# 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.**

### 1.0.1 If you're using Datahub:

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

### 1.0.2 If you're running locally:

You'll need to perform some extra setup. #### First-time setup * Install Anaconda following the instructions here: https://www.anaconda.com/products/distribution * Create a conda environment: `conda create -n cs170 python=3.8` * Activate the environment: `conda activate cs170` * See for more details on creating conda environments https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html * Install pip: `conda install pip` * Install jupyter: `conda install jupyter`

**Every time you want to work**

- 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**

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

```
In [ ]: 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
```

### 1.0.3 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 $\omega$, you should return $[1, \omega, \omega^2, \omega^3, \ldots]$.

```
In [ ]: 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^{
            """
            # 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

```
In [ ]: 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()
```

```
In [ ]: 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. Be mindful of the order that we list coefficients in. 2. You might find the following function useful.

*Points:* 1

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

```
In [ ]: def dft_naive(coeffs):
            """
            args:
                coeffs:np.array = list of numbers representing the coefficients of a polynomial where c
                            is the coeffiecient of the term x^i
            return:
                List containing the results of evaluating the polynomial at the roots of unity. Can be
                [P(w_0), P(w_1), P(w_2), ...]
            """
            # BEGIN SOLUTION
            n = hyperceil(len(coeffs))
            return [sum(omega**i * c_n for i, c_n in enumerate(coeffs)) for omega in roots_of_unities(n
            """Alternate solution using np.poly1d
                n = hyperceil(len(coeffs))
                p = np.poly1d(coeffs[::-1])
                return [sum(omega**i * c_n for i, c_n in enumerate(coeffs)) for omega in roots_of_uniti
            """
            # END SOLUTION
```

```
In [ ]: 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

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

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

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

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 $\omega$ 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`.

```
In [ ]: def fft(coeffs, roots):
            """
            args:
                coeffs (np.array): array of numbers representing the coefficients of a polynomial where
                            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!

```
In [ ]: 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

```
In [ ]: 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")
```

```
In [ ]: 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

```
In [ ]: def ifft(vals, roots):
            """
            args:
                val (np.array): numpy array containing the results of evaluating the polynomial at the
                            [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
                    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
```

```
In [ ]: 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

```
In [ ]: def pad(coeffs, to):
            """
            args:
                coeffs:List[] = list of numbers representing the coefficients of a polynomial where coe
                            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 co
                            is the coeffiecient of the term x^i
                coeffs2:List[] = list of numbers representing the coefficients of a polynomial where co
                            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

```
In [ ]: 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.

```
In [ ]: 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):

```
In [ ]: 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:

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

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

```
In [ ]: 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.

```
In [ ]: 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
```

```
In [ ]: 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):

```
In [ ]: 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 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.

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