j} M_{n/2}(\omega^{-2}) \end{bmatrix}$$

The notation $\omega^{-j}M_{n/2}(\omega^{-2})$ is used to mean the matrix obtained from $M_{n/2}(\omega^{-2})$ by multiplying the j^{th} row of this matrix by ω^{-j} (where the rows are indexed starting from 0). You may assume that n is a power of two.

Solution: Firstly, the matrix after rearranging can be written as follows:

$$M_n(\omega^{-1}) = \left[\begin{array}{ccc|ccc} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-1} & \omega^{-3} & \omega^{-5} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hline 1 & \omega^{-2k} & \omega^{-4k} & \omega^{-k} & \omega^{-k}\omega^{-2k} & \omega^{-k}\omega^{-4k} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{array}\right]$$

The top left corner is exactly $M_n(\omega^{-2})$, and we don't need to make any modifications to it. From a mathematical point of view, we can argue that this is the case by showing that the (i,j)-th element of $M_{n/2}(\omega)^{-2}$ is the (i,j)-th element of $M_n(\omega^{-1})$ after rearrangement. This element after rearrangement can be written as $(\omega^{-1})^{i(2j)}$, since the $2j$-th column gets shifted to the j-th column. Combining this with the fact that $M_{n/2}(\omega^{-2}) = (\omega^{-2})^{ij}$, then we can confirm that the top left quadrant exactly equals $M_{n/2}(\omega^{-2})$.

For the bottom left corner, a similar story exists. The only difference being that here, instead of the i-th column, we are looking at the $i + \frac{n}{2}$-th column. However, we can write these matrix elements as:

$$(\omega^{-1})^{(i+\frac{n}{2})(2j)} = \omega^{-2ij-jn} = \omega^{-2ij}(\omega^n)^j = \omega^{-2ij}$$

where in the last step I've used the fact that since ω is an element from the n-th root of unity, then $\omega^n = 1$.

Now for the top right corner. Notice that we can write these entries as $\omega^{-3} = \omega^{-1}\omega^{-2}$. In terms of the original matrix elements, column j gets mapped to column $j + \frac{n}{2}$, meaning that every entry within this matrix can be written as $\omega^{-i(j+\frac{n}{2})} = \omega^{-ij}\omega^{-\frac{in}{2}}$. This can be further simplified:

$$\omega^{-ij}\omega^{-\frac{in}{2}} = \omega^{-ij}(\omega^n)^{\frac{i}{2}} = \omega^{-ij}$$

Again, using the fact that $\omega^n = 1$ since ω is an element from the n-th roots of unity. Now, using the fact that j is odd, we can write j as $2k + 1$, so therefore:

$$\omega^{-ij} = \omega^{-i(2k+1)} = \omega^{-2ik-i} = \omega^{-i}\omega^{-2ik}$$

The ω^{-2ik} points to the terms within $M_{n/2}(\omega^{-2})$, and the ω^{-i} correspond to the ω^{-j} term. Sorry for the mismatch in notation.

Finally, the bottom right corner. Here, column j gets mapped to $j + \frac{n}{2}$, and row i also gets mapped to $i + \frac{n}{2}$. Therefore, the elements of our matrix are:

$$\omega^{-(i+\frac{n}{2})(j+\frac{n}{2})} = \omega^{-ij-(i+j)\frac{n}{2}-\frac{n^2}{4}}$$

Note that all the terms with an n in the exponent will immediately cancel since we can always write them as ω^n raised to some exponent. Therefore, we indeed get ω^{-ij} out of the resulting expression. Finally, using the fact that j is odd we can write $j = 2k + 1$ so we get:

$$\omega^{-i(2k+1)} = \omega^{-2ik-i} = \omega^{-i}\omega^{-2ik}$$

The only thing I cannot recover from the original equation is the negatve sign in front of this.

3 Counting k-inversions

A *k-inversion* in a bitstring b is when a 1 in the bitstring appears k indices before a 0; that is, when $b_i = 1$ and $b_{i+k} = 0$, for some i. For example, the string 010010 has two 1-inversions (starting at the second and fifth bits), one 2-inversion (starting at the second bit), and one 4-inversion (starting at the second bit).

Devise an algorithm which, given a bitstring b of length n, counts all the k-inversions, for each k from 1 to $n - 1$. Your algorithm should run faster than $\Theta(n^2)$ time. You can assume arithmetic on real numbers can be done in constant time.

Give a 3-part solution.

Solution: Here we will compute cross correlation using FFT for our algorithm:

1. Copy our bitstring to another place in memory, and flip all its bits (i.e. $1 \longrightarrow 0$, $0 \longrightarrow 1$.)
2. Reverse the order of the bitstring, so the n-th bit appears first,
3. Compute the cross correlation of these two bitstrings using FFT
4. The cross correlation returns a list of length $2n$, from which we only want indices 1 thorugh $n - 1$ (where we're indexing from 0).
5. Reverse this final list, then the i-th element will correspond to the number of i-inversions in the bitstring.

Now for a proof of correctness: Firstly, we bit flip so that when we compute the cross correlation algorithm, a dot product of $1 \cdot 1$ will correspond to a location where the *initial* bitstring has a 1, and the *flipped* bitstring will have a 0 (in other words, this is the location of an inversion). We reverse the bitstring before we invoke cross correlation with FFT because the algorithm itself reverses the bits back, so in effect we are reversing the order when we pass it in, only to have the CC algorithm reverse it back for us. This guarantees that when we are computing cross correlation, we are computing the cross correlation between the original bitstring and the flipped bitstring in original order.

When we cross correlate this way, at every time step, we are multiplying the i-th element of the original bistring to the $n - i$-th element of the flipped bitstring, which will only equal 1 if the original bitstring has a 1 in the i-th location, and a 0 in the $n - i$-th location, corresponding to an $n - i$-inversion. For every 1 that we accumulate in this step, it will correspond to 1 $n - i$ inversion, and this will be accumulated as the value of the i-th term in the returned list, since this is the i-th inversion that we are checking.

Formally, this means that if we have a bitstring $b = 0010$, then we cross-correlate it with $\overline{b} = 1101$, and flip it to be $\overline{b}_{\text{rev}} = 1011$ so that the FFT cross-correlation algorithm flips it back and computes the cross correlation between b and $\overline{b}$. Then, the algorithm begins by checking the first cross-correlation (so checking for $n - 0 = n$ inversion:

$$\begin{array}{r} 00000010 \\ 1101 \end{array}$$

Then it moves on to the second (n-1 inversion)

$$\begin{array}{r} 00000010 \\ 1101 \end{array}$$

Then finally getting to the 1st inversion (where we have a 1):

$$00000010$$
$$1101$$

We see that the second last 1 and the last 1 align together here, indicating that there is a 1-inversion between the 3^{rd} and 4^{th} indices. Finally, we return the values that this FFT algorithm gives us to output the count for the number of inversions.

Finally for a runtime anlaysis: The process of flipping bits takes $O(n)$ time and so does reversing the bits. This doesn't matter in the grand scheme of things, since we invoke cross correlation (using FFT), which completes its task in $O(n \log n)$ time, hence the $O(n)$ terms don't matter. Therefore, the total runtime of this algorithm is $O(n \log n)$.

4 Protein Matching

Often times in biology, we would like to locate the existence of a gene in a species' DNA. Of course, due to genetic mutations, there can be many similar but not identical genes that serve the same function, and genes often appear multiple times in one DNA sequence. So a more practical problem is to find all genes in a DNA sequence that are similar to a known gene.

To model this problem, let g be a length-n string corresponding to the known gene, and let s be a length-m string corresponding to the full DNA sequence, where $m \geq n$. We would like to solve the following problem: find the (starting) location of all length n-substrings of s which match *exactly* match g. For example, using 0-indexing, if $g = ACT$, $s = ACTACTA$, your algorithm should output 0 and 3.

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

Hint: if stuck, refer to discussion Q4.

Solution: Here, since we're given $O(mn)$ time constraint, we can just take our string g, and "slide" it across our longer string s and see where all the indices match. If they do, then we can save this index in a list somewhere, and output this list at the end.

(b) Assume g and s are given as bitstrings, i.e. every character is either 0 or 1. Give a $O(m \log m)$ time algorithm.

Hint: if stuck, refer to discussion Q4.

Solution: We'll follow discussion Q4 here. Since the bitstrings are 0 or 1 in either case, we can represent the bitstrings as polynomials, such that 0 becomes -1 and 1 becomes 1. We do this so that when we cross correlate the two bitstrings together, we would get a -1 for every character that we don't match, and get a 1 instead when our characters do match. Since we want s to exactly match g, then we are looking for the case where the sum of this product over all characters equals the length of s. This can be done using FFT, specifically by computing cross correlation. Also, this is valid since the vectors do not need to be of the same length in order to cross correlate.

The resulting list which is outputted by the cross-correlation algorithm will give us a measure of the "similarity" between the "slice" of g and s. Since we want them to match exactly, we want to look for the values in the list that equal s.

This procedure, as described in lecture, takes $O(m \log m)$ time, satisfying the desired runtime constraint. Specifically, this procedure performs polynomial multiplication twice, which takes $O(m \log m)$ time, and takes constant time to generate and access the values of the bitstring. Hence, the total algorithm does indeed take $O(m \log m)$ time, as desired.

(c) Assume more generally that we know that g and s combined use at most α distinct characters $c_1, c_2, \ldots c_\alpha$ (for example in the previous part, we had $\alpha = 2$. For DNA sequences, we'd have $\alpha = 4$). Give an $O(m \log m)$ time algorithm to find all substrings of length n in s that match g.

Hint: Represent the strings as complex vectors.

Solution: Following the hint, we can represent the strings as complex vectors. Specifically, we since we have α characters we can do the following:

- Assign every character in g to a unique complex vector. Specifically, every unique character will be assigned to one of the α roots of unity, so their algebraic expression would be $e^{\frac{2\pi}{\alpha}}$

- Then, for the corresponding character in s, assign its value to $e^{-\frac{2\pi}{\alpha}}$. In other words, if the character A in g was assigned to $e^{\frac{\pi}{\alpha}}$, then A in s would be assigned to $e^{-\frac{\pi}{\alpha}}$.

Then, we can do the same thing as we did in part (b), by measuring the similarity between each successive slice of s with g, via cross correlation. In this case, the indices that correspond to a matching substring would be the entries from the resulting list that have an imaginary part equal to zero, and these are the indices that we want to return.

As for the proof of correctness: Notice that in our cross-correlation algorithm, the dot product will only equal 1 when we have a matching A in s with a matching A in g, since we have $e^{\frac{\pi}{\alpha}} e^{-\frac{\pi}{\alpha}} = e^0 = 1$ (using the example value of A I defined above). In any other case, the dot product would not yield a purely real number but rather one with a complex part, indicating that there is a mismatch.

For the runtime, this algorithm runs in exactly the same runtime as part (b), since we're executing the same algorithm. The work required to assign the values takes $O(m+n)$ time, but this is insignificant compared to the $O(m \log m)$ time required to execute FFT.

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 $0 \le A[i] \le n$ (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.

Hint 1: elements can be encoded using exponents!

Hint 2: if stuck, refer to discussion Q3.

Solution: We look to Discussion Q3 for guidance on how to design our algorithm. In particular, we can make three copies of the arrays, and encode the array elements as exponents of some polynomial. In particular, let the polynomial be represented as:

$$P(x) = \sum_{a \in A} x^a$$

Then, we can compute $R(x) = P^2(x) = r_0 + r_1 x + \cdots + r_{2n} x^{2n}$. For a specific r_k, then we have:

$$r_k = \sum_{j=0}^{k} p_j p_{k-j}$$

Due to the way that the polynomials are defined, then we know that $p_j p_{k-j} = 1$ only when j and $k-j$ exist within A. Therefore, r_k counts the number of times that we have $A[i] + A[j] = k$, so we can just read off the value of r_k to get that result. Finally, we need to take this polynomial and multiply it again with $P(x)$, since we're looking for $A[i] + A[j] + A[k] = n$. Repeating the same process, we compute a third polynomial $Q(x) = q_0 + q_1 x + \cdots + q_{3n} x^{3n} = P(x)R(x)$. Just like earlier, we can write a specific term within this polynomial as:

$$q_k = \sum_{j=0}^{k} p_j r_{k-j}$$

Note again that p_j exists only if $j \in A$, and since r_{k-j} denotes the number of times we have the sum $A[i] + A[j] = k - j$, then q_k counts the number of times we have $A[i] + A[j] + A[l] = k$. Since we want whether or not this sum can equal n, we just look for whether $q_n > 0$. If it is, then we can return true, otherwise we return false.

As for a quick runtime analysis: We know that polynomial multiplication runs in $O(n \log n)$ time, and we're doing this twice, which still means that we are performing this task in $O(n \log n)$ time, satisfying the runtime constraint.

6 [Coding] FFT & Evaluating Predictions

For this week's homework, you'll implement implement FFT and then apply FFT as a black box to implement some fast algorithms. There are two ways that you can access the notebook and complete the problems:

1. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-fa23-coding`

   and navigate to the `hw03` folder. Refer to the `README.md` for local setup instructions.

2. **On Datahub**: Click here and navigate to the `hw03` folder if you prefer to complete this question on Berkeley DataHub.

Notes:

- *Submission Instructions:* Please download your completed `fft.ipynb` file or submission `.zip` file and submit it to the gradescope assignment titled "Homework 3 Coding Portion".
- *OH/HWP Instructions:* Designated coding course staff will provide conceptual and debugging help during office hours and homework parties.
- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.