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".

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?

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.

(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.

(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 $C(x)$.

(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:

(a) The degree of C will be the sum of the degrees of A and B, which is $3 + 3 = 6$. Thus, we will need at least $6 + 1 = 7$ points to be able to recover the coefficients of C after applying inverse FFT. However, n must be a power of 2, so we want to set $n = 8$.

(b) 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.

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

(d) 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 $\begin{bmatrix} C(1) & C(\omega) & \ldots & C(\omega^{n-1}) \end{bmatrix}$.

(e) We note that if ω is a primitive nth root of unity, then ω^{-1} is also a primitive nth root of unity. This can be seen by recalling that $\omega^{-1} = e^{-i\theta}$ if $\omega = e^{i\theta}$. We now make a simple observation that the only property we used in providing that $M_n(\omega)$ decomposed into 4 smaller matrices was that ω was a primitive nth root of unity. Therefore, if we apply the mapping $\omega \mapsto \omega^{-1}$ which maintains this property, the proof still holds. Applying this map will yield the rearrangement in the problem statement.

In this solution, ω will be used to refer to the first n^{th} root of unity and α will be used to refer to the first $(n/2)^{\text{th}}$ root of unity.

First we will show that the upper left quarter of the matrix is equal to $M_{n/2}(\omega^{-1})$. The entry at coordinate (j, k) of this quarter is simply ω^{-2jk} (because of the rearrangement of the columns). But as we have seen (see the discussion section for the second week for instance), since ω is the first n^{th} root of unity and n is even, ω^2 is the first $(n/2)^{\text{th}}$ root of unity. Thus this entry is simply α^{-jk}, exactly the entry at coordinate (j, k) of $M_{n/2}(\omega^{-1})$.

Now observe that the entry in coordinate (j, k) of the bottom left quarter of the matrix is simply

$$\omega^{-(n/2+j)(2k)} = \omega^{-(nk+2jk)} = \omega^{-nk}\omega^{-2jk}$$

And since $\omega^n = 1$, this is just ω^{-2jk}, exactly as in the previous case.

Now we turn to the upper right quarter of the matrix. We want to show that the entry at coordinate (j, k) of this quarter is simply ω^{-j} times the entry at coordinate (j, k) of the upper left quarter, namely ω^{-2jk}. The entry at coordinate (j, k) of the upper right quarter is

$$\omega^{-j(2k+1)} = \omega^{-j}\omega^{-2jk}$$

as desired.

Finally, we need to show that the entry at coordinate (j, k) of the lower right quarter of the matrix is simply $-\omega^{-j}$ times the entry at coordinate (j, k) of the upper left quarter, namely ω^{-2jk}. The entry in question is

$$\omega^{-(j+n/2)(2k+1)} = \omega^{-2jk-nk-j-n/2} = \omega^{-n/2}\omega^{-j}\omega^{-nk}\omega^{-2jk}$$

Now note that $(\omega^{-n/2})^2 = 1$ but $\omega^{-n/2} \neq 1$ so it must be -1 and, as before, $\omega^{-nk} = 1$. So the expression above simplifies to $-\omega^{-j}\omega^{-2jk}$.

Comment. In general, if x is a complex number and n and m are integers then $(x^n)^{1/m} \neq (x^{1/m})^n$ and so the expression $x^{n/m}$ is not really well defined. As a simple example of this, take $n = 2, m = 2$ and $x = -1$. Then $((-1)^2)^{1/2} = 1^{1/2} = 1$ but $((-1)^{1/2})^2 = i^2 = -1$. In light of this, in might seem worrisome that we were so cavalier with our $n/2$'s above. The reason that this is not a problem is that $n/2$ is an integer, and raising complex numbers to integer powers is much better behaved operation than raising them to noninteger powers.[1]

[1] As we saw in class, the result from part (c) means that we can efficiently multiply vectors by the matrix $M_n(\omega^{-1})$.

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: We will use the $\Theta(n \log n)$-time FFT-based cross-correlation algorithm from lecture.

Algorithm description

Construct two vectors: vector $\vec{p} = [0, \dots, 0, b_0, \dots, b_{n-1}]$ is the "padded" vector which has length $2n$ and consists of every bit in b individually, preceded by n zeros; vector $\vec{f} = [(1 - b_0), \dots, (1 - b_{n-1})]$ is the "flipped" vector which consists of every bit in b, flipped (replace 0 with 1 and 1 with 0).

Run the cross-correlation algorithm on these vectors. The dot product between f and $p[j : n+j]$ is the number of $(n-j)$-inversions, which we can report for j from 1 to $n-1$. (Note that the first and last dot products will be ignored, since there are no 0- or n-inversions.)

Proof of correctness

There is a k-inversion starting at index i if and only if $b_i = 1$ and $b_{i+k} = 0$; which in turn is true if and only if $(b_i)(1 - b_{i+k}) = 1$. Consider our dot product $f \cdot p[j : n+j] = \sum_{i=0}^{n-1} f_i p_{i+j}$; but since $p_{i+j} = 0$ whenever $i + j < n$, this is $\sum_{i=n-j}^{n-1} f_i p_{i+j} = \sum_{i=0}^{j-1} f_{i+n-j} p_{n+i}$ which, by the definition of f and p, is $\sum_{i=0}^{j-1} (1 - b_{i+n-j}) b_i$. But by the first sentence, each term of this sum is 1 if there is a $(n-j)$-inversion starting at i, and 0 otherwise, so this sum is the number of $(n-j)$-inversions.

Since we know from lecture that the cross-correlation algorithm computes these products correctly, we can conclude that our whole algorithm is correct.

Runtime

Constructing our two vectors p and f takes a total of $\Theta(n)$ time, and we know from lecture that the cross-correlation algorithm takes $\Theta(n \log n)$ time. Arranging the results correctly may take $\Theta(n)$ time, or constant time, depending on implementation. So, since we do no other work, our algorithm as a whole takes $\Theta(n \log n)$ time.

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:

For each of $i \in \{0, 1, \ldots, m-n\}$ starting points in s, check if the substring $s[i : i+n-1]$ matches g. The check takes $O(n)$ time at each of $O(m)$ starting points, so the time complexity is $O(mn)$.

(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: Let g' be g with 0's replaced by -1's. Let $p_1(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$ where $a_d = g'(n-d-1)$ for all $d \in \{0, 1, \ldots, n-1\}$. Similarly, let $p_2(x) = b_0 + b_1 x + \cdots + b_{m-1}x^{m-1}$ where $b_d = s'(d)$ for all $d \in \{0, 1, \ldots, m-1\}$, where again s' is just s 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 g.

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 $c_{n-1+j} = \sum_{i=0}^{n-1} a_{n-1-i} b_{j+i} = \sum_{i=0}^{n-1} g'(i) s'(j+i)$ for any $j \in \{0, 1, \ldots, m-n\}$, which is exactly the dot product of the substring in s' starting at index j and the string g'. If these strings differ in at most k positions, then this dot product will be at least $n - 2k$. Thus all we need is to compute $p_3(x)$, and output all the j's between 0 and $m-n$ such that $c_{n-1+j} = n$.

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, $O(m \log m)$, is dominated by the FFT and inverse FFT steps, which each take $O(m \log m)$ time. The point-wise product and search for $c(i) = n$ each take $O(m)$ time.

(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: The idea is similar to the previous part. Let ω_α denote the α-th root of unity $e^{2\pi i/\alpha}$. We repeat the algorithm from the previous part, except we define g' to be a vector where $g'(i) = \omega_\alpha^j$ if $g(i) = c_j$. Similarly, we define s' to be the vector where $s'(i) = \omega_\alpha^{-j}$ if $s(i) = c_j$. This gives us that $g'(i)s'(i) = 1$ if and only if $g(i) = s(i)$.

Now if we look at the dot product $\sum_{i=0}^{n-1} g'(i)s'(j+i)$, this dot product can only be n if the length n substring starting at index j of s matches g exactly. To see this: If $g(i) \neq s(j+i)$, then $g'(i)s'(i)$ is a α-th root of unity that isn't 1. The only way to add up n α-th roots of unity with sum n is if all these roots of unity are 1 (since every other root of unity has real part less than 1). So, once we compute p_3, we just return all j such that $c_{n-1+j} = n$.

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.

Hint 1: elements can be encoded using exponents!

Hint 2: if stuck, refer to discussion Q3.

Solution:
Main idea Exponentiation converts multiplication to addition. For example, observe $x^3 * x^2 = x^{2+3} = x^5$. This gives us the idea to represent the lists as polynomials, with the elements in the lists in the exponents (This is a very common trick that you should remember!). For example, we can represent the array $[1, 3]$ as $x^1 + x^3$. If we multiply this with the polynomial for $[2, 4]$, $x^2 + x^4$, we get the polynomial $x^3 + 2x^5 + x^7$. Notice that $3, 5, 7$ correspond to the possible sums of an element in $[1, 3]$ and an element in $[2, 4]$.

More formally, 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.

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 [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.

fft

September 13, 2023

1 FFT and Polynomial Multiplication

Here, you will implement FFT, and then use FFT as a black box to some applied problems.

Note that the functions you write build upon one another. Therefore, it will be necessarily to correctly solve previous problems in the notebook to solve later problems.

[ ]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

[ ]:
```
import otter
import scipy

assert (
    otter.__version__ >= "4.4.1" and scipy.__version__ >= "1.10.0"
), "Please reinstall the requirements and restart your kernel."

grader = otter.Notebook("fft.ipynb")
import numpy as np
from numpy.random import randint
from time import time
import tqdm
import scipy
import matplotlib.pyplot as plt
import numpy.random as random

rng_seed = 42
```

If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

If you're running locally:

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

1.0.1 Q1.1) Roots of Unity

n-th roots of unity are defined as complex numbers where $z^n = 1$.
Another equivalent definition is the numbers $e^{\frac{2\pi ik}{n}}$ where $k = 0, 1, \cdots, n-1$.
First, write a function that, given n, outputs all n-th roots of unity. Note that n does not have to be even.

You can alternatively calculate it in form $a + bi$, but it is not required.

Hints: 1. There are multiple ways to calculate roots of unity. You may find the following constants and functions useful. Depending on how you choose to complete this part, you may not need all of them: * `np.e` * `np.pi` * `np.exp()` * `np.zeros()` * `np.roots()` 2. If you're unsure of what a particular function does, you can always consult the appropriate Numpy/Scipy/Python documentation or type `?` followed by a function name in a notebook cell to get more information. For example, `?np.exp` will give you more information about the `np.exp()` function.

3. Python supports arithmetic with complex numbers. You can enter complex literals in the form `a + bj` where `a` and `b` are integers. For example, you can have `2 + 3j`. If you want to cast a real number to a complex type, you need to write something like `1 + 0j` or `complex(1)`.

4. Make sure you return the roots in the correct order: if the principal root is ω, you should return $[1, \omega, \omega^2, \omega^3, \ldots]$.

```
[ ]: def roots_of_unities(n):
         """
         args:
             n:int = n describes which root of unity to return
         return:
             a list of n complex numbers containing the n-th roots of unity [w^0,␣
      ↪w^1, w^2, ..., w^{n-1}]
         """
         # BEGIN SOLUTION
         # solution based on finding the primitive root of unity then raising it to␣
      ↪different powers
         principal_root = np.exp(2 * np.pi * 1j / n)
         roots = principal_root ** np.arange(n)
         return roots
         """
         # Alternate solution based on finding the roots of the polynomial x^n - 1 =␣
      ↪0
         poly = np.zeros(n + 1)
         poly[0] = 1
         poly[-1] = -1
         roots = np.roots(poly)
         # sort the roots so that they are in the correct order
         sorted_roots = sorted(roots, key=lambda x: np.angle(x) % (2 * np.pi))
         return sorted_roots
         """
         # END SOLUTION
```

To make sure your helper function is correct, we can draw the resulting values on the unit circle. Run the following cell and make sure the output is as you expect it to be:

Points: 0.5

```
[ ]: N = 16  # feel free to change this value and observe what happens
     roots = roots_of_unities(N)

     # Plot
     f, ax = plt.subplots()
     f.set_figwidth(4)
     f.set_figheight(4)
     plt.scatter([r.real for r in roots], [r.imag for r in roots])
     ax.spines["left"].set_position("zero")
     ax.spines["right"].set_color("none")
     ax.yaxis.tick_left()
     ax.spines["bottom"].set_position("zero")
     ax.spines["top"].set_color("none")
     ax.set_xlim([-1.2, 1.2])
     ax.set_ylim([-1.2, 1.2])
     ax.set_xticks([-1, -0.5, 0.5, 1])
     ax.set_yticks([-1, -0.5, 0.5, 1])
     ax.xaxis.tick_bottom()
```

```
[ ]: grader.check("q1.1")
```

1.1 Discrete Fourier Transform

In CS170, we define the process that transforms polynomials from coefficient representation to value representation (at next 2^n-th roots of unities) as "Discrete Fourier Transform".
For example, for polynomial $P(x) = 1 + x + x^2 + x^3$, its value representation would be $P(1) = 4, P(i) = 0, P(-1) = 0, P(-i) = 0$.

1.1.1 Q1.2) Naive DFT

Write an naive algorithm that calculated DFT.

Hints: 1. The input list is a list of coefficients but you might find `np.poly1d` useful. https://numpy.org/doc/stable/reference/generated/numpy.poly1d.html 2. You might find the following function useful.

Points: 1

```
[ ]: # Utility function, returns next 2^n
     hyperceil = lambda x: int(2 ** np.ceil(np.log2(x)))
```

```
[ ]: def dft_naive(coeffs):
         """
         args:
```

```
        coeffs:np.array = list of numbers representing the coefficients of a␣
↪polynomial where coeffs[i]
                        is the coeffiecient of the term x^i
    return:
        List containing the results of evaluating the polynomial at the roots␣
↪of unity. Can be either a list or np.array
        [P(w_0), P(w_1), P(w_2), ...]
    """
    # BEGIN SOLUTION
    n = hyperceil(len(coeffs))
    p = np.poly1d(coeffs[::-1])
    return [p(omega) for omega in roots_of_unities(n)]
    # END SOLUTION
```

[ ]:
```
grader.check("q1.2")
```

1.1.2 Q1.3) FFT

FFT is an algorithm that calculates DFT in $\mathcal{O}(n \log n)$ time.

Now, you'll implement FFT by itself. The way it is defined here, this takes in the coefficients of a polynomial as input, evaluates it on the n-th roots of unity, and returns the list of these values. For instance, calling

$$FFT([1,2,3,0],\ [1,i,-1,-i])$$

should evaluate the polynomial $1+2x+3x^2$ on the points $1, i, -1, -i$, returning

$$[6,\ -2+2i,\ 2,\ -2-2i]$$

Recall that to do this efficiently for a polynomial

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1}$$

we define two other polynomials E and O, containing the coefficients of the even and odd degree terms respectively,

$$E(x) := a_0 + a_2 x + \cdots + a_{n-2}x^{n/2-1}, \qquad O(x) := a_1 + a_3 x + \cdots = a_{n-1}x^{n/2-1}$$

which satisfy the relation

$$P(x) = E(x^2) + xO(x^2)$$

We recursively run FFT on E and O, evaluating them on the $n/2$-th roots of unity, then use these values to evaluate P on the n-th roots of unity, via the above relation.

Implement this procedure below, where “coeffs” are the coefficients of the polynomial we want to evaluate (with the coefficient of x^i at index i), and where

$$\text{roots} = [1, \omega, \omega^2, \dots, \omega^{n-1}]$$

for some primitive n-th root of unity ω where n is a power of 2. (Note: Arithmetic operations on complex numbers in python work just like they do for floats or ints. Also, you can use A[::k] to take every k-th element of an array A)

Debugging tip: 1. Remember that your inputs are `np.arrays`. Unless you specifically cast to `List`, you can't append or concatenate `np.arrays` in the same way as Python lists. You can use `np.append` to append to an array, and `np.concatenate` to concatenate two `np.arrays`.

```
[ ]: def fft(coeffs, roots):
         """
         args:
             coeffs (np.array): array of numbers representing the coefficients of a␣
      ↪polynomial where coeffs[i]
                         is the coeffiecient of the term x^i
             roots (np.array): array containing the roots of unity [w_0, w_1, w_2, ..
      ↪., w_{n-1}]
         return:
             List containing the results of evaluating the polynomial at the roots␣
      ↪of unity
             [P(w_0), P(w_1), P(w_2), ...]
         """
         n = len(coeffs)
         assert n == hyperceil(n)
         # BEGIN SOLUTION
         if n == 1:
             return coeffs[:]
         e_coeff = coeffs[0::2]
         o_coeff = coeffs[1::2]
         e = fft(e_coeff, roots[::2])
         o = fft(o_coeff, roots[::2])

         res = np.zeros(n, dtype=np.complex_)
         for i in range(n // 2):
             res[i] = e[i] + roots[i] * o[i]
             res[i + n // 2] = e[i] + roots[i + n // 2] * o[i]
         return res
         # END SOLUTION
```

1.1.3 Testing

Here's a sanity check to test your implementation. Calling $FFT([1,2,3,0],[1,1j,-1,-1j])$ should output $[6,\ -2+2j,\ 2,\ -2-2j]$ (Python uses j for the imaginary unit instead of i.)

Feel free to add your own testing cells here as well!

```
[ ]: expected = [6, -2 + 2j, 2, -2 - 2j]
     actual = fft([1, 2, 3, 0], [1, 1j, -1, -1j])
     assert np.allclose(expected, actual), f"expected: {expected}\n actual: {actual}"
     print("success")
```

If you correctly implemented the FFT algorithm, and aren't naively evaluating on each point, the result should only work when the `roots` parameter are the roots of unity. Therefore, the call $FFT([1, 2, 3, 0], [1, 2, 3, 4])$ should NOT return the values of $1 + 2x + 3x^2$ on the inputs $[1, 2, 3, 4]$ (which would be $[6, 17, 34, 57]$):

Points: 3

```
[ ]: not_expected = [6, 17, 34, 57]
     actual = fft([1, 2, 3, 0], [1, 2, 3, 4])
     assert not np.allclose(expected, actual), f"NOT expected: {expected}\n actual:␣
      ↪{actual}"
     print("success")
```

```
[ ]: grader.check("q1.3")
```

1.1.4 Q1.4) Inverse FFT

Now that you know your FFT is correct, implement IFFT.

IFFT can be implemented in less than 3 lines, you can check the bottom of DPV page 75 to see how it is finished.

Hints: 1. Python supports computing the inverse of a complex number by raising it to the power of -1. In other words, suppose `x` is a complex number, the inverse is `x**(-1)`. 2. We pass in `roots` as a parameter for a reason, make sure you use it instead of hardcoding your roots of unity in both your `fft` and `ifft` functions.

Points: 1.5

```
[ ]: def ifft(vals, roots):
         """
         args:
             val (np.array): numpy array containing the results of evaluating the␣
      ↪polynomial at the roots of unity
                         [P(w_0), P(w_1), P(w_2), ...]

             roots (np.array): numpy array containing the roots of unity [w_0, w_1,␣
      ↪w_2, ..., w_{n-1}]

         return:
             List containing the results of evaluating the polynomial at the roots␣
      ↪of unity. Can be a Python list
                 or numpy array.
```

```
        [P(w_0), P(w_1), P(w_2), ...]
    """
    n = len(vals)
    assert n == hyperceil(n)
    # BEGIN SOLUTION
    iroots = np.array(roots) ** -1
    return np.array(fft(vals, iroots) / n)
    # END SOLUTION
```

[ ]:
```
grader.check("q1.4")
```

1.2 Q1.5) Polynomial Multiplication (Optional)

1.2.1 The following subpart is optional and ungraded, but you may find it helpful for understanding how FFT is used to perform polynomial multiplication.

Now you'll implement polynomial multiplication, using your FFT function as a black box. Recall that to do this, we first run FFT on the coefficients of each polynomial to evaluate them on the n-th roots of unity for a sufficiently large power of 2, which we call n. We then multiply these values together pointwise, and finally run the inverse FFT on these values to convert back to coefficient form, obtaining the coefficient of the product. To perform inverse FFT, we can simply run FFT, but with the roots of unity inverted, and divide by n at the end.

Note that FFT and IFFT only accepts polynomials of degree $2^n - 1$, so you would need to pad coefficients to the next `hyperceil(n)`.

You may find defining the `pad` function to be helpful but are not required to do so.

Points: 0

[ ]:
```
def pad(coeffs, to):
    """
    args:
        coeffs:List[] = list of numbers representing the coefficients of a␣
 ↪polynomial where coeffs[i]
                    is the coeffiecient of the term x^i
        to:int = the final length coeffs should be after padding

    return:
        List of coefficients zero padded to length 'to'
    """
    # BEGIN SOLUTION
    return np.pad(coeffs, (0, to - len(coeffs)), "constant", constant_values=0)
    # END SOLUTION


def poly_multiply(coeffs1, coeffs2):
    """
    args:
        coeffs1:List[] = list of numbers representing the coefficients of a␣
↪polynomial where coeffs[i]
                    is the coeffiecient of the term x^i
        coeffs2:List[] = list of numbers representing the coefficients of a␣
↪polynomial where coeffs[i]
                    is the coeffiecient of the term x^i

    return:
        List of coefficients corresponding to the product polynomial of the two␣
↪inputs.
    """
    # BEGIN SOLUTION
    n = hyperceil(len(coeffs1) + len(coeffs2) - 1)
    v1 = fft(pad(coeffs1, n), roots_of_unities(n))
    v2 = fft(pad(coeffs2, n), roots_of_unities(n))
    v = v1 * v2  # only works with numpy arrays
    return ifft(v, roots_of_unities(n))
    # END SOLUTION
```

1.2.2 Testing

[ ]:
```
def round_complex_to_int(lst):
    return [round(x.real) for x in lst]


def zero_pop(lst):
    return np.trim_zeros(lst, "b")
```

Here are a couple sanity checks for your solution.

[ ]:
```
expected = [4, 13, 22, 15]
actual = round_complex_to_int(poly_multiply([1, 2, 3], [4, 5]))
print("expected: {}".format(expected))
print("actual:   {}\n".format(actual))

expected = [4, 13, 28, 27, 18, 0, 0, 0]
actual = round_complex_to_int(poly_multiply([1, 2, 3], [4, 5, 6]))
print("expected: {}".format(expected))
print("actual:   {}".format(actual))
```

One quirk of FFT is that we use complex numbers to multiply integer polynomials, so this leads to floating point errors. You can see this with the following call, which will probably not return exact integer values (unless you did something in your implementation to handle this):

[ ]:
```
result = poly_multiply([1, 2, 3], [4, 5, 6])
result
```

Therefore, if we're only interested in integers, like many of the homework problems, we have to round the result:

```
[ ]: result = round_complex_to_int(result)
     result
```

However, there might still be trailing zeros we have to remove:

```
[ ]: zero_pop(result)
```

This (hopefully) gives us exactly what we would have gotten by multiplying the polynomials normally, $[4, 13, 28, 27, 18]$.

1.2.3 Runtime Comparison

Here, we compare the runtime of polynomial multiplication with FFT to the naive algorithm.

```
[ ]: def poly_multiply_naive(coeffs1, coeffs2):
         n1, n2 = len(coeffs1), len(coeffs2)
         n = n1 + n2 - 1
         prod_coeffs = [0] * n
         for deg in range(n):
             for i in range(max(0, deg + 1 - n2), min(n1, deg + 1)):
                 prod_coeffs[deg] += coeffs1[i] * coeffs2[deg - i]
         return prod_coeffs
```

```
[ ]: poly_multiply_naive([3, 4, 5], [7, 2, 7, 4])
```

Running the following cell, you should see FFT perform similarly to or worse than the naive algorithm on small inputs, but perform significantly better once inputs are sufficiently large, which should be apparent by how long you have to wait for the naive algorithm to finish on the largest input (you might need to run the next cell twice to see the plot for some reason):

```
[ ]: def rand_ints(lo, hi, length):
         ints = list(randint(lo, hi, length))
         ints = [int(x) for x in ints]
         return ints


     def record(array, value, name):
         array.append(value)
         print("%s%f" % (name, value))


     fft_times = []
     naive_times = []
     speed_ups = []

     for i in range(5):
```

```
    n = 10**i
    print("\nsize: %d" % n)
    poly1 = rand_ints(1, 100, n)
    poly2 = rand_ints(1, 100, n)
    time1 = time()
    fft_res = poly_multiply(poly1, poly2)
    fft_res = zero_pop(round_complex_to_int(fft_res))
    time2 = time()
    fft_time = time2 - time1
    record(fft_times, fft_time, "FFT time:   ")
    naive_res = poly_multiply_naive(poly1, poly2)
    time3 = time()
    naive_time = time3 - time2
    record(naive_times, naive_time, "naive time: ")
    assert fft_res == naive_res
    speed_up = naive_time / fft_time
    record(speed_ups, speed_up, "speed up: ")

plt.plot(fft_times, label="FFT")
plt.plot(naive_times, label="Naive")
plt.xlabel("Log Input Size")
plt.ylabel("Run Time (seconds)")
plt.legend(loc="upper left")
plt.title("Polynomial Multiplication Runtime")

plt.figure()
plt.plot(speed_ups)
plt.xlabel("Log Input Size")
plt.ylabel("Speedup")
plt.title("FFT Polynomial Multiplication Speedup")
```

1.3 Q2) Evaluating Predictions

Now, let's see how we can apply FFT to speed up a common signal-processing task.

Suppose we have access to the temperature for the past n days, where $y_0, y_1, \dots, y_{n-1}$ where all $y_i \in \mathbb{R}$. You are given fixed coefficients $c_0, \dots, c_{n-2}$, which give the following prediction for day $t \geq 1$:

$$p_t = \sum_{k=0}^{t-1} c_k y_{t-1-k}$$

Note that we only compute p_t for $t > 0$.

You would like to evaluate the accuracy of this prediction on the dataset by computing the *mean squared error*, given by

$$\frac{1}{n-1} \sum_{t=1}^{n-1} (p_t - y_t)^2$$

Find an $\mathcal{O}(n \log n)$ time algorithm to compute the mean squared error, given dataset $y_0, y_1, \ldots, y_{n-1}$ and coefficients $c_0, \ldots, c_{n-2}$.

You may use any functions that you implemented previously, and you may also find it helpful (but are not required) to implement and use the `pad` function defined below. For this subpart, you may also use numpy or scipy FFT and iFFT.

Hints:

1. Try walking through the computation of p_t by hand. If we want to cast this as polynomial multiplication, do we need to modify the coefficients $c_0, \ldots, c_{n-2}$ in any way?
2. What about the dataset $y_0, y_1, \ldots, y_{n-1}$?

Points: 5

```
[ ]: def pad(coeffs, to):
         """
         args:
             coeffs (np.array): list of numbers representing the coefficients of a␣
      ↪polynomial where coeffs[i]
                         is the coeffiecient of the term x^i
             to (int): the final length coeffs should be after padding

         return:
             List of coefficients zero padded to length 'to'
         """
         # BEGIN SOLUTION
         return np.pad(coeffs, (0, to - len(coeffs)), "constant", constant_values=0)
         # END SOLUTION


     def evaluate_predictions(coeffs, labels):
         """Given coefficients and labels, generate predictions for each day and␣
      ↪return the MSE between the
             predictions and the labels. See the formulas in the problem for␣
      ↪specific details on how these
             are defined.

         Args:
             coeffs (np.array): Fixed coefficients c_0, c_1, ..., c_{n-2} which are␣
      ↪used to generate the predictions
             labels (np.array): List of temperatures for the past n days, y_0, y_1, .
      ↪.., y_{n-1}
         Returns:
             int: MSE between the predictions and the labels over all n-1 days.
         """
         # BEGIN SOLUTION
         # computing p_t for all values of t would usually take O(n^2) time,
         # but we can reduce it to O(n log n) using FFT.
```

```
    # prepend coeffs with 0 to shift everything
    coeffs_shift = np.insert(coeffs, 0, 0)
    n = hyperceil(len(coeffs_shift) + len(labels) - 1)
    C = fft(pad(coeffs_shift, n), roots_of_unities(n))
    Y = fft(pad(labels, n), roots_of_unities(n))
    P = C * Y
    p = ifft(P, roots_of_unities(n))[1 : len(labels)]

    # now that p is computed, we can compute the MSE in linear time.
    return np.mean((p - labels[1:]) ** 2)
    # END SOLUTION
```

[ ]:
```
grader.check("q2")
```

1.4 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

[ ]:
```
grader.export(pdf=False, force_save=True, run_tests=True)
```