In this final lecture, we will see an algorithm that can multiply two polynomials of degree d in $O(d \log d)$ time. It is based on the famous *Fast Fourier Transform* algorithm, and combines classic ideas from algorithm design (e.g., divide-and-conquer) with algebraic techniques (polynomials) and math (complex numbers) in an extraordinarily cool way.

Objectives of this lecture

In this lecture, we will

- review some math that we will need today, including polynomials and complex numbers
- derive the *Fast Fourier Transform* algorithm and use it for multiplying polynomials
- see some applications of fast polynomial multiplication

1 Preliminaries

1.1 Polynomials

Recall that a polynomial of degree d is a function p that looks like

$$p(x) \coloneqq \sum_{i=0}^{d} c_i x^i = c_d x^d + c_{d-1} x^{d-1} + \ldots + c_1 x + c_0$$

A polynomial of degree d can be described by a vector of its coefficients $\langle c_d, c_{d-1}, \ldots, c_1, c_0 \rangle$. According the *unique reconstruction theorem*, a polynomial can also be uniquely described by its values at $d+1$ distinct points $x_0, \ldots, x_d$.

Polynomials can be multiplied to yield another polynomial. The product of a degree n and degree m polynomial is a polynomial of degree $n+m$. Let $A(x)$ and $B(x)$ be two polynomials of degree d

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d, \\ B(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d, \end{aligned}$$

then their product is the polynomial $C(x)$:

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d},$$

where

$$c_k = \sum_{\substack{0 \le i,j \le k \\ i+j=k}} a_i \cdot b_j = \sum_{0 \le i \le k} a_i \cdot b_{k-i}.$$

Computing the product C directly via the definition would take $O(d^2)$ time assuming we can perform all of the necessary arithmetic operations on the coefficients in constant time. Karatsuba's algorithm can improve this to $O(d^{1.58})$. Today, we will see how to do it even faster.

1.2 Complex numbers and roots of unity

The field of *complex numbers* consists of numbers of the form

$$a + bi$$

where $a, b \in \mathbb{R}$ and i is the special *imaginary unit*, which is defined to be the solution to the equation $i^2 = -1$. Complex numbers are very useful for analyzing the solutions to polynomials, since every polynomial equation has a solution over the complex numbers, even though it might not have any real-valued solution.

Roots of unity A *root of unity* is a fancy way of saying an n^{th} root of 1 for some value of n. that is, a complex number ω is an n^{th} root of unity if it satisfies

$$\omega^n = 1.$$

There are exactly n complex n^{th} roots of unity, which can be written as

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

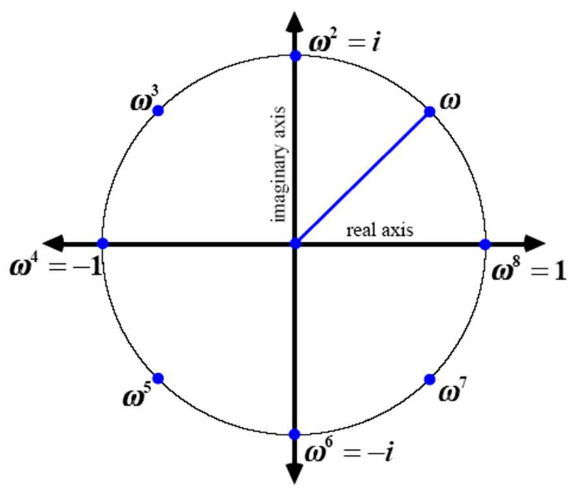
Observe the useful fact that

$$e^{\frac{2\pi ik}{n}} = \left(e^{\frac{2\pi i}{n}}\right)^k,$$

i.e., the roots of unity can all be defined as powers of the $k = 1^{\text{st}}$ one. We call this one a *primitive* n^{th} *root of unity.* More specifically, ω is a primitive n^{th} root of unity if

$$\begin{aligned} \omega^n &= 1 \\ \omega^j &\neq 1 \qquad \text{for } 0 < j < n \end{aligned}$$

The following figure shows the eighth roots of unity. As the figure suggests, in general, the n^{th} roots of unity are always equally spaced around the unit circle.



This graphical depiction also shows us a very useful property of the roots of unity.

Lemma: Halving lemma

For any even $n \geq 0$, the squares of the complex n^{th} roots of unit are precisely the $(n/2)^{\text{th}}$ roots of unity.

Exercise

Prove the halving lemma algebraically using the definition of the n^{th} roots of unity.

2 Polynomial Multiplication: The High Level Idea

By default, we usually represent polynomials in the coefficient representation, but we recall that if we know the value of a degree d polynomial at $d+1$ distinct points, that uniquely determines the polynomial. Not only that, but if we had A and B in this point-value representation, we could, in $O(d)$ time compute the polynomial C in that same representation: simply multiply the values of A and B at the specified points together. This is much faster than multiplying polynomials directly via the coefficient representation which takes $O(d^2)$ time.

This leads to the following outline of an algorithm for this problem:

> Let $N = 2d + 1$ so the degree of C is less than N.
> (1) Pick N points $x_0, \ldots, x_{N-1}$ according to a secret formula.
> (2) Evaluate $A(x_0), \ldots, A(x_{N-1})$ and $B(x_0), \ldots, B(x_{N-1})$.
> (3) Now compute $C(x_0), \ldots, C(x_{N-1})$ where $C(x) = A(x)B(x)$.
> (4) Interpolate to get the coefficients of C.

The reason we like this is that multiplying is easy in the "value on N points" representation. So step 3 is only $O(N)$ time.

If we ignore steps (2) and (4), this algorithm just takes $O(N) = O(d)$ time. However, the best algorithm that we know so far to perform those steps each take $O(d^2)$ time: To compute the point representation, we could use Horner's rule $2d + 1$ times, and to interpolate, we can use Lagrangian interpolation in $O(d^2)$ time.

We therefore shift the goalposts to finding a fast algorithm for performing steps (2) (4). Note that we have some flexibility in this framework. In order to compute the product, it is sufficient to evaluate the polynomial at *any* N points $x_0, \ldots x_{N-1}$, so perhaps there is some clever choice of points that will make the algorithm faster than just using any arbitrary set of points (indeed there is and this is the magic of the algorithm!)

Let's focus on forward direction first. In that case, we've reduced our problem to the following:

> **GOAL:** Given a polynomial A of degree $< N$, evaluate A at N points of our choosing in total time $O(N \log N)$. Assume N is a power of 2.

3 To Point-Value Form (The Fast Fourier Transform)

First here's a little intuition for how this is going to work. Consider the case where the degree of A is 7, and we want to evaluate it at two points: 1 and -1.

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7.$$

So:

$$\begin{aligned} A(1) &= a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 \\ A(-1) &= a_0 - a_1 + a_2 - a_3 + a_4 - a_5 + a_6 - a_7 \end{aligned}$$

These two computations take a total of 14 additions.

Suppose instead we were to compute $Z = a_0 + a_2 + a_4 + a_6$ and $W = a_1 + a_3 + a_5 + a_7$. Now $A(1) = Z + W$ and $A(-1) = Z - W$. This can be done in a total of only 8 additions. This optimization may not seem like much, but if we can figure out how to apply it recursively, it solves the problem – it gets us from $O(N^2)$ down to $O(N \log N)$.

Making it recursive To make the above idea recursive, the first step is to keep everything in terms of polynomials rather than evaluate immediately and just end up with numbers. What's the polynomial equivalent of our even and odd expressions (Z and W)? Lets say we just take those even and odd coefficients and use them to write smaller (half as big) polynomials!

$$\begin{aligned} A_{\text{even}}(x) &= a_0 + a_2x + a_4x^2 + a^6x^3, \\ A_{\text{odd}}(x) &= a_1 + a_3x + a_5x^2 + a_7x^3 \end{aligned}$$

Now the important question is how to recombine the smaller polynomials to get the larger one? Note that we basically halved all of the powers when we took the smaller polynomial, so to get back the original powers, we need to *square* them. The odd powers are then one higher, so we can multiply by x to recover those. This gives us the following important formula:

$$A(x) = A_{\text{even}}(x^2) + xA_{\text{odd}}(x^2).$$

So, to make our algorithm recursive, we want to split the polynomial into even and odd polynomials of half as many terms, divide-and-conquer on those and then combine the two halves back together using the above formula! The major question that remains is how do we choose the x values to use? 1 and -1 seemed like good choices above because they work when the polynomial is cut into odd and even, but its not obvious how to generalize that to N points.

3.1 Selecting the best points

The key insight into selecting the right points is to to keep the x^2 part of the formula in mind. When we are recombining the subproblems, the points are no longer the same set of points (but rather the squares of the previous points), so that presents a challenge.

In general, if we start with a set of N points, then take all of their squares, we get back a new set of N points. This is not very helpful since our divide-and-conquer will perform $O(N)$ work at every subproblem and therefore still take $O(N^2)$ time. Somehow, we need a magic set of points such that if we start with N of them, then square them all, we get $N/2$ points, since then the divide-and-conquer will correctly reduce the problem size at each level.

Do we know any special set of numbers that have this property??? Yes, we talked about them in the beginning of the lecture! We can use ***roots of unity*** as our set of points. When we take the square of the N N^{th} roots of unity, we get the $N/2$ $(N/2)^{\text{th}}$ roots of unity, which is exactly what we need to make our algorithm efficient!

These are the special magic numbers at which we will evaluate our polynomial. We will write A as a polynomial of degree $N-1$:

$$A(x) = a_0 + a_1x + a_2x_2 + \cdots + a_{N-1}x^{N-1},$$

and then define the DFT (Discrete Fourier Transform) of the coefficient vector $(a_0, \ldots, a_{N-1})$ to be another vector of N numbers as follows:

$$F_N(a_0, a_1, \ldots, a_{N-1}) = (A(\omega^0), A(\omega^1), \ldots, A(\omega^{N-1}))$$

That is, we are just evaluating A at the N^{th} roots of unity.

3.2 The Fast Fourier Transform algorithm

Now we can derive the fast algorithm for computing the DFT. (This is called the FFT algorithm.) Let A be the vector $(a_0, a_1, \ldots, a_{N-1})$. Let $F_N(A)_j$ denote the jth component of the DFT of the vector A.

$$\begin{aligned}
F_N(A)_j &= A(\omega^j) \\
&= \sum_{i=0}^{N-1} a_i\omega^{ij} \\
&= \sum_{\substack{i\,=\,0 \\ i \text{ even}}}^{N-1} a_i\omega^{ij} + \sum_{\substack{i\,=\,1 \\ i \text{ odd}}}^{N-1} a_i\omega^{ij} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i}\omega^{2ij} + \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1}\omega^{(2i+1)j} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i}(\omega^2)^{ij} + \omega^j \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1}(\omega^2)^{ij} \\
&= \sum_{i=0}^{\frac{N}{2}-1} a_{2i}(\omega_{N/2})^{i(j \bmod N/2)} + \omega_N^j \sum_{i=0}^{\frac{N}{2}-1} a_{2i+1}(\omega_{N/2})^{i(j \bmod N/2)}
\end{aligned}$$

In the last step we've simply used the fact that $\omega_N^2 = \omega_{N/2}$, and the observation that $\omega_{N/2}^{ij} = (\omega_{N/2}^j)^i = (\omega_{N/2}^{j \bmod N/2})^i$ because $\omega_{N/2}^j$ is a periodic funciton of j with period $N/2$.

Now the key point is that these two summations are in fact just DFTs half the size. Let's let A_{even} denote the vector of as with even subsubscripts (of length $N/2$) and A_{odd} be the odd ones. We can write the final equation above using these vectors as follows:

$$F_N(A)_j = F_{N/2}(A_{\text{even}})_{j \bmod N/2} + \omega_N^j F_{N/2}(A_{\text{odd}})_{j \bmod N/2}$$

Notice that in this derivation we did use the fact that ω is a root of unity, but we *never used* the fact that it is a primitive root of unity. That will be needed when we compute the inverse.

So we can rewrite the above recurrence as pseudocode as follows:

Algorithm: Fast Fourier Transform

FFT$([a_0, \ldots, a_{N-1}], \omega, N)$
  if $N = 1$ **then return** $[a_0]$
  $F_{\text{even}} \leftarrow$ **FFT**$([a_0, a_2, \ldots, a_{N-2}], \omega^2, N/2)$
  $F_{\text{odd}} \leftarrow$ **FFT**$([a_1, a_3, \ldots, a_{N-1}], \omega^2, N/2)$
  $F \leftarrow$ a new vector of length N
  $x \leftarrow 1$
  for $j = 0$ to $N - 1$ **do**
    $F[j] \leftarrow F_{\text{even}}[j \mod (N/2)] + x * F_{\text{odd}}[j \bmod (N/2)]$
    $x \leftarrow x * w$
  return F

Since at each recursive call we have a polynomial of half the size and evaluate on half of many points, the runtime of this algorithm follows the recurrence:

$$T(N) = 2T(N/2) + O(N),$$

which solves to $O(N \log N)$.

4 The Inverse of the DFT

Remember, we started all this by saying that we were going to multiply two polynomials A and B by evaluating each at a special set of N points (which we can now do in time $O(N \log N)$), then multiply the values point-wise to get C evaluated at all these points (in $O(N)$ time) but then we need to interpolate back to get the coefficients. In other words, we're doing $F_N^{-1}(F_N(A) \cdot F_N(B))$.

So, we need to compute F_N^{-1}. We'll develop this now.

First, we can view the forward computation of the FFT (evaluating A at $1, \omega, \omega^2, \ldots, \omega^{N-1}$) as a matrix-vector product:

$$\begin{pmatrix} \omega^{0\cdot 0} & \omega^{0\cdot 1} & \cdots & \omega^{0\cdot(N-1)} \\ \omega^{1\cdot 0} & \omega^{1\cdot 1} & \cdots & \omega^{1\cdot(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^{(N-1)\cdot 0} & \omega^{(N-1)\cdot 1} & \cdots & \omega^{(N-1)\cdot(N-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{N-1} \end{pmatrix} = \begin{pmatrix} A(\omega^0) \\ A(\omega^1) \\ \vdots \\ A(\omega^{N-1}) \end{pmatrix}$$

Note that the matrix on the left is the one where the contents of the kth row and the jth column is ω^{kj}. Let's call this matrix $\mathbf{DFT}(\omega, N)$. What we need is $\mathbf{DFT}(\omega, N)^{-1}$.

Let's see what happens if we multiply $\mathbf{DFT}(\omega^{-1}, N)$ by $\mathbf{DFT}(\omega, N)$.

$$\text{The } (k, j) \text{ entry of } \mathbf{DFT}(\omega^{-1}, N)\ \mathbf{DFT}(\omega, N) = \sum_{s=0}^{N-1} \omega^{-ks}\omega^{sj}$$

So let's try to evaluate the summation on the right in the two cases of $k = j$ and $k \neq j$.

If $k = j$ then we get:

$$\sum_{s=0}^{N-1} \omega^{-js}\omega^{sj} \quad = \quad \sum_{s=0}^{N-1} \omega^{0} \quad = \quad N$$

If $k \neq j$ then we get:

$$\sum_{s=0}^{N-1} \omega^{-ks}\omega^{sj} \quad = \quad \sum_{s=0}^{N-1} \omega^{(j-k)\cdot s} \quad = \quad \sum_{s=0}^{N-1} (\omega^{j-k})^{s}$$

Now let's make use of the fact that ω is a primitive root of unity. This means that $\omega^{j-k} \neq 1$. The sum is now a geometric series. So we can use the standard formula for summing a geometric series:

$$1 + r + r^2 + \cdots + r^{N-1} = \frac{1 - r^N}{1 - r}$$

So

$$\sum_{s=0}^{N-1} (\omega^{j-k})^{s} = \frac{1 - (\omega^{j-k})^N}{1 - \omega^{j-k}} = \frac{1 - 1}{1 - \omega^{j-k}} = 0$$

So, summarizing what we just learned:

$$\text{The } (k, j) \text{ entry of } \mathbf{DFT}(\omega^{-1}, N)\ \mathbf{DFT}(\omega, N) = \begin{cases} N & \text{if } k = j \\ 0 & \text{if } k \neq j \end{cases}$$

This is just the identity matrix times N. So what we've just proven is that:

$$\mathbf{DFT}(\omega, N)^{-1} = \frac{1}{N}\mathbf{DFT}(\omega^{-1}, N)$$

This means that we can compute the inverse of the DFT by using the FFT algorithm described above except running it with ω^{-1} instead of ω, and then multiplying the result by $1/N$. It still runs in $O(N \log N)$ time. Putting this all together we have an algorithm to multiply polynomials, or compute the convolution in $O(N \log N)$ time.

5 Applications of the FFT

5.1 Counting 2-sums

Given two lists of length n, called $a = \langle a_1, \dots, a_n \rangle$ and $b = \langle b_1, \dots, b_n \rangle$, we would like to compute the number of ways to obtain each possible sum of one element from each, i.e., the number of ways to obtain each element from the set

$$c = \{a_i + b_j \text{ for all } i, j\}$$

For this problem, we will assume that all of the elements of a and b are non-negative integers bounded by some number N. We would like an algorithm that computes the set c that is efficient with respect to N.

1. **A slow algorithm:** Just try all $O(n^2)$ pairs of numbers and compute their sum, then count these results in an array. This takes $O(n^2)$ time and $O(N)$ space.

2. **A faster algorithm:** Let's see if we can speed this up. We want to answer the problem: for each number $k \leq 2N$, *how many ways* can we write k as a sum of an element $i \in a$ and an element $j \in b$? Let $A[i]$ denote the number of occurrences of the element i in a, and $B[j]$ denote the number of occurrences of j in B. The answer is

$$C[k] = \sum_{\substack{0 \leq i,j \leq n \\ i+j=k}} A[i] \cdot B[j]$$

This operation turns out to be so useful and common that it has a name: the vector C is called the *convolution* of the vectors A and B.

Definition: Discrete convolution

Given two vectors A and B, their *dicrete convolution* C is a vector where

$$C[k] = \sum_{\substack{0 \leq i,j \leq n \\ i+j=k}} A[i] \cdot B[j].$$

But hold on... this formula looks awfully familiar. That's just the formula for the coefficient of the product of two polynomials!!!

Theorem: Computing discrete convolutions fast

Given two vectors A and B of length n, we can compute their discrete convolution in $O(n \log n)$ time by treating A and B as the coefficients of polynomials and computing their product using the FFT-based polynomial multiplication algorithm.

This therefore allows us to solve the counting 2-sums problem in $O(N \log N)$ time. Just write down the count vectors A and B in $O(N)$ time, then compute their convolution via the FFT-based polynomial multiplication algorithm. The coefficients of the result are the answer!

6 More applications of FFT

Optional content — Will not appear on the homeworks or the exams

6.1 Cyclic Convolutions

Consider these two vectors:

$$\begin{aligned} a &= [1,1,1,1,0,0,0,0] \\ b &= [1,1,1,1,0,0,0,0] \end{aligned}$$

their convolution is:

$$c = [1,2,3,4,3,2,1,0]$$

And this is precisely what you'd expect if you multiplied the corresponding polynomials together. But a natural question to ask is: what would the result be if there were some non-zeros in all those zero positions of a and b? What is the algorithm doing with those? Here's an example of it.

$$\begin{aligned} a &= [1,2,3,4,0,0,0,0] \\ b &= [0,0,0,0,0,1,0,0] \\ c &= [4,0,0,0,0,1,2,3] \end{aligned}$$

You can see that it just wraps the results around modulo N. This is called the *cyclic convolution*. And it's defined as follows:

$$c_k = \sum_{\substack{0 \le i,j < N \\ i+j \bmod N = k}} a_i * b_j = \sum_{0 \le i < N} a_i * b_{(k-i) \bmod N}$$

So this is what our FFT-based polynomial multiplication algorithm is really doing with the two input vectors.

Here are some other examples of cyclic convolutions.

```
A =     1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
B =     1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
conv A B =    1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0


A =     1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
B =     0  0  0  0  0  0  0  1  0  0  0  0  0  0  0  0
conv A B =    0  0  0  0  0  0  0  1  2  3  4  5  6  7  8  0


A =     1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
B =     0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0
conv A B =    4  5  6  7  8  0  0  0  0  0  0  0  0  1  2  3
```

```
A =    1  2  3  4  5  6  7  8  0  0  0  0  0  0  0  0
B =    0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0
conv A B =    7   8   0   0   0   1   2   3   4   5   7   9  11   4   5   6


A =    1  2  3  4  5  6  7  8  9 10 11 12  0  0  0  0
B =    0  0  0  0  0  1  0  0  0  0  1  0  0  0  0  0
conv A B =   19   8   9  10  11  13   2   3   4   5   7   9  11  13  15  17


A =    1  0  1  1  0  0  1  1  1  1  1  0  1  0  0  0
B =    1  1  0  1  0  0  0  0  0  0  0  0  0  0  0  0
conv A B =    1   1   1   3   1   1   2   2   2   3   3   2   2   2   0   1
```

6.2 Digital Filtering

Suppose you wanted to simulate what it would sound like to be in a dungeon (or concert hall, or a forest, or anywhere else). You want to take a recorded sound and transform it to what it sounds like in some other environment. Here's one way to go about doing that.

You could go to the dungeon and measure what is called an *impulse response* of the environment. This is done by generating a loud very short bang (the impulse) and recording for the next several seconds what comes back (the response) as a result of that impulse.

The input to this problem is a signal of length n, which is a sequence of numbers, $S = s_0, s_1, \ldots, s_{n-1}$. Let the response also be a sequence of n numbers (pad it out if it's too short), and call it $R = r_0, r_1, \ldots, r_{n-1}$. Now the convolution of the two sequences is

$$c_k = \sum_{\substack{0 \le i,j \le k \\ i+j=k}} s_i * r_j$$

(A picture is needed here.)

So to apply our algorithm to compute this, we would let N be the next power of two greater than $2n-2$. Then we would pad both of the vectors S and R with zeros, then we would compute the convolution using the FFT technique.

Making a practial digital filter for applications is much more complicated. But at the core of these algorithms is an FFT-based convolution algorithm. This is needed to make it run fast enough.

6.3 String Matching

1. Suppose we are given a text $t = t_0 t_1 \ldots t_{n-1}$ and a pattern $p = p_0 p_1 \ldots p_{m-1}$. Find all occurrences of p in t in $O(n \log n)$ time using FFT.

   Let $X[i] = \sum_{j=0}^{m-1} (t_{i+j} - p_j)^2$ for all $0 \le i < n-m$. There is a match at index i if and only if $X[i] = 0$. Now $X[i] = \sum_{j=0}^{m-1} (t_{i+j} - p_j)^2 = \sum_{j=0}^{m-1} (p_j^2 - 2p_j t_{i+j} + t_{i+j}^2)$.

So define $Y[i] = \sum_{j=0}^{m-1} p_j t_{i+j}$ for all $0 \le i < n-m$. It remains to calculate Y.

Let t' be the reverse of t. Then

$$Y[i] = \sum_{j=0}^{m-1} p_j t_{i+j} = \sum_{j=0}^{m-1} p_j t'_{n-1-i+j}$$

Let $k = n-1-i-j$. Then

$$Y[i] = \sum_{j+k=n-1-i} p_j t'_k$$

This can now be calculated using FFT. Let f be the polynomial with coefficients $p_1, p_2, \ldots, p_{m-1}$ and g be the polynomial with coefficients $t'_0, t'_1, \ldots, t'_{n-1}$. Then $Y[i]$ is the coefficient of x^{n-1-i} in $f \cdot g$.

2. Now suppose p contains wildcard characters, which can match any character t. Finds all matches of p in t in $O(n \log n)$ time.

   Replace each wildcard with 0. Let

   $$X[i] = \sum_{j=0}^{m-1} t_{i+j} p_j (t_{i+j} - p_j)^2$$

   Then there is a match at index i if and only if $X[i] = 0$. The rest is the same as in part (a).

   Specifically, suppose we want to calculate $Y[i] = \sum_{j=0}^{m-1} p_j^a t_{i+j}^b$ for some constants a and b. Let f be the polynomial with coefficients $p_1^a, p_2^a, \ldots, p_{m-1}^a$ and g be the polynomial with coefficients $t'^b_0, t'^b_1, \ldots, t'^b_{n-1}$. Then $Y[i]$ is the coefficient of x^{n-1-i} in $f \cdot g$.

3. Suppose p contains no wildcards. Compute, in $O(n \log n)$ time, for each index $0 \le i < n-m$, the number of characters that agree between p and $t[i, i+m-1]$.

   Let's solve an easier version of this problem, where for each i, we want to calculate the number of agreements between p and $t[i, i+m-1]$ where both letters are 'a'.

   Replace each letter in p and t with 1 if it is an 'a', and 0 otherwise.

   Again define $Y[i] = \sum_{j=0}^{m-1} p_j t_{i+j}$ for all $0 \le i < n-m$. Clearly $Y[i]$ is the value we want for index i. Y can be computed with FFT as in part a.

   Now repeat for each of the other 25 letters of the alphabet. Sum together all Y arrays to obtain the final answer.