CS 170 Homework 3

Due **2/09/2021, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, 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 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.

5 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, $O(m \log m)$, is dominated by the FFT and inverse FFT steps, which each take $O(m \log m)$ time. The point-wise product and search for $c(i) \geq n - 2k$ 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.

6 FFT Coding

This semester, we are trying something new: questions which involve coding.

This link will take you to a python notebook, hosted on the Berkeley datahub, in which you will implement three functions (`FFT`, `calc_nth_root`, and `poly_multiply`) to investigate how FFT works. Once you have finished, download a PDF of your completed notebook via File → Download as → PDF via HTML, and append the downloaded pdf to the rest of your homework submission. Be careful when selecting pages on gradescope.

Note: Datahub does not guarantee 100% reliability when you save your notebook, and recommends downloading a local copy occasionally to backup progress (via File → Download as → Notebook (.ipynb)).

FFT and Polynomial Multiplication

Here, you will implement FFT, and then implement polynomial multiplication, using FFT as a black box.

In [1]:
```
import math
import cmath
from numpy.random import randint
from time import time
import matplotlib.pyplot as plt
```

FFT

First, 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

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

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

$$\mathrm{roots} = [1, \omega, \omega^2, \ldots, \omega^{n-1}]$$

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 $\mathrm{A[::k]}$ to take every k-th element of an array A)

In [2]:
```
def FFT(coeffs, roots):
    # need len(roots) to be a power of two
    # don't worry about this expression if you haven't seen it before
    assert (len(roots) & (len(roots)-1) == 0) and len(roots) != 0

    # base case
    if len(coeffs) == 0:
        return [0]*len(roots)
    elif len(roots) == 1:
        return coeffs

    # recursive calls
    even_evals = FFT(coeffs[::2], roots[::2])
    odd_evals = FFT(coeffs[1::2], roots[::2])
    # you can use additional lines here, or just a list comprehension after
    result = [even_evals[i] + roots[i] * odd_evals[i] for i in range(len(root
s)//2)] + [even_evals[i] - roots[i] * odd_evals[i] for i in range(len(roots)
//2)]

    return result
```

Testing

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

In [3]:
```
expected = [6, -2+2j, 2, -2-2j]
actual = FFT([1, 2, 3], [1, 1j, -1, -1j])
print("expected: {}".format(expected))
print("actual:   {}".format(actual))
```

```
expected: [6, (-2+2j), 2, (-2-2j)]
actual:   [6, (-2+2j), 2, (-2-2j)]
```

If you are correctly implemented the FFT algorithm and not just naively evaluating on each point, it should rely on the points being roots of unity. Therefore, the call $FFT([1, 2, 3], [1, 2, 3, 4])$ should NOT return the values of $1 + 2x + 3x^2$ on the inputs $[1, 2, 3, 4]$ (which would be $[6, 17, 34, 57]$):

In [4]:
```
not_expected = [6, 17, 34, 57]
actual = FFT([1, 2, 3], [1, 2, 3, 4])
print("NOT expected: {}".format(not_expected))
print("actual:       {}".format(actual))
```

```
NOT expected: [6, 17, 34, 57]
actual:       [6, 2, 2, -6]
```

Polynomial Multiplication

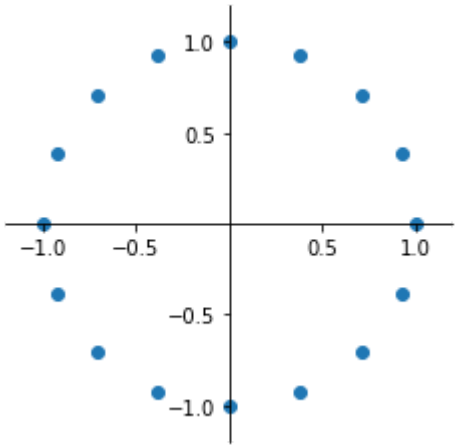
Now you'll implement polynomial multiplication, using your FFT function as a black box. Recall that to do this, we first run FFT on the coefficients of each polynomial to evaluate them on the n-th roots of unity for a sufficiently large power of 2, which we call n. We then multiply these values together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

We will need a couple helper functions to do this. The function `next_power_of_2`, which has been provided, takes in a parameter k and returns the smallest power of 2 that is $\geq k$. You will now write an additional function `calc_nth_root` that takes in a parameter n and returns the first n-th root of unity ω_n (Hint: It will be helpful to look at the definition of the n-th roots of unity for this question). Finally, `calc_all_nth_roots` takes in n and returns the list $[1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}]$.

In [5]:
```
def next_power_of_2(k):
    ret = 1
    while ret < k:
        ret *= 2
    return ret

# note that the sine function is math.sin(x) and the cosine function is math.
cos(x)
def calc_nth_root(n):
    theta_n = 2 * math.pi / n
    a = math.cos(theta_n)
    b = math.sin(theta_n)
    return a + b * 1j

def calc_all_nth_roots(n):
    root = calc_nth_root(n)
    return [root**k for k in range(n)]
```

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:

In [6]:
```
N = 16 # feel free to change this value and observe what happens
roots = calc_all_nth_roots(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()
plt.show()
```



Now, we are ready to implement the polynomial multiplication. Do so below. (Hint: 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)`.)

In [16]:
```
def poly_multiply(coeffs1, coeffs2):
    target_degree = next_power_of_2(len(coeffs1) + len(coeffs2)-1) # note tha
t this can be just the sum too, if the function is called later
    roots = calc_all_nth_roots(target_degree)

    evals1 = FFT(coeffs1, roots)
    evals2 = FFT(coeffs2, roots)
    product_evals = [evals1[i] * evals2[i] for i in range(len(evals1))]

    inverse_roots = [x**(-1) for x in roots]
    res_coeffs = [x/target_degree for x in FFT(product_evals, inverse_roots)]
    return res_coeffs
```

Testing

In [17]:
```
def round_complex_to_int(lst):
    return [round(x.real) for x in lst]

def zero_pop(lst):
    while lst[-1] == 0:
        lst.pop()
```

Here are a couple sanity checks for your solution.

In [18]:
```
expected = [4, 13, 22, 15]
actual = round_complex_to_int(poly_multiply([1, 2, 3], [4, 5]))
print("expected: {}".format(expected))
print("actual:   {}".format(actual))
```

```
expected: [4, 13, 22, 15]
actual:   [4, 13, 22, 15]
```

In [19]:
```
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))
```

```
expected: [4, 13, 28, 27, 18, 0, 0, 0]
actual:   [4, 13, 28, 27, 18, 0, 0, 0]
```

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 [20]:
```
result = poly_multiply([1, 2, 3], [4, 5, 6])
result
```

Out[20]:
```
[(4+5.329070518200751e-15j),
 (13.000000000000002+0j),
 (28-8.881784197001252e-16j),
 (27-8.881784197001252e-16j),
 (18-3.552713678800501e-15j),
 (-1.7763568394002505e-15+0j),
 -8.881784197001252e-16j,
 8.881784197001252e-16j]
```

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

In [21]:
```
result = round_complex_to_int(result)
result
```

Out[21]:
```
[4, 13, 28, 27, 18, 0, 0, 0]
```

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

In [22]:
```
zero_pop(result)
result
```

Out[22]:
```
[4, 13, 28, 27, 18]
```

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

Runtime Comparison

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

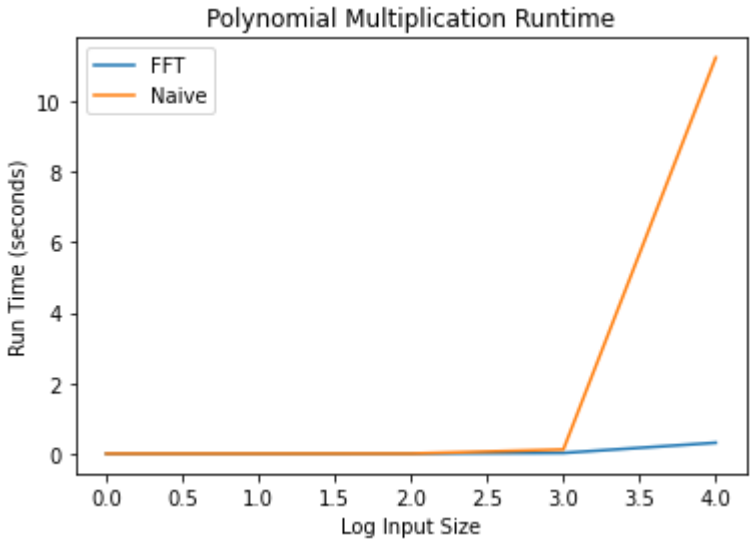
In [23]:
```
def poly_multiply_naive(coeffs1, coeffs2):
    n1, n2 = len(coeffs1), len(coeffs2)
    n = n1 + n2 - 1
    prod_coeffs = [0] * n
    for deg in range(n):
        for i in range(max(0, deg + 1 - n2), min(n1, deg + 1)):
            prod_coeffs[deg] += coeffs1[i] * coeffs2[deg - i]
    return prod_coeffs
```

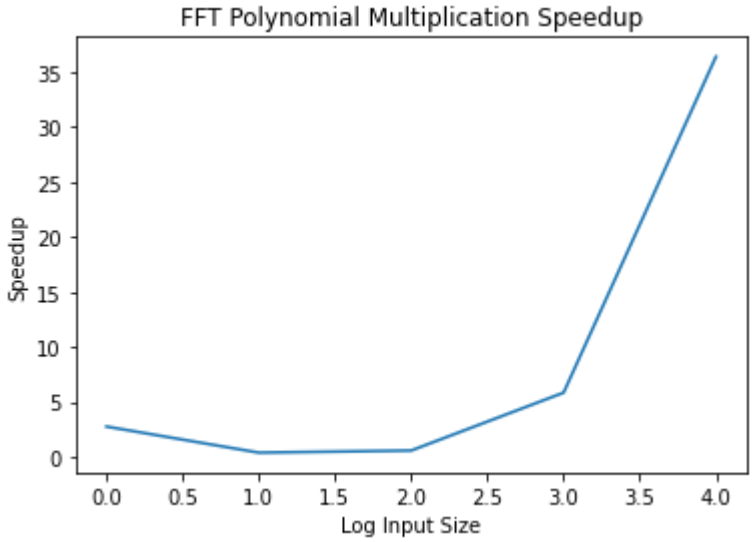
Running the following cell, you should see FFT perform similarly to or worse than the naive algorithm on small inputs, but perform significantly better once inputs are sufficiently large, which should be apparent by how long you have to wait for the naive algorithm to finish on the largest input (you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

In [24]:
```
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 = round_complex_to_int(fft_res)
    zero_pop(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")
```

```
size: 1
FFT time:   0.000024
naive time: 0.000065
speed up: 2.767677

size: 10
FFT time:   0.000246
naive time: 0.000098
speed up: 0.398642

size: 100
FFT time:   0.002132
naive time: 0.001261
speed up: 0.591123

size: 1000
FFT time:   0.020056
naive time: 0.117303
speed up: 5.848684

size: 10000
FFT time:   0.309103
naive time: 11.242984
speed up: 36.372993
```

Out[24]: `Text(0.5, 1.0, 'FFT Polynomial Multiplication Speedup')`




FFT Polynomial Multiplication Speedup
Speedup
Log Input Size
0
5
10
15
20
25
30
35
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

In [ ]: