Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Complex numbers review

A *complex number* is a number that can be written in the rectangular form $a + bi$ (i is the imaginary unit, with $i^2 = -1$). The following famous equation (*Euler's formula*) relates the polar form of complex numbers to the rectangular form:

$$re^{i\theta} = r(\cos\theta + i\sin\theta)$$

In polar form, $r \geq 0$ represents the distance of the complex number from 0, and θ represents its angle. Note that since $\sin(\theta) = \sin(\theta + 2\pi), \cos(\theta) = \cos(\theta + 2\pi)$, we have $re^{i\theta} = re^{i(\theta+2\pi)}$ for any r, θ.

The n-th *roots of unity* are the n complex numbers satisfying $\omega^n = 1$. They are given by

$$\omega_k = e^{2\pi ik/n}, \qquad k = 0, 1, 2, \ldots, n-1$$

(a) Let $x = e^{2\pi i 3/10}, y = e^{2\pi i 5/10}$ which are two 10-th roots of unity. Compute the product $x \cdot y$. Is this an n-th root of unity for some n? Is it a 10-th root of unity?

What happens if $x = e^{2\pi i 6/10}, y = e^{2\pi i 7/10}$?

For all your answers, simplify if possible.

(b) Show that for any n-th root of unity $\omega \neq 1$, $\sum_{k=0}^{n-1} \omega^k = 0$, when $n > 1$.

Hint: Use the formula for the sum of a geometric series $\sum_{k=0}^{n} \alpha^k = \frac{\alpha^{n+1}-1}{\alpha-1}$. It works for complex numbers too!

(c) (i) Find all ω such that $\omega^2 = -1$.

(ii) Find all ω such that $\omega^4 = -1$.

2 FFT intro

We will use ω_n to denote the first n-th root of unity $\omega_n = e^{2\pi i/n}$. The most important fact about roots of unity for our purposes is that the squares of the $2n$-th roots of unity *are* the n-th roots of unity. For example, if we square the primitive 8th root of unity, $\omega_8 = e^{i\cdot\frac{2\pi}{8}}$, we get the primitive 4th root of unity:

$$\omega_8^2 = e^{i\cdot\frac{2\pi}{8}\cdot 2} = e^{i\cdot\frac{2\pi}{4}} = \omega_4$$

Discrete Fourier Transform (DFT). The *discrete Fourier transform* transforms a vector $[p_0, p_1, \ldots, p_{n-1}]$ which contains the coefficients of some polynomial $P(x) = p_0 + p_1 x + \ldots + p_{n-1}x^{n-1}$ and turns it into the vector $[P(1), P(\omega_n), \ldots, P(\omega_n^{n-1})]$ which contains the evaluations of $P(x)$ at the n-th roots of unity. The transformation is realized by multiplying the input vector with the DFT matrix as follows:

$$\begin{bmatrix} P(1) \\ P(\omega_n) \\ P(\omega_n^2) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \ldots & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \ldots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \ldots & \omega_n^{(n-1)(n-1)} \end{bmatrix}}_{M} \cdot \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

So DFT can be performed in $O(n^2)$ time using a naive matrix-vector multiplication algorithm. We can also observe that the **Inverse DFT (IDFT)** matrix can be expressed as follows:

$$M^{-1} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \ldots & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \ldots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \ldots & \omega_n^{(n-1)(n-1)} \end{bmatrix}^{-1} = \frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \ldots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \ldots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \ldots & \omega_n^{-(n-1)(n-1)} \end{bmatrix}$$

(You should verify this on your own!)

So given the evaluation vector $[P(1), P(\omega_n), P(\omega_n^2), \ldots, P(\omega_n^{n-1})]$, we can reverse the DFT by computing the IDFT via the following matrix multiplication:

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix} = \underbrace{\frac{1}{n}\begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \ldots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-4} & \ldots & \omega_n^{-2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \ldots & \omega_n^{-(n-1)(n-1)} \end{bmatrix}}_{M^{-1}} \cdot \begin{bmatrix} P(1) \\ P(\omega_n) \\ P(\omega_n^2) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix}$$

This can be done in $O(n^2)$ time using a naive matrix-vector multiplication algorithm.

Fast Fourier Transform! The *Fast Fourier Transform* $\text{FFT}(p, n)$ is an algorithm that takes in a coefficient vector $p = [p_0, p_1, \ldots, p_{n-1}]$ and corresponding order n, and performs DFT in only $\mathcal{O}(n \log n)$ time. Note that in order to make FFT work, p is always padded with zeroes so that n is a power of 2.

If we let $P_E(x) = p_0 + p_2 x + \dots p_{n-2} x^{n/2-1}$ and $P_O(x) = p_1 + p_3 x + \dots p_{n-1} x^{n/2-1}$, then $P(x) = P_E(x^2) + x \cdot P_O(x^2)$, and thus $\text{FFT}(p, n)$ can be expressed as a divide-and-conquer algorithm shown below:

1. Compute $y_E = \text{FFT}(P_E, n/2)$ and $y_O = \text{FFT}(P_O, n/2)$.
2. For $j = 0, \dots, n-1$, use y_E, y_O to assign $P(\omega_n^j) \leftarrow P_E((\omega_n^j)^2) + \omega_n^j \cdot P_O((\omega_n^j)^2)$.
3. Return all the evaluations $P(\omega_n^j)$ for all $j = 0, \dots, n-1$.

You can implement this in practice by following the pseudocode described in Algorithm 1.

Algorithm 1 $\text{FFT}(p = [p_0, p_1, \dots, p_{n-1}], n)$

```
if n == 1 then
    return p
end if
y_E = FFT([p_0, p_2, ...], n/2), y_O = FFT([p_1, p_3, ...], n/2)
y = [0] · n
for j in range(n/2) do
    y[j] = y_E[j] + ω_n^j · y_O[j]
    y[j + n/2] = y_E[j] − ω_n^j · y_O[j]
end for
return y
```

(a) Let's first walk through how to compute the fourier transform of a polynomial using the DFT matrix. Consider the polynomial $P(x) = 1 + 3x + 4x^2 + 2x^3$.

(i) Compute the coefficient representation of P. In other words, find p.

(ii) Compute the DFT matrix for p.

(iii) Compute the fourier transform of p.

(b) Now that we've seen the naive method for computing the DFT, we explore how to implement FFT.

(i) Let $p = [p_0]$. What is $\text{FFT}(p, 1)$?

(ii) Use the FFT algorithm to compute $\text{FFT}([1, 4], 2)$ and $\text{FFT}([3, 2], 2)$.

(iii) Use your answers to the previous parts to compute $\text{FFT}([1, 3, 4, 2], 4)$.

(iv) Describe how to multiply two polynomials $P(x), Q(x)$ in coefficient form of degree at most d.

(v) Use the algorithm from the previous part to multiply the two polynomials $P(x) = 1 + 2x$ and $Q(x) = 3 - x$ in coefficient form.

3 Cartesian Sum

Let A and B be two sets of integers in the range 0 to $10n$. The *Cartesian sum* of A and B is defined as

$$A + B = \{a + b \mid a \in A, b \in B\}$$

i.e. all sums of an element from A and an element with B. For example, $\{1, 3\} + \{2, 4\} = \{3, 5, 7\}$.

Note that the values of $A + B$ are integers in the range 0 to $20n$. Design an algorithm that finds the elements of $A + B$ in $\mathcal{O}(n \log n)$ time. This algorithm should also tell you for each $c \in A + B$, *how many* pairs $a \in A, b \in B$ there are such that $a + b = c$.

Hint: Building off the example $\{1, 3\} + \{2, 4\} = \{3, 5, 7\}$, notice that $(x^1 + x^3) \cdot (x^2 + x^4) = x^3 + 2x^5 + x^7$. How are the coefficients and exponents related?

4 Cubed Roots of Unity

(a) We explore how we can obtain the 3^{rd} roots of unity from the 9^{th} roots of unity. The first element in each row below is a 3^{rd} root unity (e.g. $\omega_3^0 = e^{i\frac{2\pi}{3}\cdot 0}$). Next to each of these 3^{rd} roots, write down the three corresponding 9^{th} roots that cube to it (e.g. $(\omega_9^0)^3 = \omega_3^0$). We provide the first two values for you.

$\omega_3^{\mathbf{0}}$	ω_9^0,	ω_9^3,	,
$\omega_3^{\mathbf{1}}$	,	,	,
$\omega_3^{\mathbf{2}}$	,	,	,

(b) You want to run FFT on a degree-8 polynomial, but you don't like having to pad it with 0s to make (degree+1) a power of 2. Instead, you realize that 9 is a power of 3, and you decide to work directly with 9th roots of unity and use the fact proven in part (a). Say that your polynomial looks like $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_8x^8$. Describe a way to split $P(x)$ into three pieces (instead of two) so that you can make an FFT-like divide-and-conquer algorithm.

(c) What is the runtime of FFT when we divide the polynomial into three pieces instead of two?